1

Tecnologia informatica e livelli di astrazione

L'avanzamento della civiltà
è segnato dall'aumentare del numerò
di azioni significative che possiamo
compiere senza pensare.

Alfred Nort Whitehead
An Introduction to Mathematics, 1911

1.1 Introduzione

Benvenuti! È un piacere avere l'opportunità di trasmettere l'entusiasmo presente nel mondo dei sistemi di elaborazione. Questo non è un campo arido e noioso, in cui il progresso è glaciale e le nuove idee sono trascurate e si atrofizzano; i sistemi di elaborazione hanno una relazione vitale e sinergica con un comparto industriale di proporzioni rilevanti (responsabile per il 5-10% del prodotto nazionale lordo degli Stati Uniti). Una caratteristica atipica di queste industrie è di accogliere l'innovazione ad una velocità mozzafiato: dal 1985 sono comparsi una mezza dozzina di nuovi elaboratori, che potevano rivoluzionare l'industria dei calcolatori, e la rivoluzione iniziata da ciascuno di questi fu interrotta solamente perché qualcun altro costruì modelli ancora migliori.

La gara all'innovazione ha favorito un progresso senza precedenti fin dai tempi dei primi calcolatori nei tardi anni '40. Se l'industria dei trasporti avesse tenuto il passo di quella dei calcolatori, oggigiorno potremmo, per esempio, viaggiare da una costa all'altra degli Stati Uniti in 5 secondi con 1000 lire. Se ci si concede un momento di riflessione per valutare quanto questo miglioramento cambierebbe la società (si pensi che si potrebbe vivere a Tahiti e lavorare a San Francisco, o andare fino a Mosca per una serata al balletto Bolshoi), si potranno apprezzare le implicazioni di un tale cambiamento.

I calcolatori hanno determinato una terza rivoluzione della civiltà, la rivoluzione dell'informazione, che ha preso posto a fianco della rivoluzione agricola e di quella industriale. La conseguente moltiplicazione della capacità intellettiva dell'umanità ha altresì influenzato le scienze: vi è ora un nuovo filone di investigazione scientifica, con scienziati informatici che collaborano con scienziati teorici e sperimentali per esplorare le nuove frontiere dell'astronomia, della biologia, della chimica, della fisica e così via.

La rivoluzione informatica continua e, ogni volta che il costo della capacità di elaborazione si riduce di un fattore 10, si moltiplicano le opportunità per i calcolatori ed applicazioni che non erano economicamente sostenibili divengono improvvisamente percorribili. A titolo di esempio, nel passato recente si consideravano come "fantascienza informatica" applicazioni come le seguenti:

- *Sportelli bancari automatici (tipo Bancomat)*: un calcolatore incassato nei muri delle banche per distribuire contante era un concetto ridicolo negli anni '50, quando il calcolatore più economico costava circa un miliardo di lire ed aveva le dimensioni di un'auto.
- *Calcolatori nelle automobili*: finché i microprocessori non migliorarono drasticamente, nei primi anni '80, dal punto di vista del costo e delle prestazioni, un controllo delle auto tramite calcolatore era ridicolo. Oggigiorno, i calcolatori sono utilizzati per ridurre l'inquinamento aumentando l'efficienza del carburante per mezzo di controlli sul motore; altre applicazioni, nel campo della sicurezza, sono la prevenzione dei pericolosi slittamenti delle ruote durante le frenate e il controllo degli *air bag* per proteggere gli occupanti di una vettura.
- *Calcolatori portatili*: chi avrebbe mai sognato che i progressi nei sistemi di calcolo avrebbero portato a calcolatori portatili, permettendo ad esempio agli studenti di portare il calcolatore nei luoghi di ritrovo e sui treni?

- *Progetto genoma umano*: il costo delle attrezzature di calcolo che verranno utilizzate per mappare le sequenze di DNA umano arriverà a centinaia di miliardi di lire. Difficilmente qualcuno avrebbe potuto prendere in considerazione questo progetto quando i costi di elaborazione erano 10 o 100 volte superiori, come 10 o 20 anni fa.
- *Rete estesa a livello mondiale (WWW, World Wide Web)*: non ancora esistente al tempo della stesura della prima edizione di questo libro, il WWW sta ora trasformando la nostra società. Tra i suoi molteplici usi si annoverano la distribuzione di notizie, la spedizione di omaggi floreali, l'acquisto tramite cataloghi *on-line*, i viaggi virtuali per scegliere meglio le località per le vacanze, il reperimento di persone che condividono interessi esoterici, senza dimenticare azioni più pratiche quali il consultare le dispense degli autori dei libri di testo.

Chiaramente, i progressi compiuti in questa tecnologia influenzano ora quasi ogni aspetto della nostra società; i progressi nell'hardware hanno permesso ai programmatori di creare applicazioni software estremamente utili, il che spiega l'onnipresenza dei calcolatori. La fantascienza informatica di domani prevede una società senza banconote, autostrade intelligenti automatizzate e la reale ubiquità dei sistemi di calcolo: nessuno dovrà portare con sé dei calcolatori perché questi saranno già presenti ovunque.

I programmatori si sono sempre dovuti preoccupare delle prestazioni dei loro programmi poiché fornire velocemente i risultati all'utente è un elemento critico ai fini della qualità del software. Negli anni '60 e '70, un vincolo fondamentale per le prestazioni era dato dalla dimensione della memoria del calcolatore, pertanto i programmatori seguirono spesso un semplice credo: minimizzare l'occupazione di memoria per rendere i programmi più veloci. Nell'ultimo decennio, i progressi nel progetto dei calcolatori hanno notevolmente ridotto l'importanza di una ridotta occupazione di memoria e i programmatori interessati alle prestazioni devono ora comprendere gli elementi che hanno reso obsoleto il semplice modello di memoria degli anni '60: l'organizzazione gerarchica delle memorie e la natura parallela dei processori. I programmatori che mirano a costruire versioni competitive di compilatori, sistemi operativi, basi di dati ed anche programmi applicativi, dovranno quindi avere una maggior conoscenza dell'organizzazione del calcolatore.

È un onore per gli autori avere l'opportunità di spiegare cosa vi è all'interno di questa macchina rivoluzionaria, svelando il software celato in un programma e l'hardware nascosto all'interno di un calcolatore. Quando il lettore avrà finito questo testo, comprenderà i segreti della programmazione di un calcolatore nella sua lingua naturale, l'organizzazione interna dei calcolatori e come questa influenzi le prestazioni dei suoi programmi, e addirittura come impostare il progetto di un nuovo calcolatore.

Il primo capitolo pone le basi per il resto del libro, introducendo le idee e le definizioni fondamentali, ponendo in prospettiva le componenti principali del software e dell'hardware e descrivendo i circuiti integrati, che costituiscono la tecnologia che alimenta la rivoluzione informatica.

1.2 Dietro un programma

A Parigi, quando parlavo in francese, le persone mi fissavano ad occhi sbarrati; non sono mai riuscito a far comprendere a quegli idioti la loro stessa lingua.

Mark Twain, *The Innocents Abroad*, 1869

Comunicare con un sistema elettronico significa inviare dei segnali elettrici. I segnali che le macchine comprendono meglio sono quelli di on e di off, cioè di acceso e spento, e di conseguenza il loro alfabeto è composto da due soli elementi; come nell'alfabeto italiano le 21 lettere non rappresentano un limite alle informazioni che possono essere scritte, allo stesso modo le due lettere utilizzate dai calcolatori non limitano le loro funzioni. I due simboli che rappresentano queste lettere sono i numeri 0 e 1 e il linguaggio delle macchine viene comunemente definito numerazione in base 2, o numerazione binaria; ciascuna "lettera" prende il nome di *cifra binaria* o *bit* (dall'inglese binary digit). I calcolatori obbediscono ciecamente ai comandi che vengono loro impartiti: un comando è detto *istruzione*. Le istruzioni, che sono semplicemente collezioni di bit che il calcolatore è in grado di comprendere, possono essere interpretate come numeri. Ad esempio la sequenza di bit:

```
1000110010100000
```

dice al calcolatore di sommare due numeri. Il Capitolo 3 spiega perché vengono utilizzati numeri per rappresentare sia le istruzioni che i dati; qui non si vuole anticipare troppo, ma il concetto di rappresentare istruzioni e dati con numeri è alla base della scienza dei calcolatori.

I primi programmatori comunicavano con il calcolatore utilizzando direttamente i numeri binari, ma il procedimento era estremamente noioso e così presto inventarono nuove notazioni più vicine al linguaggio dell'uomo. All'inizio tali notazioni erano tradotte a mano in numeri binari, ma il procedimento era ancora faticoso; fu con l'aiuto delle stesse macchine che i pionieri informatici inventarono i programmi per tradurre le notazioni simboliche in numeri binari. Il primo di questi programmi fu chiamato *assemblatore*; un programma di questo tipo traduce la notazione simbolica di un'istruzione nella corrispondente notazione binaria. Ad esempio, quando il programmatore scrive

```
add A, B
```

l'assemblatore traduce questa notazione in

```
1000110010100000
```

Tale istruzione dice al calcolatore di sommare i due numeri A e B; il nome dato a questo linguaggio simbolico, usato ancora ai giorni nostri, è *assembler*.

Nonostante l'assembler usato oggi sia molto migliorato rispetto alle prime versioni, è ancora molto diverso dal modo con cui uno scienziato impartisce ordini al calcolatore per le sue simulazioni di fluidodinamica, o da quello che il contabile usa per la gestione del bilancio. Il linguaggio assembler richiede che il programmatore scriva una linea di codice per ogni istruzione che il calcolatore deve eseguire, forzandolo a organizzare il suo pensiero secondo la struttura della macchina.

Questa considerazione ispirò una semplice domanda: se per semplificare la programmazione è possibile scrivere un programma che traduce da linguaggio assembler ad istruzioni binarie, che cosa impedisce di scrivere un programma che traduca una notazione a più alto livello in linguaggio assembler?

La risposta era semplice: nulla. L'obiettivo di creare un traduttore ad alto livello era sicuramente più impegnativo di quanto realizzato con l'assembler, ma la sfida venne immediatamente accettata con entusiasmo. I programmatori di oggi devono la loro produttività, e la loro salute mentale, a questa idea. I programmi che accettano notazioni più vicine al linguaggio naturale si chiamano *compilatori* ed i linguaggi che essi *compilano* sono detti *linguaggi di programmazione ad alto livello*. Tali linguaggi consentono al programmatore di scrivere espressioni ad alto livello del tipo:


```
A + B
```

Il compilatore tradurrà questa espressione nell'istruzione assembler

```
add A, B
```

L'assemblatore a sua volta la convertirà nell'istruzione binaria che dice al calcolatore di sommare i due numeri A e B

```
1000110010100000
```

La figura 1.1 mostra la relazione fra questi programmi ed i corrispondenti linguaggi.

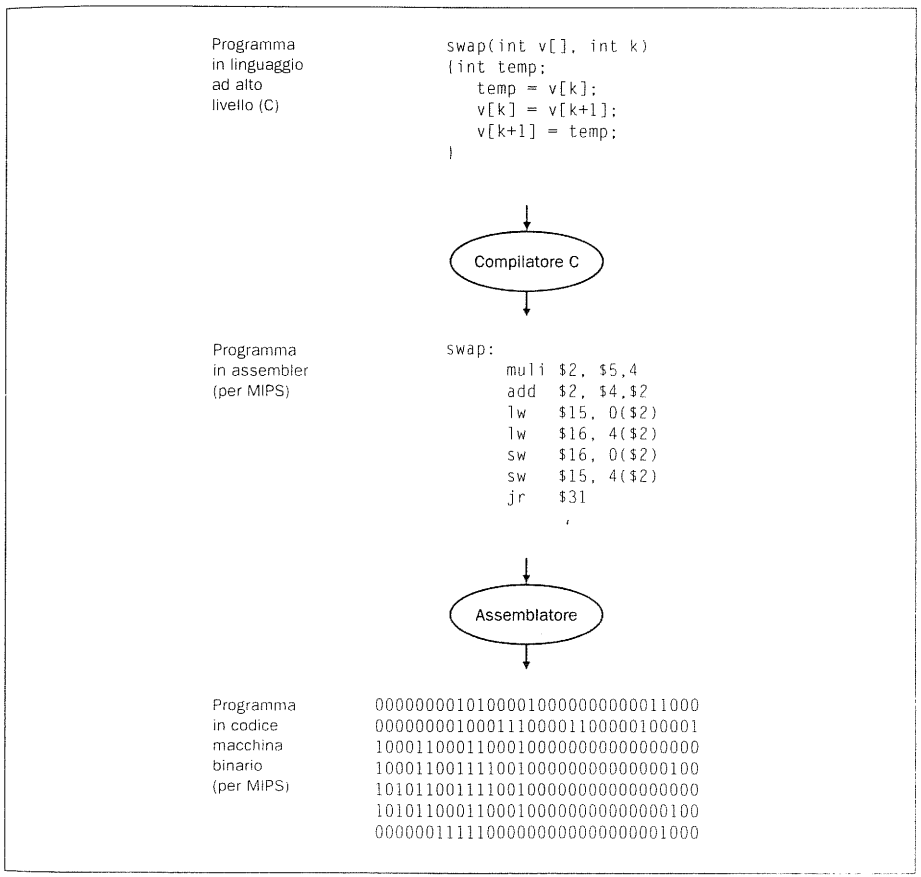


Figura 1.1 Programma in linguaggio C compilato in istruzioni assembler e quindi assemblato in codice macchina binario. Benché la traduzione da linguaggio ad alto livello a codice macchina binario venga mostrata suddivisa in due fasi, alcuni linguaggi eliminano lo stadio intermedio e producono direttamente il codice macchina. Linguaggi e compilatori di questo tipo saranno esaminati in maggiore dettaglio nel Capitolo 3 (la Web Extension III, disponibile all'indirizzo *www.mkp.com/cod2e.htm*, illustra il linguaggio C ai programmatori Pascal).

La programmazione ad alto livello offre diversi vantaggi molto importanti. Per prima cosa consente al programmatore di pensare in modo più naturale, usando parole della lingua inglese e notazioni algebriche, con il risultato che il programma assomiglia più ad un testo che a tavole piene di strani simboli misteriosi (figura 1.1). Inoltre la programmazione ad alto livello permette di progettare i linguaggi basandosi sull'uso che se ne prevede: il Fortran fu progettato per l'esecuzione di calcoli scientifici, il Cobol per applicazioni gestionali, il Lisp per la manipolazione di simboli e così via.

Il secondo vantaggio dei linguaggi di programmazione ad alto livello è l'incremento della produttività dei programmatori. Una delle poche opinioni universalmente condivise che riguardano lo sviluppo del software è che il tempo richiesto per sviluppare un programma è proporzionale al numero di linee necessarie ad esprimere una data idea nel linguaggio prescelto; i linguaggi ad alto livello, nei confronti della programmazione assembler, hanno l'evidente vantaggio della brevità .

L'ultimo vantaggio è che i linguaggi di programmazione ad alto livello, dal momento che compilatori e assemblatori possono tradurre i programmi nelle istruzioni binarie di qualsiasi macchina, fanno sì che il programma generato sia indipendente dal calcolatore per il quale viene sviluppato. I tre vantaggi sono così importanti che ai giorni nostri ben pochi programmi sono scritti in assembler.

Quando la programmazione diventò matura, molti professionisti si resero conto che riutilizzare programmi già fatti era molto più efficiente che svilupparli da zero ed i programmatori iniziarono a collaborare per creare librerie contenenti routine di largo utilizzo.

Una delle prime *librerie di subroutine* fu sviluppata allo scopo di permettere l'introduzione di sequenze di dati nel calcolatore e l'esportazione dei risultati delle elaborazioni; ad esempio, erano incluse procedure per la gestione delle stampanti, tra cui quella che controllava la presenza della carta prima dell'inizio della stampa. Questo insieme di programmi permetteva anche la gestione di altre periferiche di ingresso/uscita, come cassette o dischi magnetici, o periferiche video.

Divenne presto evidente che un insieme di programmi funzionava in modo più efficiente se un altro programma si faceva carico della supervisione: appena un programma terminava, il supervisore avviava il programma successivo in coda, evitando ritardi. Questi programmi di supervisione, che presto inclusero le librerie di ingresso/uscita, sono la base di quelli che oggi si chiamano *sistemi operativi*. I sistemi operativi sono programmi che gestiscono le risorse del calcolatore a vantaggio dei programmi che operano sulla macchina stessa.

Il software iniziò ad essere suddiviso in categorie secondo il suo uso. Il software che fornisce servizi di comune utilità è chiamato *software di sistema*; i sistemi operativi, i compilatori e gli assemblatori sono esempi di software di sistema. In contrapposizione ai programmi orientati al programmatore, il nome di *software applicativo*, o semplicemente *applicazione*, è dato ai programmi orientati agli utenti, come i fogli elettronici o gli elaboratori di testo. La figura 1.2 mostra lo schema classico dei livelli gerarchici di hardware e software.

Questo modello semplificato può generare qualche problema: all'interno della figura 1.2 è davvero corretto collocare i compilatori nel livello "software di sistema"? I compilatori producono codice impiegato sia a livello di software di sistema sia a livello applicativo e le applicazioni solitamente non generano chiamate ai compilatori quando sono in esecuzione; un'organizzazione più realistica del software è mostrata in figura 1.3. Così come guardando attentamente le fibre che

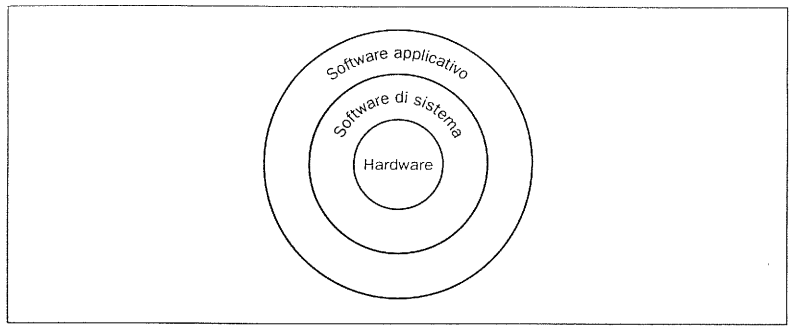


Figura 1.2 Una vista semplificata di hardware e software strutturati in livelli gerarchici secondo lo schema classico degli anelli concentrici; lo schema si sviluppa a partire dal nucleo, che rappresenta l'hardware, fino all'anello più esterno, che rappresenta il software più vicino all'utente.

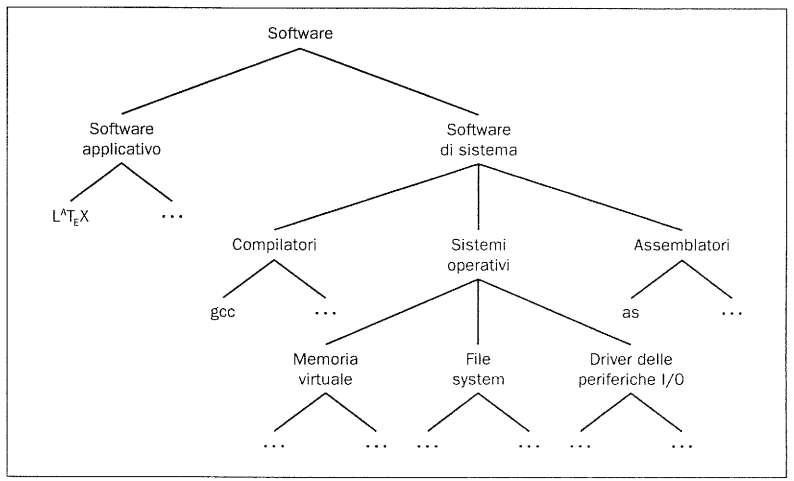


Figura 1.3 Esempio di organizzazione gerarchica del software per un sistema di calcolo. I termini al centro del grafico, come LaTeX e gcc, sono esempi di programmi Unix. I termini in basso, come memoria virtuale, saranno introdotti nei Capitoli 7 e 8.

compongono una corda si può notare che sono a loro volta costituite da fibre più sottili, allo stesso modo il software non consiste di strati monolitici, ma è composto da molti programmi costruiti uno sull'altro.

1.3 Dentro la scatola

Dopo aver dato un'occhiata ai vostri programmi per scoprire cosa ci sia dietro, apriremo la "scatola" del calcolatore per imparare qualcosa sull'hardware sottostante.

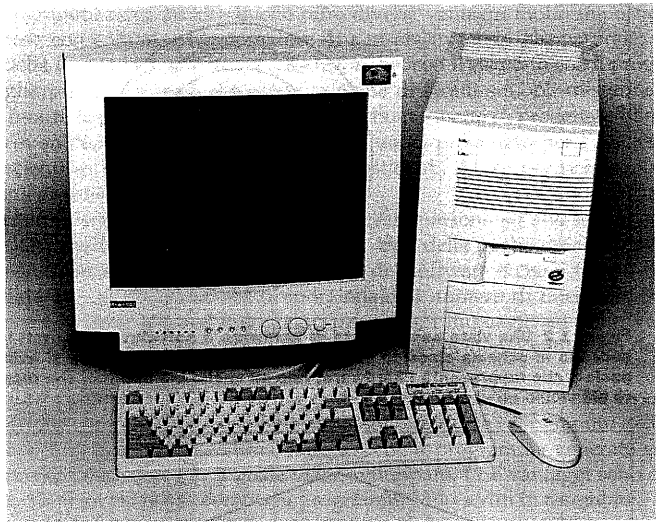


Figura 1.4 Un calcolatore da tavolo. Lo schermo a raggi catodici (CRT, *Cathod Ray Tube*) è il principale dispositivo di uscita, mentre la tastiera e il mouse sono i principali dispositivi di ingres-

La figura 1.4 mostra un tipico calcolatore da tavolo con tastiera, mouse, schermo ed un contenitore al cui interno è presente altro hardware. Ciò che non è visibile nella fotografia è la rete che collega il calcolatore alle stampanti e ai dischi. Questa fotografia rivela due componenti chiave dei calcolatori: i *dispositivi di ingresso*, come la tastiera ed il mouse, ed i *dispositivi di uscita*, come lo schermo e le stampanti. Come suggeriscono i nomi, il flusso di ingresso fornisce dati al calcolatore, mentre quello di uscita corrisponde al risultato dell'elaborazione inviato all'utente. Alcuni dispositivi, come le reti ed i dischi, gestiscono sia un flusso di ingresso che uno di uscita. Il capitolo 8 descriverà in maggior dettaglio i dispositivi di ingresso ed uscita (I/O, *Input/Output*), mentre qui ci si limiterà a presentare una panoramica introduttiva sull'hardware del calcolatore, partendo dai dispositivi esterni di I/O.

Anatomia di un mouse

L'idea del mouse mi è venuta mentre ascoltavo una presentazione ad una conferenza sui calcolatori. Il relatore era così noioso che cominciai a sognare ad occhi aperti ed ebbi l'intuizione.

Doug Engelbart

Benché molti utenti considerino oggi i mouse come qualcosa di scontato, l'idea di un dispositivo di puntamento come il mouse fu elaborata solo trent'anni or sono. Engelbart presentò per la prima volta un sistema dotato di mouse su un prototipo di ricerca nel 1967. Il sistema denominato *Alto* ispirò tutte le workstation e i Macintosh successivi e già nel 1973 includeva un mouse come dispositivo di puntamento. Entro la fine degli anni '80, tutte le workstation e molti personal computer includevano tale dispositivo, mentre divenivano popolari nuove interfacce utente

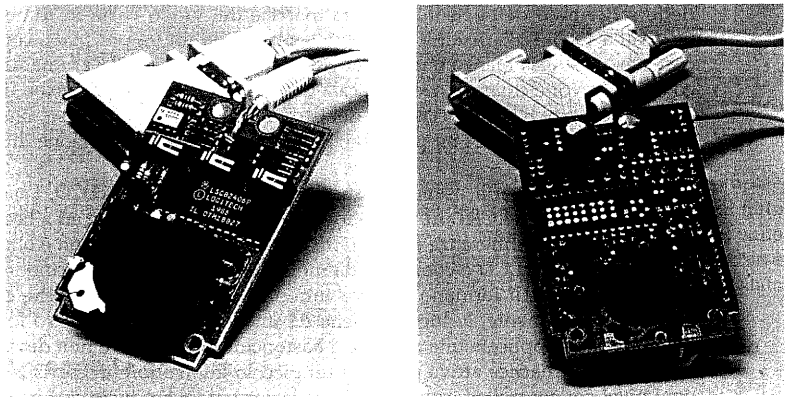


Figura 1.5 Dentro un mouse. (Per gentile concessione di Logitech.)

basate su video grafici e mouse. Il mouse è veramente un oggetto assai semplice, come mostra la fotografia nella figura 1.5.

La versione meccanica consiste di una grossa sfera montata in maniera tale da fare contatto con un paio di rotelle, una posizionata lungo l'asse X e l'altra lungo l'asse Y. Queste rotelle sono connesse a contatori meccanici, oppure fanno muovere una ruota dotata di fessure attraverso la quale un diodo a emissione di luce (LED, *Light-Emitting Diode*) invia un raggio ad un sensore luminoso. In entrambe le soluzioni i movimenti del mouse fanno ruotare la grossa sfera, che fa muovere la rotella sull'asse X o quella sull'asse Y, o ambedue, a seconda che il mouse sia spostato in direzione verticale, orizzontale o diagonale. Benché ci siano molte modalità di gestione per questi dispositivi di puntamento, nella sostanza gli spostamenti di ciascuna rotella incrementano o decrementano alcuni contatori posti da qualche parte nel sistema. I contatori servono a registrare di quanto il mouse sia stato spostato ed in quale direzione.

Attraverso il vetro

Attraverso il video di un calcolatore ho fatto atterrare un aeroplano sul ponte di una portaerei in movimento, ho osservato una particella nucleare colpire una buca di potenziale, ho volato su un razzo quasi alla velocità della luce e ho visto un calcolatore rivelare i suoi meccanismi più interni.

Ivan Sutherland, il padre della grafica al calcolatore, citato in "Computer Software for Graphics," *Scientific American*, 1984

Il dispositivo di I/O più affascinante è probabilmente il video grafico. Sfruttando la stessa tecnologia del televisore, un *video a scansione basato su un tubo a raggi catodici* (CRT, *Cathode Ray Tube*) scandisce un'immagine una linea alla volta, da 30 a 75 volte al secondo (figura 1.6). A questa *frequenza di rinfresco* (*refresh rate*) non si percepisce alcuno sfarfallio sullo schermo.

L'immagine è composta da una matrice di elementi, o *pixel*, che può essere rappresentata come una matrice di bit, chiamata *bit map*. A seconda della dimensione

dello schermo e della risoluzione, la matrice del video varia in dimensioni da 512x340 a 1560x1280 pixel. Il video più semplice ha un bit per pixel, consentendo così che questo sia bianco o nero; nei video che supportano 256 sfumature di bianco e nero, a volte chiamati video a scala di grigi, vi sono 8 bit per pixel. Un video a colori può usare 8 bit per ciascuno dei tre colori fondamentali (rosso, blu e verde), per un equivalente di 24 bit per pixel, permettendo la visualizzazione di milioni di colori diversi.

I calcolatori portatili usano spesso video a cristalli liquidi (LCD, *Liquid Crystal Display*) al posto dei CRT. In tal modo si ottiene un video sottile ed a basso consumo. La principale differenza sta nel fatto che il pixel di un LCD non è una sorgente di luce. Un tipico LCD è composto da molecole a forma di bastoncino immerse in un liquido, le quali formano un'elica intrecciata che devia la luce che colpisce lo schermo. Tipicamente, la luce proviene da una sorgente posta dietro lo schermo. Quando viene applicata una corrente, i bastoncini si allineano e non deviano più la luce. L'LCD a matrice attiva possiede un piccolo interruttore in corrispondenza di ogni pixel per controllare con accuratezza la corrente e produrre così immagini più nitide.

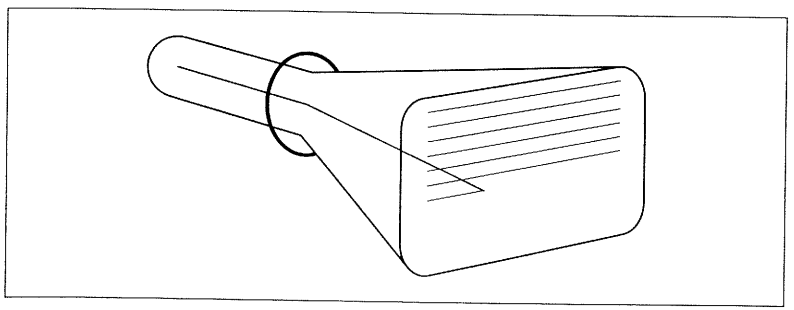


Figura 1.6 Un video CRT. Un raggio viene sparato nel vuoto da un cannone elettronico verso uno schermo rivestito di fosfori. La bobina di controllo posta sul collo del CRT provvede al puntamento del cannone. I sistemi di scansione raster, usati nei televisori ed in quasi tutti i calcolatori, tracciano l'immagine sullo schermo (composta da una serie di punti o pixel) una linea per volta. Lo schermo è aggiornato da 30 a 70 volte al secondo.

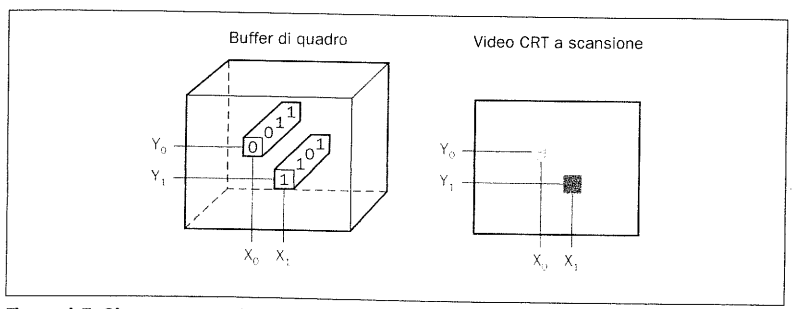


Figura 1.7 Ciascuna coordinata nel buffer di quadro a sinistra determina la sfumatura della corrispondente coordinata per il video CRT a scansione posto a destra. Il pixel (X_0, Y_0) contiene il vettore di bit 0011, che corrisponde sullo schermo ad una sfumatura di grigio più chiara del vettore di bit 1101 del pixel (X_1, Y_1).

Indipendentemente dal video, il supporto hardware per la grafica in un calcolatore consiste principalmente in un *buffer di aggiornamento della scansione*, o *buffer di quadro* (*frame buffer*), che memorizza la bit map. L'immagine che deve essere rappresentata sullo schermo è memorizzata nel buffer di quadro e, per ciascun pixel, il corrispondente vettore di bit è inviato all'elettronica di controllo del video grafico alla frequenza di aggiornamento. La figura 1.7 mostra un buffer di quadro con 4 bit per pixel.

Lo scopo della bit map è quello di rappresentare fedelmente ciò che compare sullo schermo. Le difficoltà nella realizzazione dei sistemi grafici nascono dal fatto che l'occhio umano è molto sensibile nel rilevare cambiamenti anche minimi sullo schermo; per esempio, quando è in corso l'aggiornamento dello schermo, l'occhio è in grado di rilevare l'incongruenza tra la porzione dello schermo che è già stata modificata e quella che non lo è ancora stata.

Aprendo la scatola

Aprendo la scatola che contiene il calcolatore, si può vedere una scheda di sottile plastica verde coperta da dozzine di piccoli rettangoli grigi o neri. La figura 1.8 mostra il contenuto dello stesso calcolatore da tavolo della figura 1.4; la *scheda madre* (*motherboard*) compare verticalmente sul fondo, con un'unità per la gestione del floppy disk e l'alimentatore sulla sinistra.

I piccoli rettangoli sulla scheda madre contengono dei dispositivi tecnologicamente molto avanzati, denominati *circuiti integrati* o *chip*. La scheda è composta da tre parti: la parte di connessione con i dispositivi di I/O menzionati precedentemente, la memoria ed il processore. I dispositivi di I/O sono collegati attraverso le due ampie schede attaccate perpendicolarmente alla scheda madre, all'incirca a metà della parte di destra.

La *memoria* è il luogo nel quale i programmi sono memorizzati durante la loro esecuzione; essa contiene inoltre i dati necessari ai programmi in esecuzione. Nella figura 1.8 la memoria si trova sulle due piccole schede attaccate perpendicolarmente a metà della scheda madre; ciascuna scheda di memoria contiene otto circuiti integrati.

Il *processore* è la parte attiva della scheda, quella che esegue fedelmente le istruzioni di un programma: somma numeri, esegue test su numeri, segnala ai dispositivi di I/O di attivarsi e così via. Il processore è il grande oggetto quadrato sotto le schede di memoria nell'angolo in basso a destra della figura 1.8. A volte il processore viene chiamato CPU, abbreviazione di *Central Processing Unit*.

Scendendo ancora di livello all'interno dell'hardware, la figura 1.9 rivela alcuni dettagli sul processore della figura 1.8. Il processore comprende due componenti principali: l'unità di elaborazione dei dati (o semplicemente unità di elaborazione) e l'unità di controllo, rispettivamente il braccio e la mente del processore. L'unità di elaborazione dei dati (*datapath*) esegue le operazioni aritmetiche, mentre l'unità di controllo indica all'unità di elaborazione, alla memoria e ai dispositivi di I/O che cosa fare a seconda delle istruzioni del programma. Il Capitolo 5 illustra l'unità di elaborazione dei dati e quella di controllo in un sistema di ridotta complessità, mentre il Capitolo 6 descrive le modifiche necessarie per ottenere un progetto ad alte prestazioni.

A questo punto sono state identificate le componenti più importanti di un calcolatore. Quando in questo libro verrà trattato un punto importante, così importante da sperare che il lettore lo ricordi per sempre, questo verrà evidenziato denomi-

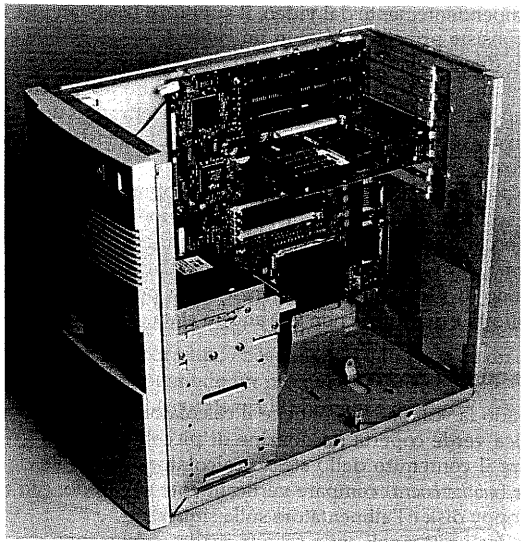


Figura 1.8 L'interno di un personal computer. La scheda verticale sul fondo è una scheda a circuiti stampati (PCB, *Printed Circuit Board*), che in un PC si chiama *scheda madre* (*motherboard*), la quale contiene gran parte dell'elettronica del calcolatore; la figura 1.11 è una fotografia di insieme di tale scheda, ruotata di 90 gradi. Il processore corrisponde al grosso rettangolo nero nell'angolo in basso a destra della scheda. La figura 1.19 è una fotografia del processore prima che sia posizionato nel package nero. Le due grandi schede attaccate perpendicolarmente nella parte alta della scheda sulla destra contengono le intefacce di I/O verso la rete locale Ethernet ed una scheda video per un CRT. Le due piccole schede attaccate perpendicolarmente a metà della scheda madre contengono i chip di memoria. Il grosso contenitore in basso a sinistra contiene l'alimentatore e al di sopra si trovano un'unità per la gestione dell'hard disk ed una per la gestione del floppy disk.

nandolo "*Quadro di insieme*". Vi è circa una dozzina di quadri di insieme in questo libro, ed il primo è relativo alle cinque componenti di un calcolatore.

Quadro d'insieme

Le cinque componenti classiche di un calcolatore sono l'unità di ingresso, l'unità di uscita, la memoria, l'unità di elaborazione e l'unità di controllo; le ultime due sono a volte combinate e denominate processore. La figura 1.10 mostra l'organizzazione tipica di un calcolatore; questa organizzazione è indipendente dalla tecnologia hardware utilizzata. È possibile associare una qualsiasi parte di un qualunque calcolatore, passato o presente, ad una di queste cinque categorie. Per aiutare il lettore a ricordare tutto ciò, le cinque componenti di un calcolatore sono riportate sulla pagina iniziale dei prossimi capitoli mettendo in evidenza la parte trattata nel capitolo.

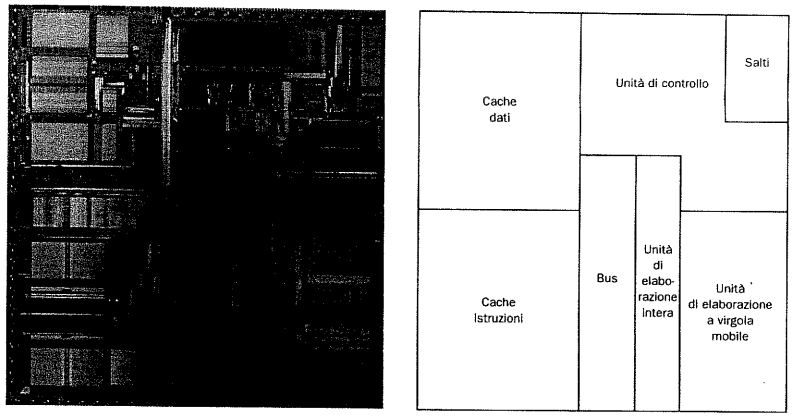


Figura 1.9 L'interno del chip del processore della scheda mostrata nella figura 1.8. Questo chip è un Pentium della Intel: la foto a sinistra è un primo piano del chip, mentre il disegno a destra identifica i blocchi più importanti. L'area del die è 91 mm² e contiene circa 3,3 milioni di transistor. La memoria cache occupa quasi un milione di transistor; il Capitolo 7 spiega perché vengano destinate alle cache tante risorse. Altri componenti del chip sono descritti nei capitoli successivi: la tecnica di predizione dei salti è trattata nel Capitolo 6, i bus nel Capitolo 8 . Fotografia gentilmente concessa da Intel.

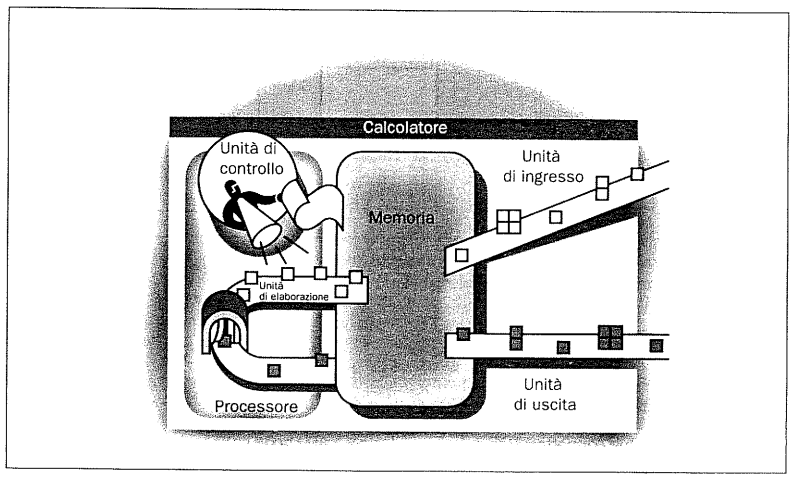
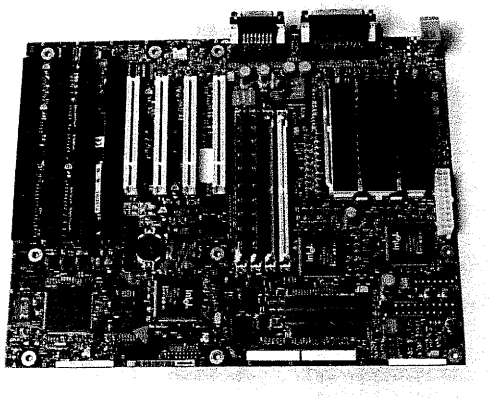


Figura 1.10 L'organizzazione di un calcolatore con le cinque componenti classiche. Il processore riceve le istruzioni e i dati dalla memoria; l'unità di ingresso scrive i dati nella memoria e l'unità di uscita legge i dati dalla memoria. L'unità di controllo invia i segnali che determinano le operazioni svolte dall'unità di elaborazione, dalla memoria, dall'unità di ingresso e da quella di uscita.

Scendendo nei dettagli di ciascuna componente dell'hardware si scoprono gli aspetti interni della macchina. Questa operazione è stata fatta per il processore e adesso sarà ripetuta per la memoria. La scheda della figura 1.11 contiene due tipi di



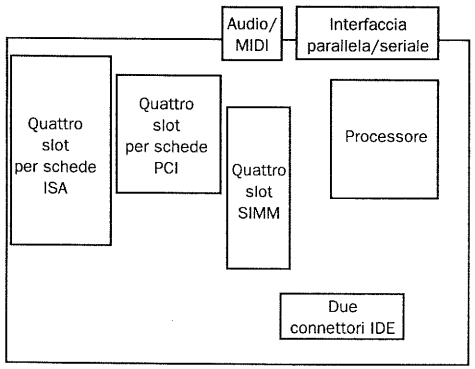


Figura 1.11 Dettaglio della scheda madre di un PC. Questa scheda usa un processore Pentium Pro della Intel, posizionato sul bordo superiore destro della scheda. Il processore è coperto da alcuni elementi in metallo nero, chiamati *dispersori di calore*, che contribuiscono a raffreddare il chip. Il package del Pentium Pro contiene delle memorie cache ad alta velocità. La memoria principale è contenuta sulle due piccole schede perpendicolari alla scheda madre al centro. I chip di DRAM sono montati su queste schede (chiamate *SIMM*, che sta per *Single Inline Memory Module*) che sono poi incastrate nei connettori appositi. Gran parte di ciò che resta della scheda è composto di connettori per i dispositivi esterni di I/O: le interfacce audio/MIDI e parallele/seriali sul bordo superiore, i quattro slot per schede PCI al di sotto, i quattro slot per schede ISA a sinistra e i due connettori IDE in basso a destra. Tali schede collegano la scheda principale a stampanti, altoparlanti, CRT, reti locali, dischi e così via.

memorie: la memoria DRAM e la cache. DRAM sta per *Dynamic Random Access Memory* (*Memoria Dinamica ad Accesso Casuale*); oggi si usano varie DRAM per contenere insieme le istruzioni ed i dati di un programma. La parte RAM del termine DRAM indica che contrariamente a quanto succede nelle memorie ad accesso sequenziale, come i nastri magnetici, in una DRAM gli accessi alla memoria richiedono lo stesso tempo indipendentemente dalla porzione di memoria che viene letta. La memoria *cache* consiste di una memoria piccola, ma veloce, che funge da

buffer verso la memoria DRAM. Questa è una definizione non tecnica di cache che permette di astrarre da molti dettagli specifici.

Si sarà notato un tema comune nelle descrizioni del software e dell'hardware: man mano che si scava in profondità nell'hardware e nel software si scoprono sempre maggiori informazioni e, viceversa, dettagli di livello inferiore sono nascosti al fine di offrire un modello più semplice ai livelli superiori. L'uso di tali livelli, o *astrazioni*, è la principale tecnica utilizzata nel progetto di sistemi di calcolo estremamente sofisticati.

Uno dei livelli di astrazione più importanti è quello corrispondente all'interfaccia tra l'hardware ed il software di più basso livello. A causa della sua importanza, ad essa è stato associato un nome: *architettura dell'insieme di istruzioni*, o *architettura del set di istruzioni*, o semplicemente *architettura*, dell'elaboratore. L'architettura del set di istruzioni include tutto ciò che i programmatori hanno bisogno di sapere per creare un programma in linguaggio macchina correttamente funzionante e quindi le istruzioni, i dispositivi di I/O e così via. Le componenti di un'architettura sono discusse nei Capitoli 3, 4, 7 e 8.

Questa interfaccia standard permette ai progettisti di calcolatori di parlare di funzioni indipendentemente dall'hardware che le realizza. Per esempio si può parlare delle funzioni svolte da un orologio digitale (misurare il tempo, visualizzare l'ora, programmare la sveglia) indipendentemente dall'hardware dell'orologio (cristalli di quarzo, video a LED, bottoni in plastica). I progettisti di calcolatori distinguono l'architettura dall'*implementazione* dell'architettura utilizzando lo stesso approccio: un'implementazione è una struttura fisica, un'organizzazione hardware che realizza l'astrazione prevista dall'architettura. Queste idee portano ad un altro Quadro di Insieme.

Quadro d'insieme

Sia l'hardware che il software consistono di diversi livelli gerarchici, all'interno dei quali ogni livello nasconde alcuni dettagli al livello superiore. Questo principio di astrazione è il modo in cui sia i progettisti dell'hardware che i progettisti del software affrontano la complessità dei sistemi di calcolo. Un'interfaccia chiave tra i livelli di astrazione è l'architettura del set di istruzioni (Instruction Set Architecture), ossia l'interfaccia tra l'hardware ed il software di più basso livello. Questa interfaccia astratta permette a varie implementazioni, con costo e prestazioni diversi, di eseguire lo stesso software.

Gli utenti dei sistemi Macintosh hanno ben chiaro l'impatto di un cambiamento nell'architettura del set di istruzioni: i programmi progettati per l'architettura PowerPC non sono eseguibili sulle macchine basate su 68000 ed i programmi basati su 68000 non sono completamente eseguibili dal PowerPC. Al contrario la famiglia 80x86 della Intel offre varie implementazioni della stessa architettura: i programmi scritti per il vecchio 8086 nel 1978 possono essere eseguiti sul più recente Pentium Pro. Come illustra il Capitolo 3, la Intel ha aggiunto nel corso degli anni molte novità, ma tutte le generazioni che si sono succedute sono sempre state vincolate a consentire senza problemi l'esecuzione dei vecchi programmi.

Un posto sicuro per i dati

Credo che la Silicon Valley (in italiano Valle del Silicio) abbia il nome sbagliato. Se si guarda indietro ai dollari fatturati per i vari prodotti negli ultimi dieci anni, gli introiti legati ai dischi magnetici sono stati molto maggiori di quelli derivanti dal silicio. Quel posto dovrebbe essere ribattezzato Iron Oxide Valley (in italiano Valle dell'Ossido di Ferro).

Al Hoagland, uno dei pionieri dei dischi magnetici, 1982

Fin qui si è visto come si possano acquisire dati, eseguire elaborazioni usando dati e visualizzare dati. Tuttavia, se si fosse sul punto di perdere l'alimentazione del calcolatore, tutto sarebbe perso, dal momento che la memoria interna al calcolatore è *volatile*; questo significa che quando cessa l'alimentazione essa dimentica tutto. Al contrario, il nastro di una cassetta per un apparecchio stereo quando viene spento non dimentica la musica registrata, dal momento che il nastro è magnetico ed usa quindi una tecnologia per le memorie di tipo *non volatile*.

Al fine di distinguere tra la memoria usata per immagazzinare i programmi mentre questi vengono eseguiti e la memoria non volatile usata per memorizzare i programmi tra un'esecuzione e l'altra, si usa il termine *memoria primaria* o *memoria principale* per la prima e *memoria secondaria* per la seconda. Le DRAM hanno dominato il settore delle memorie principali sin dal 1975, mentre i dischi magnetici hanno dominato quello delle memorie secondarie a partire dal 1965.

Esistono due tipi principali di dischi magnetici: i floppy disk e gli hard disk. Il concetto fondamentale alla base del funzionamento di questi dischi è lo stesso: un piatto rotante ricoperto da materiale magnetico in grado di registrare informazioni. La differenza principale sta nel fatto che il floppy disk è fatto di un materiale flessibile, il mylar, mentre gli hard disk sono metallici. I floppy disk possono essere rimossi e portati in giro, mentre la maggior parte degli hard disk oggi non è rimovibile. La capacità di un floppy disk varia da 1,44 MB nei floppy disk di basso costo a 100 MB per i floppy disk di tipo Zip.

Un altro supporto rimovibile è il compact disk ottico, o CD, che è meno costoso ma più lento di un hard disk. All'estremità inferiore della scala delle prestazioni stanno i nastri magnetici, usati per realizzare le copie di backup dei dischi; trovare un dato su un nastro magnetico può richiedere un tempo dell'ordine dei secondi.

Come mostra la figura 1.12, un hard disk magnetico consiste di un insieme di piatti, che ruotano attorno ad un asse ad una velocità compresa tra i 3600 ed i 7200 giri al minuto; i piatti in metallo sono coperti su ambedue le facce da materiale magnetico in grado di memorizzare informazioni, analogo a quello che si trova sui nastri delle cassette. Per leggere e scrivere le informazioni su un hard disk, poco al di sopra di ciascuna superficie è posizionato un *braccio* mobile contenente una piccola bobina elettromagnetica chiamata *testina di lettura/scrittura*. Poiché utilizzano le stesse testine e gli stessi mezzi di stabilizzazione adottati dalla tecnologia degli hard disk, le unità Zip si avvicinano più alle prestazioni ed alla capacità degli hard disk che a quelle dei floppy disk tradizionali.

Il diametro degli hard disk varia oggi di circa un fattore 5, da 1,3 a 5,25 pollici, e si è ridotto nel corso degli anni per tenere conto dei nuovi prodotti; workstation, personal computer, calcolatori portatili e calcolatori palmari hanno tutti ispirato nuove forme e dimensioni per i dischi. Tradizionalmente, i dischi più grossi hanno le migliori prestazioni, i dischi più piccoli hanno i minori costi unitari ed il miglior costo per megabyte è di solito associato a dischi di tipo intermedio.

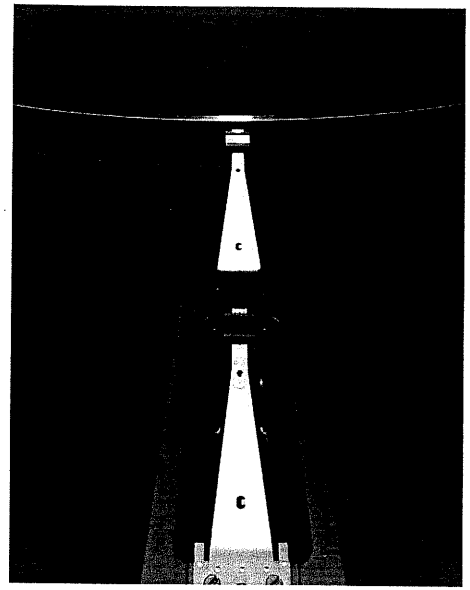


Figura 1.12 Un disco costituito da 10 piatti e dalle testine di lettura/scrittura. Fotografia gentilmente concessa da Storage Technology Corp.

L'uso di componenti meccaniche fa sì che i tempi di accesso per i dischi magnetici siano molto più alti che per le DRAM: i dischi richiedono tipicamente da 5 a 20 millisecondi, mentre le DRAM richiedono da 5 a 100 nanosecondi, il che rende le DRAM circa 100.000 volte più veloci. Tuttavia, a parità di capacità di immagazzinamento, i dischi hanno costi molto più bassi di quelli delle DRAM, dal momento che i costi di produzione per una certa quantità di memoria su disco sono minori di quelli per la stessa quantità di memoria realizzata su circuito integrato. Nel 1997 il costo per megabyte di un disco era circa 50 volte minore del costo per megabyte di una DRAM.

Riassumendo, ci sono tre principali differenze tra dischi magnetici e memoria principale: i dischi non sono volatili, dal momento che sono magnetici; hanno un tempo di accesso più alto, dal momento che sono dei dispositivi meccanici; il loro costo per megabyte è minore, dal momento che hanno un'altissima capacità di memorizzazione ad un costo modesto.

Comunicare con altri calcolatori

È stato spiegato come sia possibile acquisire, elaborare, visualizzare e salvare dati, ma manca ancora un punto tra quelli che caratterizzano i calcolatori di oggi: le reti di calcolatori. Esattamente come il processore mostrato nella figura 1.10 è collegato alla memoria ed ai dispositivi di ingresso/uscita, così le reti collegano tra loro i calcolatori, permettendo agli utenti di estendere le possibilità di calcolo attraverso le comunicazioni. Le reti sono divenute così diffuse da costituire l'ossatura degli attuali sistemi di calcolo: una nuova macchina priva di un'eventuale interfaccia di

rete sarebbe ridicola. I calcolatori connessi in rete presentano diversi significativi vantaggi:

- *Comunicazioni:* i calcolatori scambiano informazioni ad alta velocità.
- *Condivisione di risorse:* anziché avere macchine con propri dispositivi di I/O, questi possono essere condivisi dai calcolatori in rete.
- *Accesso remoto:* essendo possibile connettere i calcolatori a grande distanza, gli utenti non hanno la necessità di essere vicini al calcolatore che stanno usando.

Le reti variano in lunghezza e prestazioni ed il costo della comunicazione aumenta proporzionalmente sia alla velocità di comunicazione, sia alla distanza coperta dalle informazioni. La rete forse più diffusa è *Ethernet*; la sua lunghezza massima è limitata a circa un chilometro e la versione più diffusa richiede almeno un secondo per inviare un milione di byte di dati. La sua lunghezza e velocità rendono Ethernet adatta alla connessione di calcolatori posti sullo stesso piano di un edificio; di conseguenza essa è un esempio di quella che è generalmente chiamata una *rete locale* (*Local Area Network*). Le *reti geografiche* (*Wide Area Network*) che attraversano i continenti sono l'ossatura di Internet, che supporta il World Wide Web; esse sono tipicamente realizzate con fibre ottiche e sono noleggiate dalle società di telecomunicazioni.

1.4 I circuiti integrati: alimento per l'innovazione

Pensavo che [i calcolatori] sarebbero stati un concetto universalmente applicabile, come lo è un libro. Ma non pensavo che si sarebbero sviluppati così velocemente, perché non prevedevo che saremmo stati in grado di integrare su un chip tante parti quante in effetti riuscimmo. Il transistor comparve inatteso. Avvenne tutto molto più rapidamente di quanto ci attendessimo.

J. Presper Eckert, co-inventore dell'ENIAC, in un discorso del 1991

I progettisti di calcolatori partecipano ad una sfida continua per la realizzazione del miglior calcolatore ed adottano ogni novità della tecnologia elettronica per vincere la sfida; questo ha fatto sì che i processori e le memorie siano migliorati ad una velocità incredibile. La figura 1.13 riporta le tecnologie che sono state utilizzate in diversi periodi, con una stima delle prestazioni relative per unità di costo nelle diverse tecnologie. Questo paragrafo esplora la tecnologia che è stata alla base dell'industria dei calcolatori a partire dal 1975 e che continuerà prevedibilmente ad esserlo nel prossimo futuro; questa tecnologia pone i limiti alle potenzialità dei calcolatori e determina la loro velocità di evoluzione, per cui tutti i professionisti dell'informatica dovrebbero avere una certa familiarità con i circuiti integrati.

Anno	Tecnologia utilizzata nei calcolatori	Prestazioni relative per unità di costo
1951	Valvola	1
1965	Transistor	35
1975	Circuito integrato	900
1995	Circuito a larghissima scala di integrazione (VLSI)	2400000

Figura 1.13 Prestazioni relative per unità di costo delle tecnologie utilizzate dai calcolatori in funzione del tempo. Fonte: Computer Museum, Boston.

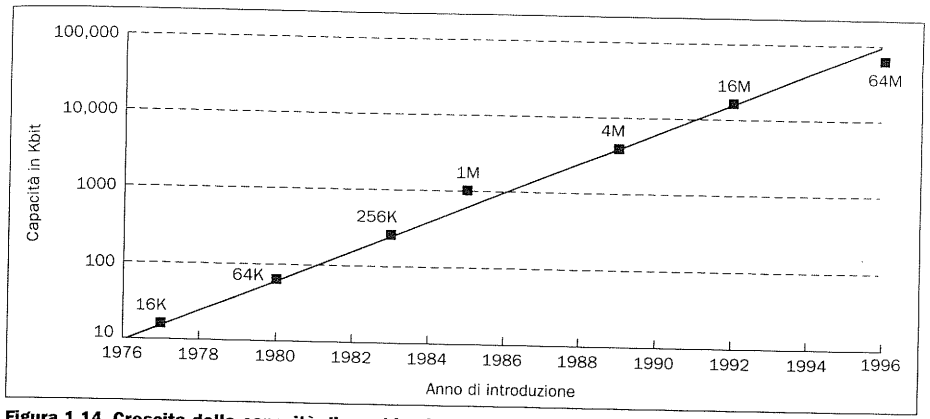


Figura 1.14 Crescita della capacità di un chip di DRAM in funzione del tempo. L'asse y è misurato in Kbit, dove K=1024 (2^{10}). L'industria delle DRAM ha quadruplicato la capacità ogni 3 anni, con un incremento del 60% annuo, per quasi 20 anni. Fece eccezione la DRAM da 1 Mbit, arrivata un anno prima del previsto; una seconda eccezione fu la DRAM da 64 Mbit, in ritardo di un anno. La regola pratica "quattro volte ogni tre anni" è detta *legge di crescita delle DRAM*.

Un *transistor* si comporta semplicemente come un interruttore, aperto/chiuso, controllato dall'elettricità. Un *circuito integrato* combina decine di migliaia di transistor in un singolo frammento di silicio (detto *chip*). Per descrivere l'enorme incremento nel numero di transistor, dalle migliaia ai milioni, si usa l'attributo *a larghissima scala*, formando il termine *circuito integrato a larghissima scala di integrazione (VLSI, Very Large-Scale Integrated circuit)*.

La crescita del tasso di integrazione è rimasta notevolmente stabile; la figura 1.14 mostra la crescita della capacità delle DRAM dal 1977. I produttori hanno puntualmente quadruplicato la capacità dei loro dispositivi ogni 3 anni, arrivando ad un incremento superiore ad un fattore 16000 in soli 20 anni! Questa notevole velocità di evoluzione in termini di costo, prestazioni e capacità dei circuiti integrati condiziona il progetto dell'hardware e del software, evidenziando la necessità di comprendere a fondo questa tecnologia.

Iniziamo dal principio. La produzione di un chip parte dal *silicio*, una sostanza che si trova nella sabbia; poiché il silicio non conduce bene l'elettricità, è detto *semiconduttore*. Con uno speciale processo chimico è possibile aggiungere dei materiali al silicio per permettere a piccole aree di comportarsi in uno di questi tre modi:

- come un eccellente conduttore elettrico (paragonabile a un filo di rame o di alluminio);
- come un eccellente isolante elettrico (paragonabile alle guaine in plastica o al vetro);
- come un'area che può condurre *oppure* non condurre sotto particolari condizioni (analogamente ad un interruttore).

I transistor rientrano nell'ultima categoria e un circuito VLSI non è altro che una combinazione di milioni di conduttori, isolanti ed interruttori prodotti all'interno di un singolo piccolo contenitore (detto *package*).

Il processo produttivo dei circuiti integrati è critico al fine di determinare il costo dei chip e quindi importante per i progettisti di calcolatori. La figura 1.15 schema-

tizza tale processo, che inizia con un lingotto di silicio cristallino, simile ad una salsiccia gigante. Attualmente i lingotti hanno un diametro di 15-30 cm e sono lunghi 30-60 cm; il lingotto è tagliato in dischi (detti *wafer*) non più spessi di 2 mm. I wafer attraversano una serie di lavorazioni, nel corso delle quali vi vengono depositate delle sostanze chimiche, secondo uno *schema* prestabilito (in inglese *pattern*), per creare transistor, conduttori e isolanti come sopra discusso.

Una singola microscopica falla nel wafer, o una sbavatura nel deposito di uno solo delle decine di schemi di materie chimiche, possono far sì che quell'area del wafer non funzioni; questi *difetti* rendono virtualmente impossibile produrre un wafer perfetto. Per gestire le imperfezioni sono state utilizzate diverse strategie, la più semplice delle quali consiste nel porre molti componenti indipendenti sullo stesso wafer; il wafer viene poi tagliato separando i diversi componenti, chiamati *die* (schegge) o più informalmente noti come *chip*. Il processo di separazione dei die permette di scartare solo quei chip che contengono i difetti, anziché l'intero wafer; l'entità di questo scarto è misurata dalla cosiddetta *resa* del processo, definita come la percentuale di die funzionanti rispetto al loro numero totale sul wafer.

Una volta identificati i chip funzionanti, questi vengono montato all'interno dei contenitori (*package*) e gli ingressi e le uscite vengono collegati ai piedini del package; il processo che collega il die al package viene detto *bonding*. I dispositivi finiti vengono poi collaudati un'ultima volta, per via dei possibili errori nel processo di bonding, e spediti ai clienti.

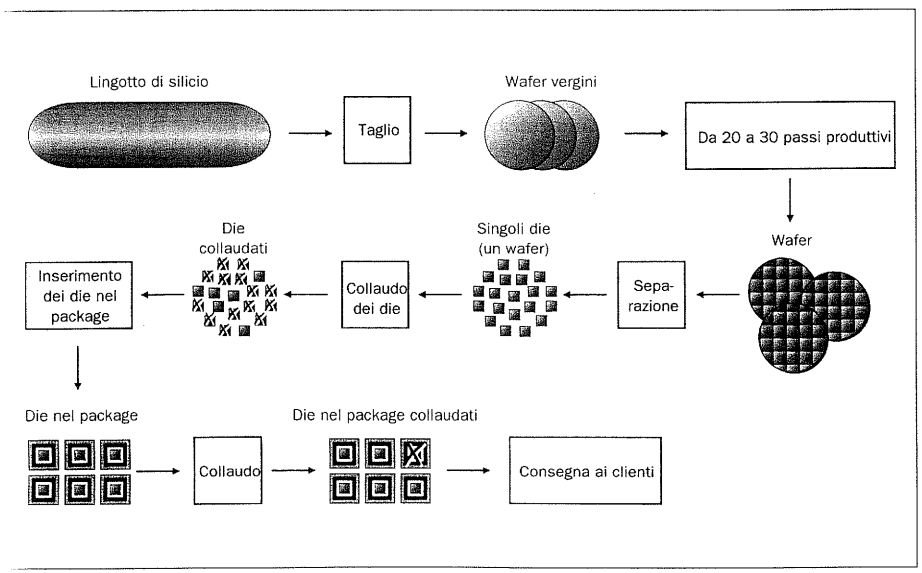


Figura 1.15 Il processo produttivo dei chip. Dopo essere stati tagliati dal lingotto di silicio, i wafer puliti subiscono da 20 a 30 passi produttivi fino a contenere i pattern voluti (si veda la figura 1.17). I wafer vengono poi tagliati in die (figura 1.18) e ciascuno di questi viene collaudato. In figura, un wafer ha prodotto 20 die, di cui solo 6 hanno passato il collaudo (una X significa che il die è difettoso); la resa è in questo caso di 6/20, o 30%. I die buoni sono poi posti nei package (figura 1.19) e ulteriormente collaudati prima della spedizione ai clienti. In questo collaudo finale è stata trovata una parte danneggiata.

Approfondimento: Occasionalmente verranno inseriti degli approfondimenti contenenti idee non essenziali ad una prima lettura, ma che potrebbero interessare lettori con conoscenze più avanzate.

Un parametro di recente interesse nel progetto di calcolatori è l'efficienza energetica. Esso è vitale non solo per i calcolatori portatili, al fine di estendere la vita delle batterie, ma anche nei calcolatori da tavolo al crescere della frequenza di clock dei chip; per esempio, il microprocessore Alpha 21264 dissipa ben 72 W a 600 MHz. La potenza dissipata può divenire un elemento limitante per il crescere delle prestazioni.

1.5 Un caso reale: la produzione dei chip Pentium

Ogni capitolo contiene un paragrafo sui "casi reali", il cui scopo è correlare i concetti descritti nel libro con le esperienze quotidiane che riguardano il calcolatore. Questi paragrafi sono sempre relativi alla tecnologia dei PC IBM e spesso anche a quella dei calcolatori Apple Macintosh. In questo capitolo i concetti sui circuiti integrati descritti nel paragrafo precedente verranno illustrati attraverso l'esempio dei chip del PC IBM.

La figura 1.16 è la fotografia di un wafer contenente più processori, corrispondenti ciascuno ad un unico chip, prima che vengano separati; comprende esemplari del chip mostrato in primo piano in figura 1.9. La figura 1.17 è la fotografia di un wafer di processori Pentium Pro, il successore del Pentium. La figura 1.18 mostra un singolo die di Pentium Pro.

Si noti che più i die sono piccoli, maggiore è il numero che può essere contenuto in un wafer: nel wafer dal diametro di 8 pollici di figura 1.16 ci sono 196 processori

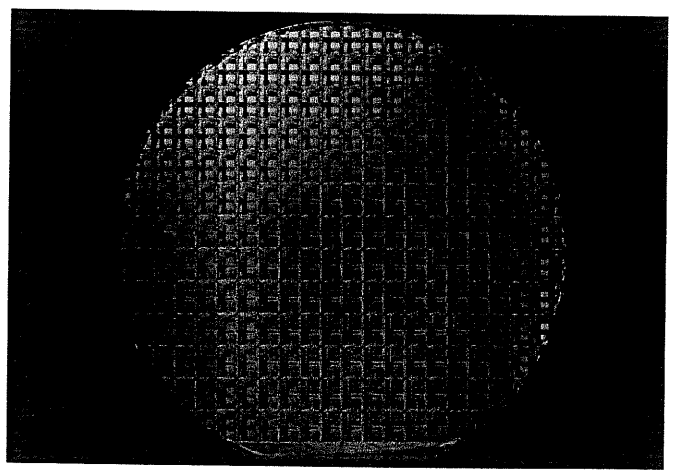


Figura 1.16 Un wafer dal diametro di 200 mm (8 pollici) contenente processori Intel Pentium. Il numero di Pentium nel wafer sfruttato al 100% è 196. L'area del die è di 91 mm² e contiene circa 3,3 milioni di transistor. La figura 1.9 è la microfotografia di uno di questi die. Le molte decine di chip parzialmente arrotondati sul bordo del wafer sono inutilizzabili e sono incluse perché in questo modo è più facile creare le maschere per la lavorazione del silicio. Fotografia gentilmente concessa da Intel.

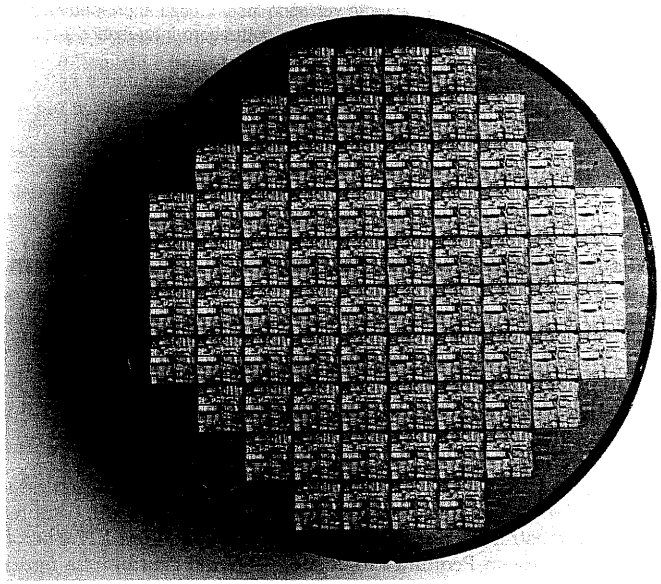


Figura 1.17 Un wafer dal diametro di 200 mm (8 pollici) contenente processori Intel Pentium Pro. Il numero di Pentium Pro nel wafer sfruttato al 100% è 78. L'area del die è di 306 mm^2 e contiene circa 5,5 milioni di transistor. La figura 1.18 è la microfotografia di uno di questi die. Wafer gentilmente concesso da Intel.

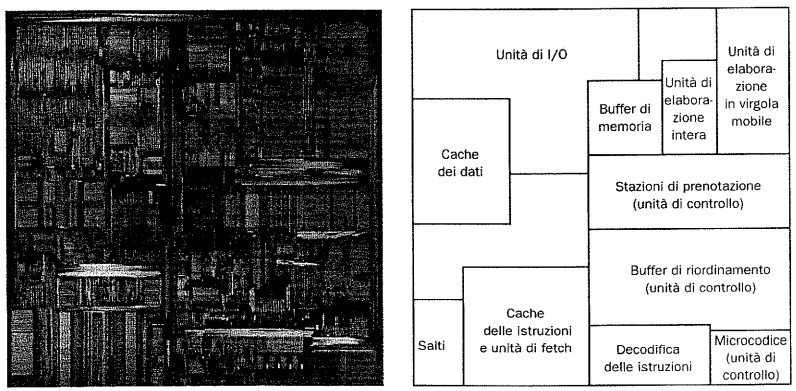


Figura 1.18 Un die di Pentium Pro: la fotografia (a sinistra) mostra il primo piano del chip, mentre il disegno (a destra) descrive i principali blocchi. L'area del die è di 306 mm^2 e contiene circa 5,5 milioni di transistor. La memoria cache è una piccola frazione del chip (solo 1 milione dei 5,5 milioni di transistor) dal momento che il Pentium Pro è inserito nel contenitore con una cache esterna da 31 milioni di transistor. I blocchi presenti all'interno del Pentium Pro sono descritti nei capitoli successivi: il microcodice nel Capitolo 5, la decodifica delle istruzioni, le stazioni di prenotazione (reservation stations), il buffer di riordinamento e la predizione dei salti nel Capitolo 6, i buffer della memoria e della cache nel Capitolo 7. Fotografia gentilmente concessa da Intel.

Pentium, mentre nel wafer di figura 1.17 ci sono soltanto 78 processori Pentium Pro, che sono più grandi del Pentium. Dal momento che il costo del wafer non dipende da quello che contiene, un minor numero di die significa costi più elevati; i costi sono ulteriormente accresciuti dal fatto che die più grandi hanno probabilità maggiori di contenere difetti, e quindi di non funzionare correttamente.

Di conseguenza il costo dei chip cresce molto rapidamente con l'area (gli esercizi da 1.46 a 1.53 entrano più in dettaglio sul costo dei chip). È ovvio che i progettisti di calcolatori debbano avere molta familiarità con la tecnologia che utilizzano, tanto da essere sicuri che i costi dovuti all'aumento dell'area dei chip siano giustificati dal miglioramento delle prestazioni.

La figura 1.19 mostra i contenitori dei due chip. Si noti che il Pentium Pro in realtà contiene due die! Invece di aumentare ulteriormente le dimensioni del die del Pentium Pro, i progettisti dell'Intel hanno deciso di costruire un secondo die: come mostrano gli esercizi, due die piccoli costano meno di uno grande e il secondo die contiene un chip di memoria cache esterna, descritto nel Capitolo 7.

I progettisti devono conoscere sia le tecnologie hardware che quelle software per essere in grado costruire calcolatori competitivi. Lavorando con il silicio, devono capire le nozioni base della costruzione dei circuiti integrati per quanto riguarda costi e prestazioni e, allo stesso modo, devono conoscere i principi relativi al software che più è legato all'hardware del calcolatore, cioè ai sistemi operativi e ai compilatori.

1.6 Errori e trabocchetti

La scienza deve iniziare dai miti e dalla critica dei miti.

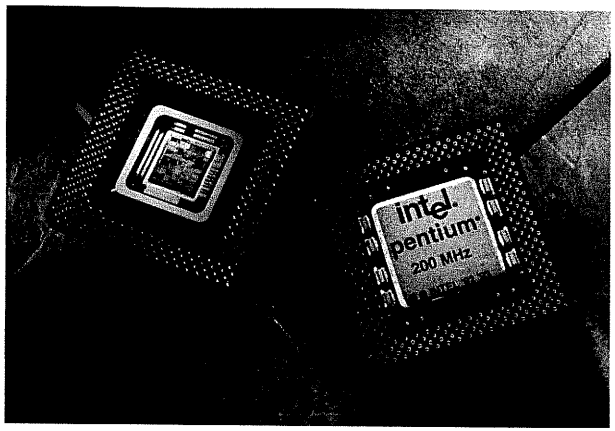
Sir Karl Popper, The Philosophy of Science, 1957

L'obiettivo di questo paragrafo, che si potrà trovare in ogni capitolo, è quello di sfatare alcune credenze sbagliate che si possono incontrare parlando di calcolatori: tali opinioni sbagliate saranno chiamate *errori* e discutendo di un errore si cercherà di fornire un controesempio. Si discuterà anche di *trabocchetti*, ossia di sbagli in cui è facile cadere; spesso i trabocchetti sono delle generalizzazioni di principi che sono veri solo in un contesto limitato. Lo scopo di questi paragrafi è quello di aiutare il lettore ad evitare di incappare in questi errori nel progetto o nell'uso delle macchine.

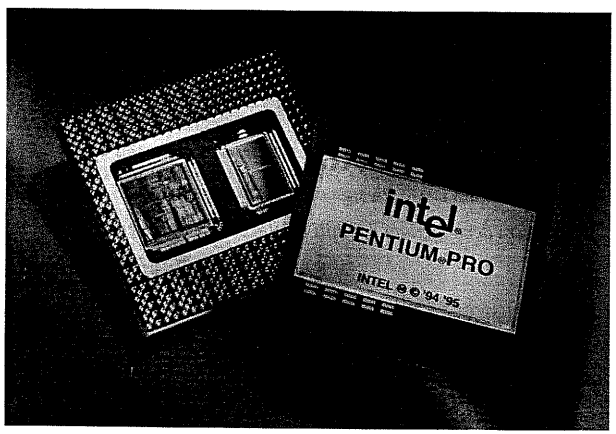
Errore: i calcolatori sono stati costruiti nello stesso modo antiquato per troppo tempo e questo modello di calcolo ormai sorpassato è giunto al termine del proprio ciclo di vita.

Per essere un modello di calcolo antiquato sta sicuramente migliorando rapidamente. La figura 1.20 mostra le prestazioni delle migliori workstation esistenti sul mercato negli anni tra il 1987 e il 1997. Il Capitolo 2 spiega il modo corretto per misurare le prestazioni. Il grafico mostra una linea che indica un miglioramento del 54% all'anno, il che corrisponde a raddoppiare le prestazioni approssimativamente ogni 18 mesi. Al contrario dell'affermazione riportata sopra, i calcolatori stanno migliorando le proprie prestazioni in maniera più veloce oggi che in ogni altro periodo della loro storia, con un miglioramento di 100 volte tra il 1987 e il 1997!

Trabocchetto: ignorare l'inesorabile progresso dell'hardware quando si pianifica un nuovo elaboratore.



a.



b.

Figura 1.19 Il contenitore del Pentium (a) e del Pentium Pro (b). Il Pentium ha 296 pin, mentre il Pentium Pro ne ha 387. I pin in più consentono un parallelismo più elevato fra il processore e la memoria centrale, con conseguente trasferimento dati più veloce e possibilità di maggiore indirizzamento verso la memoria. Il secondo chip del Pentium Pro è una memoria cache esterna (vedere la figura 7.32 per un'altra immagine). Fotografie gentilmente concesse da Intel.

Si supponga di aver deciso di introdurre un elaboratore nei prossimi tre anni e di sostenere che sarà un grande successo commerciale dal momento che è tre volte più veloce di qualunque altro disponibile oggi. Sfortunatamente, l'elaboratore sarà probabilmente un fiasco, poiché il tasso medio di crescita delle prestazioni nell'industria produrrà elaboratori con le stesse prestazioni. Ad esempio, assumendo un tasso di crescita annuale nelle prestazioni del 50% circa, un elaboratore avente oggi prestazioni pari a x avrà probabilmente prestazioni pari a $1{,}5^3x=3{,}4\ x$ tra tre anni. Il nuovo elaboratore non avrà nessun vantaggio in termini di prestazioni! Molti progetti all'interno delle ditte produttrici di calcolatori sono cancellati o perché i loro responsabili ignorano questa regola, o perché il progetto arriva al termine in ritar-

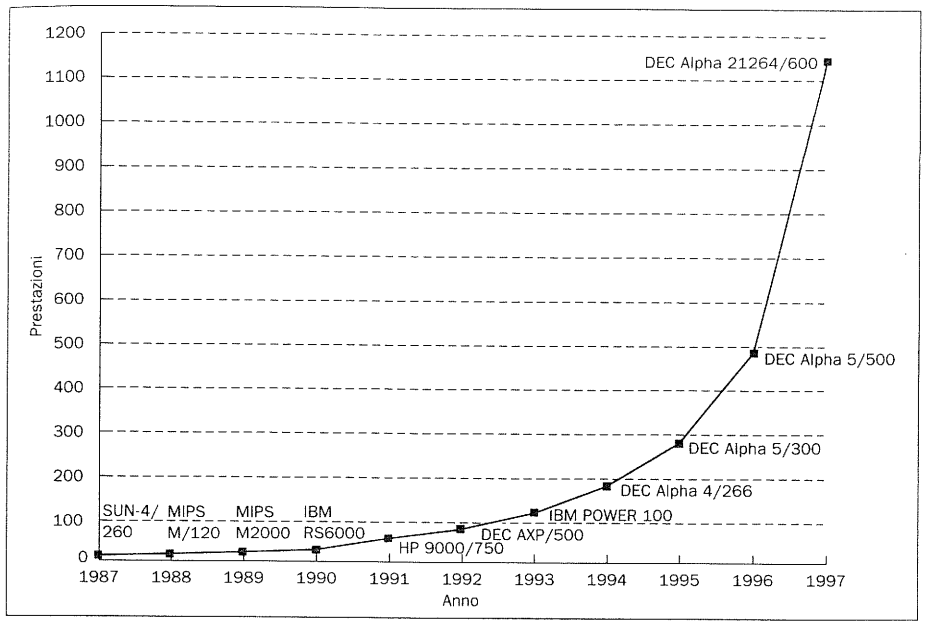


Figura 1.20 Miglioramento nelle prestazioni delle workstation negli anni 1987-1997. In questo caso le prestazioni sono date approssimativamente come rapporto di velocità rispetto al VAX-11/780, che rappresenta un termine di confronto comunemente usato. Il tasso di miglioramento nelle prestazioni è di circa 1,54 all'anno, raddoppiando quindi ogni 1,6 anni. Questi risultati sono relativi ai programmi campione per il calcolo intero SPEC92 (SPECbase_int92); si veda il capitolo 2, paragrafo 2.6, per maggiori dettagli sui programmi SPEC. Dal momento che i relativi punti nel grafico possono essere difficilmente leggibili, si riporta per ogni macchina, a partire dal 1987, il valore delle prestazioni tra parentesi: Sun-4/260 (9), MIPS M/120 (13), MIPS M2000 (18), IBM RS6000/540 (24), HP 9000/750 (51), Digital 3000 AXP/500 (80), IBM POWERstation 100 (117), Digital Alphastation 4/266 (183), Digital Alphastation 5/300 (280), Digital Alphastation 5/500 (481), e un elaboratore basato sul microprocessore a 600 MHz Alpha 21264 (1140). Gli ultimi due elaboratori sono stati valutati sui programmi SPECint95base e le loro prestazioni moltiplicate per un fattore opportuno per tener conto delle prestazioni rispetto ai programmi SPECbase92.

do e le prestazioni dell'elaboratore sono al di sotto della media. Questo fenomeno può verificarsi in qualunque campo, ma i rapidi miglioramenti in termini di rapporto costo/prestazioni lo rendono un problema cruciale nell'industria dei calcolatori.

1.7 Note conclusive

Mentre ... l'ENIAC è dotato di 18000 valvole e pesa 30 tonnellate, i calcolatori del futuro potranno avere 1000 valvole e forse peseranno solo una tonnellata e mezza.

Popular Mechanics, marzo 1949

Sebbene sia difficile prevedere esattamente il costo e le prestazioni dei calcolatori futuri, si può scommettere tranquillamente che saranno molto migliori degli attua-

li. Per giocare una parte attiva in questo progresso, i progettisti di calcolatori ed i programmatori dovranno tuttavia comprendere un più vasto spettro di problematiche.

Sia i progettisti dell'hardware che quelli del software costruiscono i sistemi di elaborazione per strati gerarchici, detti livelli di astrazione, in cui ciascuno strato nasconde dei dettagli allo strato superiore. Questo principio di astrazione è fondamentale per comprendere i calcolatori odierni, ma non significa che i progettisti si possano limitare a conoscere la tecnologia utilizzata nel solo strato di loro competenza. Forse l'esempio più importante di astrazione è l'interfaccia tra l'hardware ed il software di basso livello, detta *architettura del set di istruzioni*; se l'architettura dell'insieme di istruzioni rimane invariata, lo stesso software sarà eseguibile su molte implementazioni (presumibilmente con costo e prestazione variabili) dell'architettura stessa. D'altro canto, alcune innovazioni potrebbero richiedere un cambiamento di interfaccia, forzando quindi una variazione dell'architettura.

Le tecnologie chiave per i processori moderni sono i compilatori ed il silicio: per partecipare attivamente al progresso occorre comprendere alcune caratteristiche di entrambi. Di pari importanza rispetto alla comprensione della tecnologia dei circuiti integrati è la comprensione dei tassi di evoluzione tecnologica previsti. Un esempio di tali relazioni è il ben noto aumento di quattro volte nella capacità delle DRAM ogni tre anni. Mentre il silicio continua a sostenere una rapida evoluzione dell'hardware, nuovi criteri di organizzazione dei calcolatori ne hanno migliorato prezzo e prestazioni. Due delle idee chiave sono lo sfruttamento del parallelismo nel processore, tipicamente realizzato attraverso la tecnica del *pipelining*, e lo sfruttamento della località degli accessi alla gerarchia di memoria, tipicamente implementato mediante le memorie *cache*.

Organizzazione del testo

Alla base di questi livelli di astrazione vi sono i cinque componenti classici di un calcolatore: unità di elaborazione dati, unità di controllo, memoria, unità di ingresso e unità di uscita (figura 1.10). Queste cinque componenti servono anche come quadro di riferimento per i restanti capitoli di questo libro:

- *Unità di elaborazione dati*: Capitoli 4, 5 e 6
- *Unità di controllo*: Capitoli 5 e 6
- *Memoria*: Capitolo 7
- *Unità di ingresso*: Capitolo 8
- *Unità di uscita*: Capitolo 8

Il Capitolo 6 descrive come le pipeline nel processore si avvalgano del parallelismo, ed il Capitolo 7 descrive come le gerarchie di memoria sfruttino la località. I restanti capitoli costituiscono l'introduzione e la conclusione di questo materiale: il Capitolo 2 tratta la valutazione delle prestazioni dell'intero calcolatore. Il Capitolo 3 descrive l'architettura degli insiemi di istruzioni (l'interfaccia tra i compilatori e la macchina) ed enfatizza il ruolo dei compilatori e dei linguaggi di programmazione nell'uso delle possibilità offerte dalle istruzioni. Il Capitolo 9 conclude la trattazione con una discussione sui multiprocessori.

1.8 Inquadramento storico e approfondimenti

Una branca della scienza in piena attività è come un immenso formicaio; l'individuo quasi svanisce nella massa delle menti che turbinano, che trasmettono le informazioni da una parte all'altra e le diffondono alla velocità della luce.

Lewis Thomas, "Natural Science", in The Lives of a Cell, 1974

Un paragrafo sull'inquadramento storico chiude ciascuno dei capitoli del testo. L'intenzione è tracciare lo sviluppo di un'idea attraverso una serie di realizzazioni, descrivere progetti particolarmente importanti e fornire riferimenti bibliografici a chi è interessato ad approfondire l'argomento trattato. Questo paragrafo descrive il contesto storico in cui si sono evolute alcune delle idee chiave presentate nel capitolo di apertura; lo scopo è svelare la storia dell'uomo dietro il progresso tecnologico e collocare le conquiste all'interno di tale contesto. Conoscere il passato aiuta a capire le forze che in futuro influenzeranno il mondo dei calcolatori.

I primi calcolatori elettronici

J. Presper Eckert e John Mauchly alla Moore School della Pennsylvania University costruirono quello che è universalmente considerato come il primo esempio funzionante di calcolatore elettronico di uso generale (*general-purpose*). Questa macchina, chiamata ENIAC (Electronic Numerical Integrator and Calculator), fu finanziata dall'esercito degli Stati Uniti d'America e divenne operativa durante la Seconda Guerra Mondiale, ma la sua esistenza non fu divulgata fino al 1946. L'ENIAC era un calcolatore general-purpose usato per il calcolo delle tavole di puntamento dell'artiglieria; a forma di U, era lungo 24 metri, alto 3 e largo un paio di metri (vedere figura 1.21). Ciascuno dei 20 registri a 10 cifre misurava 60 centimetri. In totale l'ENIAC era composto da 18000 valvole.

L'ENIAC era due ordini di grandezza più grande dei calcolatori che si costruiscono ai giorni nostri, benché fosse più di quattro ordini di grandezza più lento,

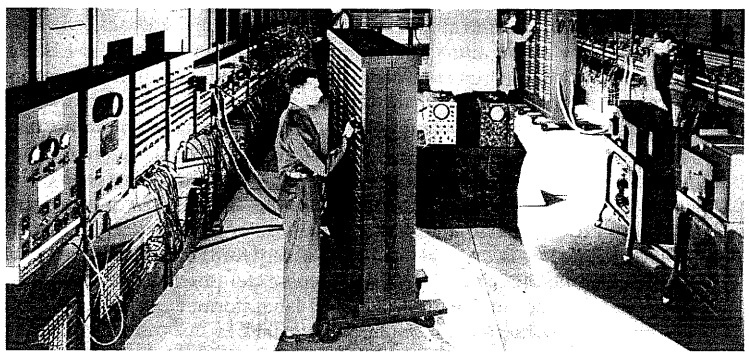


Figura 1.21 L'ENIAC, il primo calcolatore elettronico general-purpose al mondo. Si noti l'etichetta del tribunale nell'angolo in basso a destra: si riferisce alla citata causa di attribuzione del brevetto. Fotografia gentilmente concessa dall'Istituto Charles Babbage, University of Minnesota.

eseguendo 1900 somme al secondo; l'ENIAC consentiva di eseguire salti condizionati ed era programmabile, caratteristica che lo distingueva in modo netto dai calcolatori precedenti. La programmazione era fatta a mano, inserendo cavi elettrici e azionando interruttori, ed i dati erano introdotti mediante schede perforate; la programmazione di un tipico calcolo richiedeva da mezz'ora ad un giorno intero. L'ENIAC era un calcolatore general-purpose, i cui limiti erano principalmente la piccola quantità di memoria e la noiosa modalità di programmazione.

Nel 1944 John von Neumann entrò a far parte del progetto ENIAC. Lo scopo del gruppo era migliorare le modalità di programmazione dei calcolatori, discutendo della possibilità di memorizzare i programmi sotto forma di numeri; von Neumann contribuì a concretizzare le idee e scrisse un documento proponendo la costruzione di un calcolatore, l'EDVAC (Electronic Discrete Variable Automatic Computer), fornito di una memoria per mantenere sia i dati che le istruzioni. Herman Goldstine distribuì il documento a nome di von Neumann, con grande disappunto di Eckert e Mauchly, i cui nomi furono omessi. Il documento ha costituito la base di quello che oggi viene comunemente definito col nome di *calcolatore di von Neumann*. Molti pionieri dell'informatica ritengono che il termine abbia dato troppo credito a von Neumann, che espose le idee, e troppo poco agli ingegneri, Eckert e Mauchly, che lavorarono realmente sulle macchine; per questa ragione il nome non comparirà in nessun'altra parte di questo libro.

Nel 1946 Maurice Wilkes della Cambridge University visitò la Moore School per partecipare alla parte conclusiva di un ciclo di seminari sullo sviluppo dei calcolatori elettronici. Quando ritornò a Cambridge, Wilkes decise di intraprendere un progetto per costruire un calcolatore in grado di memorizzare programmi chiamato EDSAC (Electronic Delay Storage Automatic Calculator). L'EDSAC, mostrato in figura 1.22, divenne operativo nel 1949 e fu il primo calcolatore funzionante e completo con una memoria per dati e istruzioni [Wilkes 1985]. Un piccolo prototipo chiamato Mark-I, costruito alla Manchester University nel 1948, è considerato il primo calcolatore funzionante di questo tipo. Il Paragrafo 3.4 del Capitolo 3 spiega il concetto di memorizzazione dei programmi.

Nel 1947 Eckert e Mauchly inoltrarono la domanda per ottenere il brevetto del calcolatore elettronico. Il preside della Moore School, chiedendo che il brevetto fosse assegnato all'università, spinse Eckert e Mauchly a prendere la decisione di lasciare la scuola; la loro partenza mise in crisi il progetto EDVAC, il cui completamento fu ritardato fino al 1952.

Goldstine se ne andò per unirsi a von Neumann presso l'Istituto di Studi Avanzati (IAS) di Princeton nel 1946; insieme ad Arthur Burks essi pubblicarono un documento basandosi su quello scritto precedentemente da von Neumann [Burks, Goldstine e von Neumann 1946]. L'articolo era assolutamente straordinario per quel periodo storico; leggendolo oggi nessuno indovinerebbe mai che questa pietra miliare sia stata scritta più di 50 anni fa, dal momento che viene discussa la maggior parte dei concetti fondamentali dell'architettura dei moderni calcolatori. Questo articolo portò alla realizzazione della macchina IAS costruita da Julian Bigelow, che aveva un totale di 1024 parole (word) da 40 bit e che era circa 10 volte più veloce dell'ENIAC. Il gruppo discusse dei possibili usi della macchina, pubblicò risultati ed incoraggiò le visite al laboratorio; i risultati pubblicati e le visite ispirarono lo sviluppo di molti nuovi calcolatori.

Recentemente ci sono state alcune controversie sul lavoro di John Atanasoff, che costruì un prototipo di calcolatore elettronico nei primi anni Quaranta. La sua macchina, progettata alla Iowa State University, era un calcolatore ad uso speciale (*special-purpose*) che non funzionò mai completamente; Mauchly fece una breve visita ad Atanasoff prima di costruire l'ENIAC. L'esistenza della macchina di Atanasoff, insieme al ritardo nel registrare il brevetto dell'ENIAC (il lavoro era segreto, ed il brevetto non fu assegnato fino al termine della guerra) ed alla distribuzione dell'articolo di von Neumann sull'EDVAC, furono usati per interrompere il brevetto di Eckert e Mauchly. Benché alcune controversie ancora riguardino il ruolo di Atanasoff, il merito di aver costruito il primo calcolatore elettronico general-purpose funzionante è generalmente attribuito a Eckert e Mauchly [Stern 1980].

Figura 1.22 L'EDSAC nel 1949 fu il primo calcolatore con una memoria per dati e istruzioni. Wilkes è l'uomo con gli occhiali, inginocchiato in primo piano. Fotografia gentilmente concessa dal Computer Museum di Boston.

Un'altra macchina storica che merita di essere menzionata è un calcolatore special-purpose costruito da Konrad Zuse in Germania tra la fine degli anni Trenta e gli inizi degli anni Quaranta; benché Zuse avesse pronto il progetto di un calcolatore programmabile, il governo tedesco decise di non finanziare le ricerche scientifiche con durata maggiore di due anni perché i burocrati prevedevano che la guerra sarebbe stata vinta entro quel termine.

Dall'altra parte del Canale della Manica, durante la Seconda Guerra Mondiale furono costruiti calcolatori special-purpose per decodificare i messaggi intercettati provenienti dalla Germania; nel 1943 a Bletchley Park un gruppo di persone, che includeva Alan Turing, costruì il Colossus. Le macchine costruite vennero tenute segrete fino al 1970; dopo la guerra il gruppo ebbe poca influenza sui calcolatori commerciali inglesi.

Mentre il lavoro sull'ENIAC procedeva, Howard Aiken costruiva ad Harvard un calcolatore elettromeccanico chiamato Mark-I (nome che più tardi la Manchester University adottò per la sua macchina); al Mark-I fece seguito una macchina a relè, il Mark-II, ed una coppia di calcolatori a valvole, il Mark-III e il Mark-IV. A diffe-

renza di macchine precedenti come l'EDSAC, che contenevano una sola memoria per istruzioni e dati, il Mark-III e il Mark-IV avevano due memorie separate, una per le istruzioni ed una per i dati. Queste macchine vennero considerate anacronistiche da coloro che sostenevano la tesi di von Neumann; il termine *architettura Harvard* fu coniato per descrivere i calcolatori con due memorie separate. Con il dovuto rispetto per la storia, oggi questo termine viene usato per definire le macchine con una singola memoria principale ma con cache separate per dati e istruzioni.

Nel 1947 presso il MIT ebbe inizio il progetto Whirlwind, orientato allo sviluppo di applicazioni per l'elaborazione in tempo reale di segnali radar. Benché abbia condotto a numerose invenzioni, l'innovazione più importante fu quella delle memorie a nucleo magnetico: il Whirlwind aveva 2048 parole da 16 bit realizzate su nucleo magnetico. I nuclei magnetici rappresentarono la principale tecnologia di memorizzazione per circa 30 anni.

Gli sviluppi commerciali

Nel dicembre del 1947 Eckert e Mauchly fondarono la Eckert-Mauchly Computer Corporation; la loro prima macchina, il BINAC, fu costruita per la Northrop e apparve nell'agosto del 1949. Dopo alcune difficoltà finanziarie, l'azienda fu acquisita dalla Remington-Rand, dove fu costruito l'UNIVAC I (Universal Automatic Computer), progettato per essere venduto come calcolatore general-purpose (figura 1.23). Immesso sul mercato nel giugno 1951 al prezzo di circa 1 milione di dollari, l'UNIVAC I fu il primo calcolatore commerciale di successo: furono costruiti 48 di questi sistemi! Questo storico calcolatore, come molti altri affascinanti reperti della storia dell'informatica, può essere ammirato al Computer Museum di Boston (Massachusetts), ed al Computer History Center a Mountain View (California).

L'IBM si occupava da tempo della produzione di schede perforate e di automazione d'ufficio, ma non iniziò a costruire calcolatori fino al 1950. Il primo calcolatore IBM, l'IBM 701, fu posto sul mercato nel 1952; ne furono vendute in totale 19 unità. Nei primi anni Cinquanta molti erano pessimisti sul futuro dei calcolatori, essendo convinti che il mercato e le opportunità per queste macchine così "altamente specializzate" fossero molto limitati.

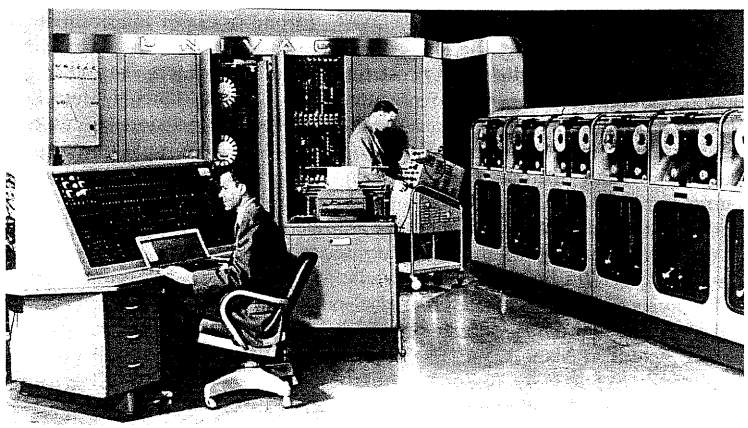


Figura 1.23 L'UNIVAC I, il primo calcolatore commerciale negli Stati Uniti. Predisse in modo corretto il risultato delle elezioni presidenziali del 1952, ma il suo pronostico non fu divulgato perché gli esperti si posero il problema degli usi che si sarebbero fatti di queste informazioni. Fotografia gentilmente concessa dall'Istituto Charles Babbage, University of Minnesota.

Nel 1964, dopo aver investito 5 miliardi di dollari, l'IBM fece una mossa audace annunciando il System/360. Un portavoce dell'IBM disse le seguenti parole:

> *Non saremo assolutamente umili in questo annuncio. Questo è l'annuncio del prodotto più importante che la nostra azienda abbia mai realizzato nella sua storia. Non è un calcolatore nel senso con cui questa parola è stata usata fino ad oggi. Non è un prodotto, ma una linea di prodotti ... le cui prestazioni variano dall'estremo inferiore a quello superiore della gamma dei calcolatori.*

Trasformando l'architettura astratta in realtà commerciale, l'IBM annunciò sei implementazioni dell'architettura del System/360 che variavano per prezzo e prestazioni di un fattore 25. La figura 1.24 mostra quattro di questi modelli. L'IBM puntò sul successo di una *famiglia di calcolatori*, più che sul singolo esemplare, e

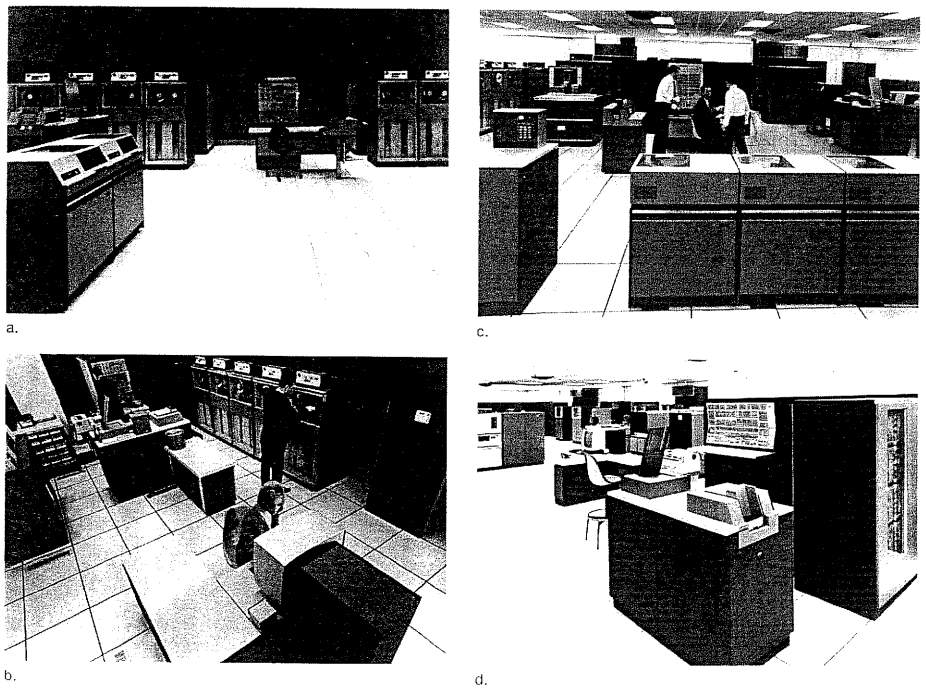


Figura 1.24 IBM System/360: i modelli 40, 50, 65 e 75 furono tutti introdotti nel 1964. Questi quattro modelli variavano per prezzo e prestazioni almeno di un fattore 10; il fattore sale a 25 se si includono i modelli 20 e 30 (non mostrati in figura). Frequenza di clock, dimensione della memoria e prezzo approssimativo per il solo processore con memoria di dimensione media: (a) Modello 40, 1,6 MHz, 32 KB-256 KB, 225000 dollari; (b) Modello 50, 2,0 MHz, 128 KB-256 KB, 550000 dollari; (c) Modello 65, 5,0 MHz, 256 KB-1 MB, 1200000 dollari; (d) Modello 75, 5,1 MHz, 256 KB-1 MB, 1900000 dollari. L'aggiunta dei dispositivi di I/O tipicamente aumentava il prezzo di un fattore variabile da 1,8 a 3,5, con i fattori più elevati per i modelli meno costosi. Fotografie gentilmente concesse da IBM.

vinse la scommessa. Il System/360 ed i suoi successori dominarono il mercato dei grandi calcolatori.

Circa un anno più tardi la Digital Equipment Corporation (DEC) introdusse il PDP-8, il primo *minicomputer* commerciale, mostrato in figura 1.25, che rappresentò una svolta nel progetto di sistemi a basso costo, permettendo alla DEC di offrire una macchina ad un prezzo inferiore ai 20000 dollari. I minicomputer furono i precursori dei microprocessori; l'Intel inventò il primo microprocessore, l'Intel 4004, nel 1971 (figura 1.26).

Nel 1963 fu annunciata la realizzazione del primo *supercomputer*. Tale annuncio non avvenne né da parte di grosse aziende né da parte di centri di ricerca sulle tecnologie avanzate. Seymour Cray fu a capo del progetto del Control Data Corporation CDC 6600 nel Minnesota; questa macchina includeva molte delle idee che si trovano negli ultimi microprocessori. In seguito Cray lasciò la CDC per fondare la Cray Research nel Wisconsin e nel 1976 annunciò il Cray-1 (figura 1.27), che fu contemporaneamente il più veloce al mondo, il più costoso e la macchina con il miglior rapporto prezzo/prestazioni per i programmi scientifici. Il 1996 tuttavia vide il passaggio della Cray Research nelle mani della Silicon Graphics, per cui oggi non esistono più aziende indipendenti di supercomputer.

Mentre Seymour Cray stava creando il calcolatore più costoso del mondo, altri progettisti stavano cercando di utilizzare il microprocessore per costruire un calcolatore così economico da poter entrare in tutte le case. Non esiste un'unica storia del *personal computer*, ma nel 1977 l'Apple II (figura 1.28) di Steve Jobs e Steve Wozniak definì gli standard di basso costo, elevato volume di produzione ed alta

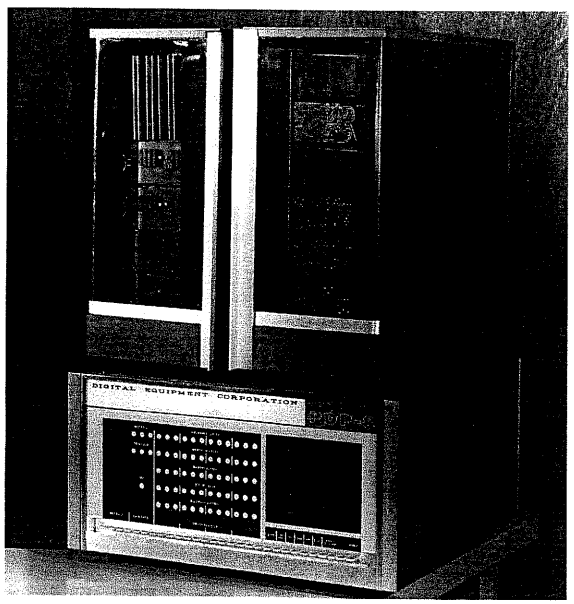


Figura 1.25 Il DEC PDP-8, il primo minicomputer commerciale, annunciato nel 1965. Fra gli altri usi, il PDP-8 fu adoperato durante la messa in scena del musical *A Chorus Line*. Fotografia gentilmente concessa dalla Digital Equipment Corporation, Corporate Photo Library.

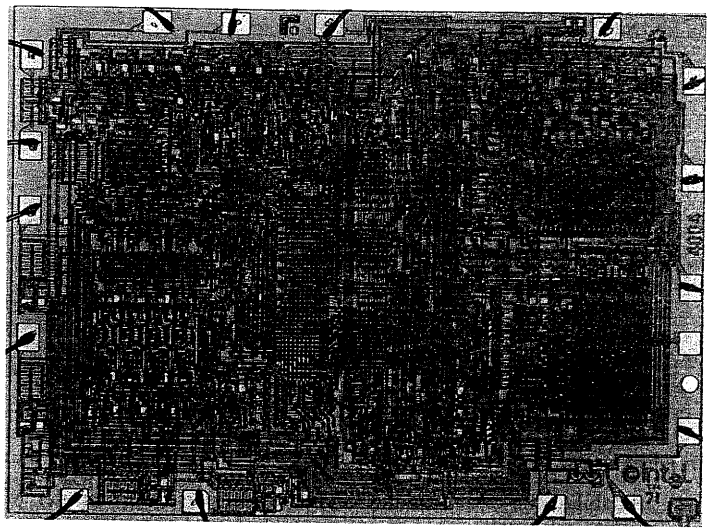


Figura 1.26 Microfotografia dell'Intel 4004, il primo microprocessore (1971). Si confronti questo microprocessore, che ha appena 2300 transistor e misura 0,3 per 0,4 cm, con il microprocessore di figura 1.18. Fotografia gentilmente concessa da Intel.

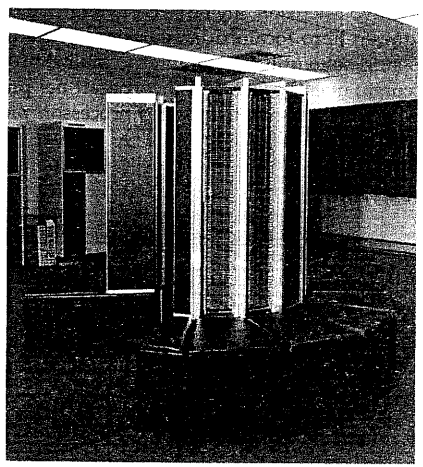


Figura 1.27 Il Cray-1, il primo calcolatore vettoriale commerciale, annunciato nel 1976. Questa macchina ebbe il merito non usuale di essere sia il calcolatore più veloce per le applicazioni scientifiche che quello con il miglior rapporto prezzo/prestazioni per tali applicazioni. Visto dall'alto, il calcolatore ricorda la lettera C. Seymour Cray scomparve nel 1996 a causa delle ferite riportate in un incidente stradale. Al tempo della sua morte questo settantenne pioniere del calcolatore stava lavorando alla sua visione del futuro dei supercomputer. (Vedere il link Cray all'indirizzo www.mpk.com/books_catalog/cod/links.html per maggiori dettagli). Fotografia gentilmente concessa dalla Cray Research, Inc.

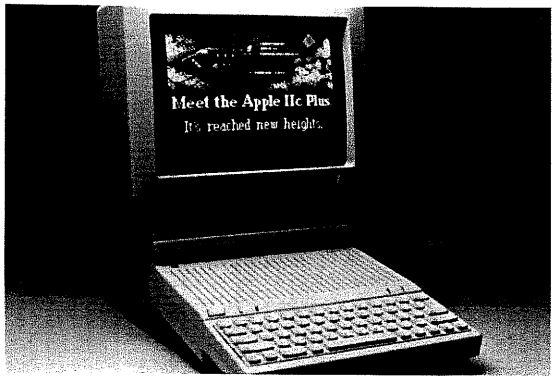


Figura 1.28 L'Apple IIC. Progettato da Steve Wozniak, il suo successo caratterizzò l'industria del computer nel 1977 e pose gli standard di costo ed affidabilità di tale industria. Fotografia gentilmente concessa da Apple Computer, Inc.

affidabilità che caratterizzano l'industria del personal computer. Ma pur avendo iniziato con quattro anni di anticipo, il personal computer Apple finì per essere secondo in quanto a popolarità: il personal computer IBM, annunciato nel 1981, divenne il calcolatore più venduto di tutti i tempi e il suo successo fece sì che il microprocessore più popolare fosse quello della Intel e che il sistema operativo più diffuso fosse quello della Microsoft. Oggi il CD più diffuso è quello contenente il sistema operativo Microsoft, anche se costa molto di più di un CD musicale!

Benché questo libro focalizzi la sua attenzione sui calcolatori general-purpose, oggi i calcolatori sono utilizzati all'interno di altri prodotti senza che spesso l'utente se ne accorga; questi processori incorporati nei prodotti diventano ogni giorno più diffusi. La tabella seguente mostra i processori più diffusi nel 1995; per quanto riguarda il MIPS solo 300000 dei 5500000 processori prodotti sono stati usati nei calcolatori, il resto è incorporato in videogiochi, stampanti laser e così via.

Insieme di istruzioni	Numero
80x86	50000000
MIPS	5500000
PowerPC	3300000
SPARC	700000
HP PA-RISC	300000
DEC Alpha	200000

Le generazioni di calcolatori

A partire dal 1952 sono stati costruiti migliaia di nuovi calcolatori utilizzando una grande varietà di tecnologie e con prestazioni ampiamente diverse. Per porre tali sviluppi nella giusta prospettiva, l'industria ha cercato di classificare i calcolatori in generazioni. Questa classificazione è spesso basata sulla tecnologia utilizzata in ciascuna generazione per l'implementazione, come mostrato in figura 1.29. Secondo la tradizione ciascuna *generazione di calcolatori* dura da 8 a 10 anni, benché molto si discuta sull'anno di inizio e sulla durata precisa, specie per le generazioni

Generazione	Periodo	Tecnologia	Prodotto innovativo
1	1950-1959	Valvole	Calcolatori elettronici commerciali
2	1960-1968	Transistor	Calcolatori più economici
3	1969-1977	Circuiti integrati	Minicomputer
4	1978-?	LSI e VLSI	Personal computer e workstation

Figura 1.29 Le generazioni dei calcolatori sono in generale determinate dal cambiamento della tecnologia di implementazione dominante. Nel caso tipico ciascuna generazione offre l'opportunità di creare una nuova classe di calcolatori e nuove aziende informatiche.

recenti. Per convenzione la prima generazione è quella dei calcolatori elettronici commerciali, e non quella dei calcolatori meccanici o elettromeccanici che li hanno preceduti.

Il successo del microprocessore ha esteso considerevolmente la durata della quarta generazione, che copre un periodo quasi pari alle tre precedenti. Tale successo è dovuto a miglioramenti rapidi ma sempre di tipo evolutivo; la durata della generazione attuale è il risultato della lunga dominazione dei microprocessori VLSI nell'industria, anche se questo fatto non viene spesso riconosciuto. Non essendoci nessuna nuova tecnologia all'orizzonte, non è chiaro se mai nascerà una nuova generazione di calcolatori.

La figura 1.30 riassume le caratteristiche principali di alcune delle macchine citate in questo paragrafo. Tenendo conto dell'inflazione, il rapporto prezzo/prestazione è cresciuto di circa 240 milioni di volte in 45 anni, o in altre parole del 54% circa ogni anno.

I lettori interessati alla storia del calcolatore sono invitati a consultare la rivista specializzata *Annals of the History of Computers*. Sono stati inoltre pubblicati diversi libri che descrivono i primi anni dell'informatica, molti dei quali scritti dagli stessi pionieri.

Anno	Nome	Dimensioni (metri cubi)	Potenza (watt)	Prestazioni (sommo/sec)	Memoria (KB)	Prezzo originario	Prezzo originario / prestazioni confrontate all'UNIVAC	Prezzo rivalutato (dollari del 1996)	Prezzo rivalutato / prestazioni confrontate all'UNIVAC
1951	UNIVAC I	28	124500	1900	48	$1000000	1	$4996749	1
1964	IBM S/360 modello 50	1,68	10000	500000	64	$1000000	263	$4140257	318
1965	PDP-8	0,23	500	330000	4	$16000	10855	$66071	13135
1976	Cray-1	1,62	60000	166000000	32768	$4000000	21842	$8459712	51604
1981	IBM PC	0,03	150	240000	256	$3000	42105	$4081	154673
1991	HP 9000/ modello 750	0,06	500	50000000	16384	$7400	3556188	$8156	16122356
1996	Intel PPro PC (200 MHz)	0,06	500	400000000	16384	$4400	47846890	$4400	239078908

Figura 1.30 Le caratteristiche dei principali calcolatori commerciali a partire dal 1950, in dollari del periodo e in dollari del 1996, tenendo conto dell'inflazione. A differenza della figura 1.24, qui il prezzo del System/360 modello 50 include i dispositivi di I/O. Fonti: Computer Museum, Boston, e Producer Price Index for Industrial Commodities.

Letture consigliate

Bell, C. G. [1984]. "The mini and micro industries," *IEEE Computer* 17:10 (Ottobre) 14-30.
Una visione personale dell'industria informatica, che comprende una descrizione delle generazioni dei calcolatori, fatta da chi la conosce a fondo.

Bell, C. G. [1996]. *Computer Pioneers and Pioneer Computers*, ACM and the Computer Museum, videocassette.
Due videocassette sulla storia del calcolatore, prodotte da Gordon e Gwen Bell, che includono le seguenti macchine e i loro inventori: l'Hardvard Mark-I, l'ENIAC, l'EDSAC, la macchina IAS e molte altre.

Burks, A. W., H. H. Goldstine e J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Rapporto allo U.S. Army Ordnance Department, p.1, anche apparso in *Papers of John von Neumann*, W. Aspray e A. Burks (a cura di), MIT Press, Cambridge, MA, e Thomas Publishers, Los Angeles, 1987, 97-146.
Un articolo classico, che spiega l'hardware e il software del calcolatore prima che il primo calcolatore con una memoria per dati e istruzioni fosse costruito. Questo testo è citato ampiamente nel Capitolo 3. Fu allo stesso tempo il documento che spiegò i calcolatori al mondo intero e l'origine di una disputa, dal momento che nella sua prima versione non comparvero i nomi di Eckert e Mauchly.

Campbell-Kelly, M. e W. Aspray [1996]. *Computer: A History of the Information Machine*, Basic Books, New York.
Due storici mettono in scena la storia del calcolatore. Il New York Times lo definisce autorevole e ben scritto.

Goldstine, H. H. [1972]. *The Computer: from Pascal to von Neumann*, Princeton University Press, Princeton, NJ.
Una visione personale dell'informatica da parte di uno dei pionieri che lavorò con von Neumann.

Hennessy, J. L., e D. A. Patterson [1996]. Paragrafi 1.4 e 1.5 di *Computer Architecture: A Quantitative Approach*, seconda edizione, Morgan Kaufmann Publishers, San Francisco.
Questi paragrafi contengono maggiori dettagli sul costo dei circuiti integrati, e spiegano le ragioni della differenza fra prezzo e costo.

Public Broadcasting System [1992]. *The Machine that Changed the World*, videocassette.
Questi cinque programmi da un'ora ciascuno includono rare riprese ed interviste con i pionieri dell'industria informatica.

Slater, R. [1987]. *Portraits in Silicon*, MIT Press, Cambridge, MA.
Brevi biografie di 31 pionieri del calcolatore.

Stern, N. [1980]. "Who invented the first electronic digital computer?" *Annals of the History of Computing* 2:4 (ottobre) 375-76.
Una inquadramento storico della controversia fra Atanasoff, e Eckert e Mauchly.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.
Una visione personale dell'informatica da parte di uno dei pionieri.

1.9 Parole chiave

Alla fine di ogni capitolo e in appendice compare una lista di parole chiave, che richiamano le idee fondamentali presentate. Coloro che non sono sicuri del significato dei termini elencati nel seguito sono invitati a fare riferimento al Glossario che si trova al fondo del libro, dove compare la definizione completa di ciascun termine.

architettura dell'insieme di istruzioni o architettura del set di istruzioni
area del die
assemblatore
astrazione
bit o cifra binaria
calcolatore elettronico general-purpose
chip
circuito integrato
circuito integrato VLSI
compilatore
die
difetto
disco magnetico
dispositivo di ingresso
dispositivo di uscita
floppy disk
generazione di calcolatori
gigabyte
hard disk
implementazione
kilobyte
libreria di subroutine
lingotto di silicio cristallino
linguaggio assembler
linguaggio di programmazione ad alto livello
megabyte
memoria
memoria ad accesso sequenziale
memoria cache
memoria dinamica ad accesso casuale (DRAM)
memoria non volatile
memoria primaria o principale
memoria secondaria
memoria volatile
modulo di memoria single in-line (SIMM)
personal computer
pipeline
pixel
resa
rete geografica (WAN)
scheda madre
semiconduttore
silicio
sistema operativo
software di sistema
supercomputer
supercomputer vettoriale
terabyte
transistor
unità centrale di elaborazione (CPU)
unità di controllo
unità di elaborazione
valvola
video a scansione basato su tubo a raggi catodici (CRT)
wafer

1.10 Esercizi

Dopo il numero di ciascun esercizio è riportata tra parentesi quadre una stima della complessità dell'esercizio in termini di tempo relativo. In media, un esercizio valutato [10] richiederà due volte il tempo richiesto da un esercizio valutato [5]. I paragrafi del testo da leggere prima di affrontare un esercizio saranno indicati tra parentesi angolari; per esempio, <§1.4> significa che dovrebbe essere stato letto il paragrafo 1.4, *I circuiti integrati: alimento per l'innovazione*, che aiuta a risolvere l'esercizio. Se la soluzione di un esercizio dipende da altri esercizi, questi saranno elencati tra parentesi graffe; per esempio, {Es. 1.51} significa che l'esercizio 1.51 dovrebbe essere stato risolto prima di affrontare l'esercizio corrente.

Esercizi da 1.1 a 1.26 Trovare la parola o la frase tra quelle della lista qui sotto che meglio corrisponde alla descrizione riportata nelle domande seguenti. Nelle risposte si usino le lettere a sinistra delle parole. Ciascuna risposta deve essere usata una volta sola.

a architettura dell'insieme di istruzioni
b assemblatore
c astrazione
d bit
e cache
f chip
g circuito integrato
h compilatore
i die
j difetto
k DRAM (memoria dinamica ad accesso casuale)
l famiglia di calcolatori
m implementazione
n istruzione
o memoria
p numero binario
q processore
r resa
s semiconduttore
t sistema operativo
u supercomputer
v transistor
w unità centrale di elaborazione (CPU)
x unità di controllo
y unità di elaborazione
z VLSI (circuito integrato ad altissima scala di integrazione)

1.1 [2] Astrazione fornita dall'hardware al software di basso livello.

1.2 [2] Parte attiva del calcolatore, che segue alla lettera le istruzioni del programma. Somma numeri, esegue test su numeri e così via.

1.3 [2] Nome alternativo per processore.

1.4 [2] Possibile approccio al progetto dell'hardware o del software. Il sistema consiste in una serie gerarchica di livelli, in cui ogni livello nasconde alcuni dettagli al livello superiore.

1.5 [2] Numero in base 2.

1.6 [2] Cifra binaria.

1.7 [2] Collezione di implementazioni della stessa architettura dell'insieme di istruzioni. Queste sono disponibili contemporaneamente e variano in termini di prezzo e prestazioni.

1.8 [2] Componente del processore che esegue operazioni aritmetiche.

1.9 [2] Componente del processore che indica all'unità di elaborazione, alla memoria e ai dispositivi di I/O cosa fare, sulla base delle istruzioni del programma.

1.10 [2] Hardware che obbedisce all'astrazione dell'architettura dell'insieme di istruzioni.

1.11 [2] Macchina ad alte prestazioni, del costo di oltre 1,8 miliardi di lire.

1.12 [2] Singolo comando per un calcolatore.

1.13 [2] Circuito integrato frequentemente utilizzato per costruire la memoria principale.

1.14 [2] Integra da dozzine a centinaia di transistor in un singolo chip.

1.15 [2] Integra da centinaia di migliaia a milioni di transistor in un singolo chip.

1.16 [2] Luogo ove si trovano i programmi mentre sono eseguiti, nonchè i dati da essi richiesti.

1.17 [2] Falla microscopica in un wafer.

1.18 [2] Soprannome per un die o un circuito integrato.

1.19 [2] Interruttore a due posizioni controllato elettricamente.

1.20 [2] Percentuale di die buoni rispetto al numero totale di die in un wafer.

1.21 [2] Programma che gestisce le risorse di un calcolatore a vantaggio dei programmi che sono eseguiti sulla macchina.

1.22 [2] Programma che traduce un'istruzione dalla forma simbolica a quella binaria.

1.23 [2] Programma che traduce da una notazione di alto livello al linguaggio assembler.

1.24 [2] Componente rettangolare che si ottiene dal taglio di un wafer.

1.25 [2] Memoria piccola e veloce che funge da buffer verso la memoria principale.

1.26 [2] Sostanza che non conduce bene l'elettricità.

Esercizi da 1.27 a 1.44 Usando le categorie presenti nella lista riportata sotto, si classifichino gli esempi seguenti. Nelle risposte si usino le lettere a sinistra delle parole. Diversamente dagli esercizi precedenti, le risposte in questo gruppo possono essere usate più di una volta.

a circuito integrato
b dispositivo di ingresso
c dispositivo di uscita
d linguaggio di programmazione di alto livello
e minicomputer
f personal computer
g semiconduttore
h software applicativo
i software di sistema
j supercomputer

1.27 [1] Assemblatore

1.28 [1] C++

1.29 [1] Video basato su tubo a raggi catodici

1.30 [1] Compilatore

1.31 [1] Cray-1

1.32 [1] DEC Alpha

1.33 [1] DRAM

1.34 [1] IBM PC

1.35 [1] Tastiera

1.36 [1] Macintosh

1.37 [1] Microprocessore

1.38 [1] Mouse

1.39 [1] Sistema Operativo

1.40 [1] Pascal

1.41 [1] Stampante

1.42 [1] Silicio

1.43 [1] Foglio elettronico

1.44 [1] Editor di testo

1.45 [10] <§1.3> In un disco magnetico i piatti che contengono i dati sono costantemente in rotazione e, in media, ci vuole mezzo giro prima che il dato richiesto esistente sul disco venga a trovarsi sotto la testina di lettura/scrittura. Assumendo che il disco stia ruotando a 5400 giri al minuto, qual è il tempo medio perché il dato venga a trovarsi sotto la testina del disco? Qual è il tempo medio se il disco gira a 7200 giri al minuto?

1.46 [5] <§§1.4, 1.5> Si assuma che il wafer A contenga un numero doppio di die del wafer B e che si usi lo stesso processo di fabbricazione per entrambi i wafer (in tal modo, il numero di difetti per unità di area è una costante). Uno dei die costerà probabilmente il doppio dell'altro, oppure la differenza sarà maggiore? Fornire una spiegazione intuitiva della risposta (non fare uso delle formule usate per gli esercizi da 1.48 a 1.53).

1.47 [5] <§§1.4, 1.5> Si stimi il numero di transistor che potrebbe essere piazzato nel punto alla fine di questa frase usando i processi di fabbricazione che hanno prodotto i die nelle figure 1.9 e 1.18.

In maggior dettaglio

Il costo dei circuiti integrati

L'approccio seguito in questo libro prevede l'inclusione negli esercizi di parti opzionali, denominate "In maggior dettaglio", lasciando all'insegnante la decisione se trattare il materiale relativo in aula, lasciarlo alla lettura a casa da parte degli studenti, o saltare del tutto tale materiale. La prima delle parti di questo tipo fornisce maggiori informazioni sul costo dei circuiti integrati, ed è utilizzata negli esercizi da 1.48 a 1.53.

Il costo di un circuito integrato può essere espresso attraverso tre semplici equazioni:

$$\text{costo per die} = \frac{\text{costo per wafer}}{\text{die per wafer} \times \text{resa}}$$

$$\text{die per wafer} = \frac{\text{area del wafer}}{\text{area del die}}$$

$$\text{resa} = \frac{1}{1 + \left(\text{difetti per area} \times \frac{\text{area del die}}{2}\right)^2}$$

La prima equazione è di immediata derivazione. La seconda è un'approssimazione, poiché non sottrae l'area vicino al bordo arrotondato del wafer, la quale non può contenere i die rettangolari. L'ultima equazione è basata su anni di osservazioni empiriche sulle rese delle fabbriche di circuiti integrati; l'esponente si riferisce al numero di passi critici di processamento all'interno del processo produttivo.

1.48 [5] <§§1.4, 1.5> Qual è la relazione approssimata tra il costo e l'area del die? La relazione approssimata può essere descritta come

$$\text{costo} = f((\text{area del die})^x)$$

per un certo x. Non si richiede di determinare f, ma è possibile determinare x attraverso i seguenti passaggi

$$\text{die per wafer} = f((\text{area del die})^y)$$

$$\text{resa} = f((\text{area del die})^z)$$

ed analizzando poi l'impatto di queste due equazioni sulla prima (si immagini ciò che y e z possono essere). Che implicazioni ha questo per i progettisti?

1.49 [15] <§§1.4, 1.5> Si confrontino le stime del numero di die per wafer calcolate con la formula precedente con i numeri reali forniti nella didascalia della figura 1.16. Si proponga una formula in grado di fornire una stima più accurata del numero di die per wafer e si fornisca una spiegazione della formula.

1.50 [10] <§§1.4, 1.5> Qual è il costo approssimato di un die nel wafer mostrato nella figura 1.16? Si assuma che un wafer di 8 pollici costi 1000 dollari e che la densità di difetti sia pari a 1 per centimetro quadro. Si usi il numero di die per wafer fornito nella didascalia della figura.

Esercizi da 1.51 a 1.53 I chip di DRAM sono significativamente cresciuti ad ogni generazione in termini di dimensione del die, ma le rese sono rimaste pressoché le stesse (dal 43% al 48%). La figura 1.31 mostra le statistiche chiave per la produzione di DRAM nel corso degli anni.

Anno	Capacità	Area del die (cm^2)	Diametro del wafer (pollici)	Resa
1980	64	0,16	5	48%
1983	256	0,24	5	46%
1985	1024	0,42	6	45%
1989	4096	0,65	6	43%
1992	16384	0,97	8	48%

Figura 1.31 Evoluzione storica della capacità della DRAM, della dimensione del die, della dimensione del wafer, e della resa. Fonte: Howard Dicken della DM Data Inc., Scottsdale, Arizona.

1.51 [5] <§§1.4, 1.5> Data la crescita nell'area del die per le DRAM, quale parametro deve migliorare per mantenere la resa inalterata (si vedano le equazioni)?

1.52 [10] <§§1.4, 1.5> {Es. 1.51} Si derivi una formula per il parametro che deve essere migliorato, identificato nell'esercizio 1.51, a partire dagli altri parametri.

1.53 [10] <§§1.4, 1.5> {Es. 1.51, 1.52} Usando la formula della risposta all'esercizio 1.52 qual è il miglioramento calcolabile per tale parametro tra il 1980 e il 1992?

1.54 [8] <§§1.1-1.5> Questo libro copre le astrazioni per i sistemi di calcolo a vari livelli di dettaglio. Si scelga un altro sistema con cui si ha confidenza e scrivano uno o due paragrafi descrivendo qualcuno dei molti diversi livelli di astrazione inerenti a tale sistema. Alcune possibilità sono le automobili, le case, gli aeroplani, la geometria, l'economia e il governo. Ci si accerti di identificare sia le astrazioni di alto livello che quelle di basso livello.

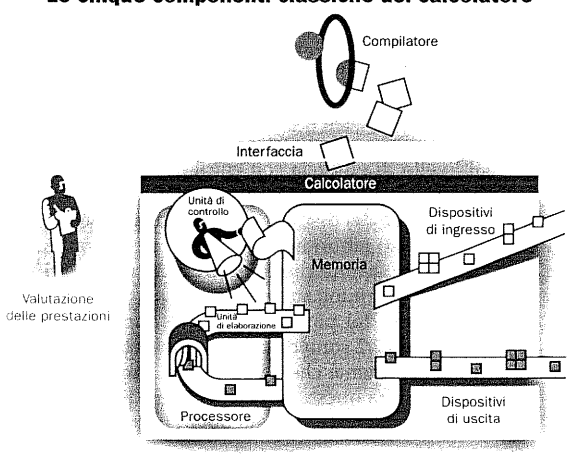
1.55 [15] <§§1.1-1.5> Un amico poco portato per i problemi tecnici vi ha chiesto di spiegare come funzionano i calcolatori. Si scriva una descrizione dettagliata di una pagina per l'amico.

1.56 [10] <§§1.1-1.5> Sotto quali aspetti riscontrate una carenza di comprensione nel funzionamento dei calcolatori? Vi sono dei livelli di astrazione che vi sono particolarmente estranei? Vi sono dei livelli di astrazione che vi sono particolarmente chiari ma relativamente ai quali avete delle domande specifiche? Si scriva almeno un paragrafo per ciascuna di queste domande.

2

Il ruolo delle prestazioni

Le cinque componenti classiche del calcolatore



Il tempo svela la verità.

Seneca
Saggi Morali, 22 d.C.

2.1 Introduzione

Questo capitolo discute le modalità con cui si misurano, si descrivono e si sintetizzano le prestazioni di un calcolatore e descrive i principali fattori da cui queste dipendono. Una ragione fondamentale per esaminare le prestazioni è il fatto che buone prestazioni dell'hardware sono spesso la chiave dell'efficacia dell'intero sistema (hardware più software).

Valutare le prestazioni di un sistema di elaborazione è un compito arduo: le dimensioni e la complessità dei moderni sistemi software, unite all'ampia varietà delle tecniche per il miglioramento delle prestazioni utilizzate dai progettisti di hardware, hanno reso tale valutazione ancora più difficile. Non basta munirsi del manuale del set di istruzioni di una macchina e di un insieme significativo di applicazioni software e pensare di poter misurare la velocità con cui vengono eseguite le applicazioni su quella macchina; infatti devono essere utilizzate metriche di misura diverse a seconda del tipo di applicazione e aspetti differenti del sistema di calcolo possono essere di volta in volta quelli più significativi per valutare le prestazioni globali.

Naturalmente le prestazioni sono quasi sempre un parametro importante quando si deve scegliere fra calcolatori diversi. Misurare le prestazioni in modo accurato e confrontare macchine diverse è un'operazione fondamentale per chi acquista, quindi lo deve essere anche per chi progetta: chi vende calcolatori lo sa bene. Spesso i venditori cercano di far apparire le loro macchine nella miglior luce possibile, anche se questa luce non sempre riflette le necessità di chi acquista; in alcuni casi le caratteristiche dei calcolatori che i venditori mettono in risalto non hanno alcuna influenza sulle normali applicazioni. Di conseguenza, quando si sceglie una macchina, è importante capire come si valutano le prestazioni e quali sono i limiti nella loro misura.

Assumendo il calcolatore come una scatola chiusa, l'interesse per le prestazioni va oltre l'obiettivo della loro valutazione. Per capire perché un frammento di software dia certi risultati in termini di prestazioni, perché un determinato insieme di istruzioni possa essere implementato in modo da avere prestazioni migliori di un altro o perché qualche caratteristica hardware influenzi le prestazioni, è necessario capire da cosa dipendono in generale le prestazioni di una macchina. Ad esempio, per migliorare le prestazioni di un sistema software potrebbe essere necessario sapere quali elementi dell'hardware contribuiscano alle prestazioni complessive e l'importanza relativa di questi elementi. Essi, ad esempio, includono il livello di efficacia con cui il programma utilizza le istruzioni della macchina, il modo in cui l'hardware sottostante implementa le istruzioni e la capacità della memoria e dei dispositivi di I/O di assolvere al loro compito. Capire il modo in cui tutti questi fattori influenzano le prestazioni è cruciale per comprendere le motivazioni che stanno dietro al progetto di aspetti particolari delle macchine, come si vedrà nei capitoli successivi.

Il resto del paragrafo descrive modi diversi di valutare le prestazioni. Nel Paragrafo 2.2 sono descritte le metriche per valutare le prestazioni sia dal punto di vista dell'utente che da quello del progettista. Nel Paragrafo 2.3 sono analizzate le relazioni fra queste metriche e viene proposta la classica equazione delle prestazioni di un calcolatore, usata poi in tutto il testo. I Paragrafi 2.4 e 2.5 descrivono come

scegliere i benchmark (programmi campione) per valutare le macchine e come sintetizzare correttamente le prestazioni di un gruppo di programmi. Il Paragrafo 2.6 propone un insieme di benchmark comunemente usati per valutare le CPU ed esamina le misure ricavate analizzando con tali benchmark alcuni processori Intel. Infine il Paragrafo 2.7 esamina alcuni dei molti trabocchetti in cui i programmatori rischiano di cadere e quelli a cui è soggetto chi ha il compito di valutare e riportare le prestazioni.

Definire le prestazioni

Quando si dice che un calcolatore ha prestazioni migliori di un altro, che cosa si intende veramente? Benché questa domanda possa apparire semplice, un'analogia con gli aerei per trasporto passeggeri può far capire come la questione sia sottile. La figura 2.1 mostra alcuni tipici modelli di aerei per trasporto passeggeri, con la loro velocità di crociera, autonomia di volo e numero di posti. Volendo trovare quale di questi aeroplani abbia le prestazioni migliori, occorrerà innanzitutto definire che cosa si intende per prestazioni. Ad esempio, considerando metriche diverse, l'aereo con la velocità di crociera più elevata è il Concorde, quello con maggiore autonomia di volo il DC-8 e quello in grado di trasportare il maggior numero di passeggeri il 747.

Aereo	Numero di posti	Autonomia di volo (km)	Velocità di crociera (km/h)	Portata (passeggeri x km/h)
Boeing 777	375	7450	980	367500
Boeing 747	470	6680	980	460600
BAC / Sud Concorde	132	6440	2170	286440
Douglas DC-8-50	146	14030	875	127750

Figura 2.1 Capacità, autonomia di volo e velocità di alcuni aerei commerciali. L'ultima colonna mostra la portata, cioè il numero di passeggeri moltiplicato per la velocità di crociera (senza tenere conto dell'autonomia e dei tempi di decollo e atterraggio).

Si supponga di definire le prestazioni in termini di velocità; questo approccio lascia comunque spazio a due possibili definizioni: si può definire come aereo più veloce quello che ha velocità di crociera più elevata, portando il singolo passeggero da un luogo ad un altro nel minor tempo. Se però l'obiettivo fosse trasportare 450 passeggeri da un luogo ad un altro, allora chiaramente il 747 sarebbe il più veloce, come mostra l'ultima colonna della figura. Allo stesso modo è possibile definire le prestazioni di un calcolatore in molti modi diversi.

Facendo eseguire un programma su due stazioni di lavoro diverse, si potrebbe dire che la più veloce è quella che finisce il lavoro per prima. Dirigendo un centro di calcolo che ha due grandi calcolatori che eseguono in time sharing lavori sottomessi da molti utenti, si potrebbe invece considerare come più veloce quello che in una giornata porta a termine più lavori. Il singolo utente di un calcolatore è interessato a ridurre il *tempo di risposta*, cioè il tempo fra l'inizio ed il completamento di un lavoro, detto anche *tempo di esecuzione*. I gestori dei centri di calcolo sono invece spesso interessati ad aumentare il *throughput*, cioè la quantità di lavoro eseguita nell'unità di tempo.

Throughput e tempo di risposta

Esempio

Esempi specifici sono usati in tutto il testo allo scopo di illustrare l'applicazione di idee nuove. Dopo aver posto il problema, ne viene data una risposta. Il lettore è invitato a risolverlo in modo autonomo o, se si sente insicuro, a proseguire nella lettura. Gli esempi proposti sono simili ai problemi che compaiono nel paragrafo di esercizi che chiude ciascun capitolo. Ecco il primo esempio:

I seguenti cambiamenti apportati ad un sistema di calcolo aumentano il throughput, diminuiscono il tempo di risposta o entrambe le cose?

1. Sostituire il processore all'interno del calcolatore con una versione più veloce.
2. Aggiungere processori addizionali ad un sistema che usa processori multipli per compiti separati, ad esempio per gestire il sistema di prenotazioni di una linea aerea.

Soluzione

Diminuire il tempo di risposta aumenta quasi sempre il throughput. Quindi nel caso 1 migliorano sia il tempo di risposta che il throughput. Nel caso 2 nessun compito viene eseguito più velocemente e di conseguenza aumenta soltanto il throughput. Se però nel secondo caso il numero di elaborazioni richieste fosse quasi elevato come il throughput, il sistema potrebbe essere costretto a mettere in coda le richieste. In questo caso aumentare il throughput può anche diminuire il tempo di risposta, dal momento che si riduce il tempo di attesa nella coda. Così, in molti sistemi di calcolo reali modificare il tempo di esecuzione influenza il throughput e viceversa.

Discutendo delle prestazioni degli elaboratori, nei primi capitoli l'attenzione sarà concentrata principalmente sul tempo di risposta (nel Capitolo 8, relativo ai dispositivi di ingresso/uscita, saranno invece esaminate alcune misure basate sul throughput). Per massimizzare le prestazioni, l'obiettivo è minimizzare il tempo di risposta o il tempo di esecuzione necessario per svolgere un determinato compito. Per una macchina X, prestazioni e tempo di esecuzione possono essere correlati nel seguente modo:

$$\text{prestazioni}_X = \frac{1}{\text{tempo di esecuzione}_X}$$

Ciò significa che per due macchine X e Y, se le prestazioni di X sono migliori di quelle di Y, allora si ha che

$$\text{prestazioni}_X > \text{prestazioni}_Y$$

$$\frac{1}{\text{tempo di esecuzione}_X} > \frac{1}{\text{tempo di esecuzione}_Y}$$

$$\text{tempo di esecuzione}_Y > \text{tempo di esecuzione}_X$$

In altre parole, il tempo di esecuzione sulla macchina Y è più elevato che sulla macchina X, se X è più veloce di Y.

Discutendo della progettazione di un calcolatore, spesso è utile correlare le prestazioni di due macchine differenti in modo quantitativo. Verrà usata la frase "X è n volte più veloce di Y" per indicare che

$$\frac{\text{prestazioni}_X}{\text{prestazioni}_Y} = n$$

Se X è n volte più veloce di Y, allora il tempo di esecuzione su Y è n volte maggiore di quello su X:

$$\frac{\text{prestazioni}_X}{\text{prestazioni}_Y} = \frac{\text{tempo di esecuzione}_Y}{\text{tempo di esecuzione}_X} = n$$

Prestazioni relative

Esempio

Se la macchina A esegue un programma in 10 secondi e la macchina B esegue lo stesso programma in 15 secondi, di quanto A è più veloce di B?

Soluzione

Sapendo che A è n volte più veloce di B se

$$\frac{\text{prestazioni}_A}{\text{prestazioni}_B} = n$$

o se

$$\frac{\text{tempo di esecuzione}_B}{\text{tempo di esecuzione}_A} = n$$

si ricava che il rapporto fra le prestazioni è

$$\frac{15}{10} = 1,5$$

e che quindi A è 1,5 volte più veloce di B.

Nell'esempio appena riportato si può anche dire che la macchina B è 1,5 volte *più lenta di* A, dal momento che

$$\frac{\text{prestazioni}_A}{\text{prestazioni}_B} = 1,5$$

vuol dire

$$\frac{\text{prestazioni}_A}{1,5} = \text{prestazioni}_B$$

Per semplicità verrà normalmente usata la terminologia *più veloce di* per confrontare le macchine in modo quantitativo. Dal momento che prestazioni e tempo di esecuzione sono uno il reciproco dell'altro, per aumentare le prestazioni occorre dimi-

nuire il tempo di esecuzione. Per evitare la potenziale confusione fra i termini *aumentare* e *diminuire*, verranno usate le frasi "migliorare le prestazioni" o "migliorare il tempo di esecuzione" per intendere rispettivamente "aumentare le prestazioni" e "diminuire il tempo di esecuzione".

2.2 La misura delle prestazioni

La misura delle prestazioni di un calcolatore è il tempo: il calcolatore che esegue un certo lavoro nel tempo minore è il più veloce. Il *tempo di esecuzione* di un programma è misurato in secondi; il tempo tuttavia può essere misurato in modi diversi, a seconda di ciò che interessa: la definizione più immediata è di *tempo assoluto, tempo di risposta* o *tempo trascorso* (*elapsed time*). Questi termini indicano il tempo totale per completare un lavoro, includendo gli accessi ai dischi, gli accessi alla memoria, le attività di ingresso e uscita, il sovraccarico dovuto al sistema operativo, in sostanza il tempo di tutte le attività richieste per completare il lavoro.

Tuttavia i calcolatori lavorano spesso in condivisione di tempo (time-sharing) e può accadere che un processore stia lavorando su più programmi contemporaneamente; in questi casi il sistema può cercare di massimizzare la produttività complessiva anziché cercare di minimizzare l'elapsed time di un singolo programma. Di conseguenza, si distingue spesso tra il tempo trascorso ed il tempo durante il quale il processore ha lavorato con un determinato obiettivo: il *tempo di esecuzione della CPU*, o più semplicemente *tempo di CPU*, tiene conto di questa distinzione ed è il tempo speso dalla CPU nell'elaborazione richiesta per raggiungere un obiettivo, non includendo il tempo speso nell'attesa di compiere operazioni di I/O o per eseguire altri programmi. Si ricordi tuttavia che il tempo di risposta percepito dall'utente sarà il tempo trascorso nell'esecuzione del programma e non il tempo di CPU. Il tempo di CPU può essere ulteriormente suddiviso nel tempo speso dalla CPU all'interno del programma, denominato *tempo di CPU relativo all'utente*, ed il tempo speso dalla CPU all'interno del sistema operativo per eseguire compiti richiesti dal programma, denominato *tempo di CPU relativo al sistema*. A volte può essere difficile distinguere con precisione il tempo di CPU relativo all'utente e quello relativo al sistema, dal momento che è spesso arduo assegnare ad un certo programma utente, piuttosto che ad un altro, la responsabilità di una determinata attività svolta dal sistema operativo.

Le varie componenti relative al tempo necessario all'esecuzione di un dato compito sono ben visibili nel comando Unix `time`, che potrebbe per esempio restituire i seguenti valori:

90.7u 12.9s 2:39 65%

Il tempo di CPU relativo all'utente è 90,7 secondi, il tempo di CPU relativo al sistema è 12,9 secondi, il tempo trascorso è 2 minuti e 39 secondi (pari a 159 secondi) e la percentuale di tempo trascorso corrispondente al tempo di CPU è

$$\frac{90,7 + 12,9}{159} = 0,65$$

ossia il 65%. Più di un terzo del tempo trascorso relativo a quest'esempio è stato speso nell'attesa di operazioni di I/O oppure eseguendo altri programmi oppure facendo l'una e l'altra cosa.

A volte, quando si esamina il tempo di esecuzione della CPU, il tempo di CPU relativo al sistema è ignorato a causa dell'inaccuratezza delle misure eseguite dai sistemi operativi e dello sbilanciamento che si avrebbe includendo il tempo di CPU relativo al sistema quando si confrontano le prestazioni di macchine con sistemi operativi diversi. D'altra parte, il codice considerato di sistema su alcune macchine è classificato come codice utente su altre e nessun programma può essere eseguito senza che vi sia una qualche componente di sistema operativo in esecuzione sull'hardware; è quindi sostenibile l'opportunità di usare la somma del tempo di CPU relativo all'utente e di quello relativo al sistema come misura del tempo di esecuzione di un programma.

Al fine di mantenere la coerenza, si manterrà la distinzione tra le prestazioni basate sul tempo trascorso e quelle basate sul tempo di esecuzione della CPU: si utilizzerà il termine *prestazioni di sistema* per fare riferimento al tempo trascorso su un sistema senza carico computazionale ed il termine *prestazioni della CPU* per riferirsi al tempo di CPU relativo all'utente. In questo capitolo ci si concentrerà sulle prestazioni della CPU, benché le discussioni relative al concetto di prestazioni possano essere applicate a misure eseguite sia rispetto al tempo trascorso, sia rispetto al tempo di CPU.

Benché agli utenti dei calcolatori interessi principalmente il parametro tempo, quando si esamina una macchina nei dettagli è conveniente ragionare di prestazioni facendo riferimento ad altre metriche; in particolare i progettisti di calcolatori utilizzano per le macchine una misura relativa alla velocità con cui l'hardware è in grado di eseguire alcune operazioni elementari. Quasi tutti i calcolatori sono costruiti usando un segnale di clock caratterizzato da una frequenza costante, che sincronizza i vari eventi all'interno dell'hardware; questi intervalli di tempo discreti sono denominati *cicli di clock* (o tick, clock tick, periodi di clock, clock, cicli). I progettisti fanno riferimento alla durata di un *periodo di clock* esprimendo il tempo necessario per completare un ciclo di clock (ad esempio 2 nanosecondi o 2 ns) oppure introducendo la *frequenza di clock* (ad esempio 500 megahertz o 500 MHz), che è l'inverso del periodo di clock. Nel prossimo paragrafo sarà formalizzata in maggior dettaglio la relazione tra i cicli di clock del progettista hardware ed i secondi dell'utente di un calcolatore.

2.3 Relazioni tra le metriche

Utilizzatori finali e progettisti solitamente utilizzano metriche di tipo diverso per valutare le prestazioni. Se si potessero mettere in relazione le varie metriche, sarebbe possibile determinare l'effetto di una variazione del progetto sulle prestazioni percepite dagli utenti. Dato che in questa fase l'interesse è rivolto alle prestazioni della CPU, la misura fondamentale è il tempo di esecuzione della CPU (o tempo di CPU); le metriche più elementari (cicli di clock e tempo di ciclo) sono legate al tempo di CPU tramite una semplice equazione:

tempo di CPU relativo a un programma	=	cicli di clock della CPU relativi al programma	×	periodo di ciclo del clock

Poiché la frequenza di clock è il reciproco del tempo di ciclo, è possibile utilizzare la seguente equazione in alternativa:

$$\text{tempo di CPU relativo al programma} = \frac{\text{cicli di clock della CPU relativi al programma}}{\text{frequenza di clock}}$$

Quest'ultima equazione chiarisce come un progettista hardware possa migliorare le prestazioni riducendo o la durata del ciclo di clock o il numero di cicli di clock necessari ad eseguire un programma. Come vedremo in questo capitolo e nei successivi Capitoli 5, 6 e 7, i progettisti spesso scelgono delle situazioni di compromesso tra il numero di cicli di clock e la durata di ciascuno di essi. Molte tecniche capaci di ridurre il numero di cicli di clock, infatti, tendono ad aumentare il tempo di ciclo.

Esempio

Miglioramento delle prestazioni

Un dato programma viene eseguito in 10 secondi dal calcolatore A, dotato di un clock a 400 MHz. Un progettista deve costruire un calcolatore B in grado di eseguire lo stesso programma in 6 secondi ed ha concluso che è possibile aumentare in modo significativo la frequenza di clock; questa modifica avrà però influenza su tutto il progetto della CPU, facendo sì che il calcolatore B richieda un numero di cicli di clock maggiore di un fattore 1,2 rispetto al calcolatore A. Dovendo dare un consiglio al progettista, quale sarà la frequenza di clock a cui puntare come obiettivo?

Soluzione

Si trovi innanzi tutto il numero di cicli di clock necessari all'esecuzione del programma su A:

$$\text{tempo di CPU}_A = \frac{\text{cicli di clock della CPU}_A}{\text{frequenza di clock}_A}$$

$$10\ \text{s} = \frac{\text{cicli di clock della CPU}_A}{400 \cdot 10^6\ \text{cicli/s}}$$

$$\text{cicli di clock della CPU}_A = 10\ \text{s} \cdot 400 \cdot 10^6\ \text{cicli/s} = 4000 \cdot 10^6\ \text{cicli}$$

Il tempo di CPU per B si può calcolare tramite la seguente equazione:

$$\text{tempo di CPU}_B = \frac{1{,}2 \cdot \text{cicli di clock della CPU}_A}{\text{frequenza di clock}_B}$$

$$6\ \text{s} = \frac{1{,}2 \cdot 4000 \cdot 10^6\ \text{cicli}}{\text{frequenza di clock}_B}$$

$$\text{frequenza di clock}_B = \frac{1{,}2 \cdot 4000 \cdot 10^6\ \text{cicli}}{6\ \text{s}} = \frac{800 \cdot 10^6\ \text{cicli}}{\text{s}} = 800\ \text{MHz}$$

Il calcolatore B dovrà quindi lavorare ad una frequenza di clock doppia rispetto ad A per eseguire il programma in 6 secondi.

Interfaccia Hardware-Software

In questo testo sono presenti delle sezioni intitolate "Interfaccia Hardware-Software", che mirano ad evidenziare le interazioni più importanti tra alcune componenti software (tipicamente programmi, compilatori, sistemi operativi) ed alcune componenti hardware del calcolatore. Oltre ad evidenziare tali interazioni, queste sezioni hanno lo scopo di ricordare che il progetto dell'hardware e quello del software interagiscono strettamente.

Le equazioni degli esempi precedenti non includevano alcun riferimento al numero di istruzioni presenti nel programma. Tuttavia, dato che il compilatore ha evidentemente generato delle istruzioni eseguibili ed il sistema ha dovuto eseguire le istruzioni componenti il programma, il tempo di esecuzione dovrà necessariamente dipendere dal numero di istruzioni che compongono un programma. È possibile esprimere il tempo di esecuzione come prodotto del numero di istruzioni eseguite per il tempo medio di esecuzione di ciascuna istruzione. Di conseguenza, il numero di cicli di clock necessari per l'esecuzione di un programma si può riscrivere come

$$\text{cicli di clock della CPU} = \text{numero di istruzioni nel programma} \times \text{numero medio di cicli di clock per istruzione}$$

Il termine *cicli di clock per istruzione*, calcolato come la media del numero di cicli di clock che le diverse istruzioni richiedono per essere completate, è spesso abbreviato con la sigla CPI (clock per istruzione). Dato che istruzioni diverse possono richiedere un tempo di esecuzione differente in funzione del compito che svolgono, il valore di CPI è la media di tutte le istruzioni eseguite nel programma. Il parametro CPI è un termine utile per confrontare architetture diverse aventi lo stesso set di istruzioni, dal momento che il numero di istruzioni necessario ad un programma è evidentemente costante.

Esempio

Utilizzo dell'equazione delle prestazioni

Siano date due implementazioni diverse dello stesso set di istruzioni. Il calcolatore A ha una durata del ciclo di clock pari ad 1 ns ed un CPI pari a 2,0 per un determinato programma, mentre il calcolatore B ha una durata del ciclo di clock pari a 2 ns ed un CPI pari a 1,2 per lo stesso programma. Quale dei due calcolatori è più veloce, limitatamente a questo programma, e di quanto?

Soluzione

È noto che ciascun calcolatore esegue lo stesso numero di istruzioni durante l'esecuzione del programma e si chiami I tale numero. Per prima cosa occorre calcolare il numero di cicli di clock del processore per ciascun calcolatore:

$$\text{cicli di clock della CPU}_A = I \cdot 2{,}0$$

$$\text{cicli di clock della CPU}_B = I \cdot 1{,}2$$

Si può ora calcolare il tempo di CPU per ciascun calcolatore:

$$\text{tempo di CPU}_A = \text{cicli di clock della CPU}_A \times \text{durata del ciclo di clock}_A = \\ = I \cdot 2{,}0 \cdot 1 \text{ ns} = 2 \cdot I \text{ ns}$$

Analogamente per B si ha

$$\text{tempo di CPU}_B = I \cdot 1{,}2 \cdot 2 \text{ ns} = 2{,}4 \cdot I \text{ ns}$$

Evidentemente il calcolatore A è più veloce di un fattore dato dal rapporto tra i tempi di esecuzione:

$$\frac{\text{prestazioni della CPU}_A}{\text{prestazioni della CPU}_B} = \frac{\text{tempo di esecuzione}_A}{\text{tempo di esecuzione}_B} = \frac{2{,}4 \cdot I \text{ ns}}{2 \cdot I \text{ ns}} = 1{,}2$$

Si può dunque concludere che il sistema A è 1,2 volte più veloce del sistema B nell'esecuzione di questo programma.

Si può quindi scrivere una semplice equazione per il calcolo delle prestazioni in termini di numero di istruzioni eseguite dal programma, CPI e durata del ciclo di clock:

$$\text{tempo di CPU} = \text{numero di istruzioni} \times \text{CPI} \times \text{durata del ciclo di clock}$$

o alternativamente

$$\text{tempo di CPU} = \frac{\text{numero di istruzioni} \times \text{CPI}}{\text{frequenza di clock}}$$

Le formule precedenti si rivelano particolarmente utili in quanto evidenziano i tre fattori principali che influiscono sulle prestazioni; si potranno utilizzare per confrontare due diverse implementazioni o per valutare un'alternativa progettuale, conoscendone l'impatto sui tre parametri.

Quadro d'insieme

La figura 2.2 riporta le metriche fondamentali a diversi livelli di astrazione ed indica che cosa viene misurato in ciascun caso; i diversi fattori si combinano nella seguente formula, che dà il tempo di esecuzione (misurato in secondi):

$$\text{tempo} = \frac{\text{istruzioni}}{\text{programma}} \times \frac{\text{cicli di clock}}{\text{istruzione}} \times \frac{\text{secondi}}{\text{ciclo di clock}}$$

Si ricordi sempre che l'unica misura completa ed affidabile delle prestazioni di un calcolatore è il tempo di esecuzione. Per esempio, ristrutturare il set di istruzioni in modo da diminuire il numero di istruzioni eseguite potrebbe richiedere un'organizzazione tale da imporre un ciclo di clock più lento, vanificando il miglioramento ottenuto sul fronte del numero di istruzioni. Analogamente, dal momento che il parametro CPI dipende dalla combinazione delle diverse istruzioni, un programma che esegue un numero minore di istruzioni non è necessariamente più veloce.

Componenti delle prestazioni	Unità di misura
tempo di CPU per l'esecuzione di un programma	secondi
numero di istruzioni	istruzioni eseguite
cicli di clock per istruzione (CPI)	numero medio di cicli di clock per istruzione
durata del ciclo di clock	secondi per ciclo di clock

Figura 2.2 Le componenti principali delle prestazioni e le relative unità di misura

Come si possono determinare i valori dei fattori che compaiono nell'equazione per il calcolo delle prestazioni? Il tempo di esecuzione della CPU può essere misurato mandando il programma in esecuzione, mentre la durata del ciclo di clock viene normalmente riportata come parte della documentazione che accompagna la macchina; il numero di istruzioni ed il CPI sono invece più difficili da ottenere. Se la frequenza di clock ed il tempo di CPU fossero noti, il numero di istruzioni ed il CPI potrebbero essere ricavati l'uno dall'altro.

Il numero di istruzioni può essere misurato attraverso opportuni strumenti software che osservano l'esecuzione di un programma oppure attraverso un simulatore dell'architettura. Alternativamente si potrebbero utilizzare dei contatori hardware, di cui alcuni processori dispongono, per tenere traccia di diverse misure, compreso il numero di istruzioni eseguite. Essendo il numero di istruzioni dipendente dall'architettura, ma indipendente dalla particolare implementazione, esso può essere misurato anche in mancanza di informazioni dettagliate. Il CPI, viceversa, dipende da un ampio spettro di dettagli progettuali relativi al calcolatore, tra cui la modalità di realizzazione del sottosistema di memoria e la struttura del processore (come meglio emergerà nei Capitoli 5, 6 e 7), oltre che dalla combinazione dei tipi di istruzioni eseguiti dall'applicazione specifica. Il CPI dunque varia da applicazione ad applicazione, come pure tra diverse implementazioni dello stesso set di istruzioni.

I progettisti solitamente determinano il CPI attraverso una simulazione approfondita dell'implementazione o mediante la combinazione di contatori hardware con i risultati della simulazione. Talvolta è possibile calcolare il numero di cicli di clock della CPU semplicemente analizzando i diversi tipi di istruzioni ed utilizzando per ciascuno il relativo numero di cicli di clock; in questi casi si applica la seguente equazione:

$$\text{cicli di clock della CPU} = \sum_{i=1}^{n} (\text{CPI}_i \text{C}_i)$$

dove C_i è il numero di istruzioni della classe i che sono state eseguite e CPI_i è il numero medio di cicli per le istruzioni della stessa classe. Il numero di tali classi è indicato da n. Si ricordi che il valore complessivo di CPI per un programma è dipendente sia dal numero di cicli per tipo di istruzione, sia dalla frequenza dei diversi tipi di istruzione durante l'esecuzione del programma.

Confronto tra frammenti di codice

Esempio

Un progettista di compilatori sta cercando di scegliere tra due sequenze di codici per un determinato calcolatore. I progettisti hardware hanno fornito le seguenti informazioni:

Classe di istruzioni	CPI per questa classe di istruzioni
A	1
B	2
C	3

Una particolare istruzione di un linguaggio ad alto livello può essere compilata ricorrendo a due diverse sequenze di codici, le quali richiedono le seguenti combinazioni di istruzioni:

Sequenza di codici	Numero di istruzioni per ciascuna classe		
	A	B	C
1	2	1	2
2	4	1	1

Quale sequenza di codici causa l'esecuzione del maggior numero di istruzioni? Quale viene eseguita più velocemente? Qual è il CPI per ciascuna delle sequenze?

Soluzione

La sequenza 1 causa l'esecuzione di 2 + 1 + 2 = 5 istruzioni, mentre la sequenza 2 ne causa 4 + 1 + 1 = 6. Pertanto il numero minore di istruzioni si otterrebbe con la sequenza di codici 1.

Per calcolare il numero di cicli di clock della CPU in ciascuna sequenza, a partire dal numero di istruzioni e dal CPI, si può utilizzare la relazione

$$\text{cicli di clock della CPU} = \sum_{i=1}^{n} (\text{CPI}_i \text{C}_i)$$

In questo caso si ottiene

$$\text{cicli di clock della CPU}_1 = (2 \cdot 1) + (1 \cdot 2) + (2 \cdot 3) = 2 + 2 + 6 = 10 \text{ cicli}$$

$$\text{cicli di clock della CPU}_2 = (4 \cdot 1) + (1 \cdot 2) + (1 \cdot 3) = 4 + 2 + 3 = 9 \text{ cicli}$$

La sequenza di codici 2 è pertanto più veloce, sebbene porti all'esecuzione di un'istruzione in più. Il CPI della sequenza 2 sarà inoltre inferiore, in quanto impiega meno cicli di clock per eseguire più istruzioni. I valori dei parametri CPI si possono calcolare come segue:

$$\text{CPI} = \frac{\text{cicli di clock della CPU}}{\text{numero di istruzioni}}$$

$$\text{CPI}_1 = \frac{\text{cicli di clock della CPU}_1}{\text{numero di istruzioni}_1} = \frac{10}{5} = 2$$

$$\text{CPI}_2 = \frac{\text{cicli di clock della CPU}_2}{\text{numero di istruzioni}_2} = \frac{9}{6} = 1,5$$

L'esempio precedente evidenzia il pericolo che si corre nell'utilizzare uno solo dei fattori (ad esempio il numero di istruzioni) nel quantificare le prestazioni; nel confrontare due sistemi, è opportuno tener conto delle tre componenti, che solamente se combinate forniscono il tempo di esecuzione. Se alcuni dei fattori sono identici, come lo era la frequenza di clock nell'esempio, le prestazioni si possono determinare dal raffronto dei soli altri fattori: essendo il valore di CPI dipendente dalla combinazione delle istruzioni, occorre considerare sia il numero di istruzioni che il CPI, anche a parità di frequenza di clock. Gli esercizi da 2.18 a 2.24 permettono di meglio indagare questi aspetti, valutando una serie di miglioramenti a calcolatori e compilatori che hanno effetti sulla frequenza di clock, sul parametro CPI e sul numero di istruzioni. Nel paragrafo seguente, invece, verrà esaminata una misura delle prestazioni che pur essendo molto diffusa non incorpora tutti i fattori e può quindi risultare fuorviante.

2.4 Scegliere i programmi per valutare le prestazioni

Un utente che ogni giorno esegue gli stessi programmi è il candidato ideale per valutare un nuovo calcolatore. L'insieme dei programmi eseguiti forma il *carico di lavoro* e per valutare due sistemi di calcolo l'utente dovrebbe semplicemente confrontare il tempo di esecuzione del carico di lavoro sulle due macchine. La maggior parte degli utenti però non è in questa situazione e deve affidarsi ad altri modelli per la misura delle prestazioni di una macchina, sperando che i modelli rispecchino in futuro le effettive prestazioni sul proprio carico di lavoro. Solitamente questa soluzione alternativa viene realizzata valutando la macchina con un insieme di *benchmark*, ossia programmi campione appositamente scelti per misurare le prestazioni: i benchmark costituiscono un carico di lavoro che gli utenti sperano possa prevedere le prestazioni relative ai casi reali.

Oggi è opinione pienamente condivisa che il miglior tipo di programma da usare come benchmark sia un'applicazione reale: possono essere applicazioni che gli utenti usano regolarmente o semplicemente applicazioni tipiche. Ad esempio, in un ambiente in cui gli utenti sono soprattutto ingegneri, i benchmark utilizzati contengono diverse applicazioni tipiche del campo ingegneristico o scientifico; se la comunità di utenti è principalmente costituita da sviluppatori di software, i migliori benchmark comprenderanno probabilmente applicativi come compilatori o sistemi per la gestione documentale. Usare come benchmark delle applicazioni reali rende molto più difficile trovare metodi semplici per accelerare la loro esecuzione. Inoltre, se si trovano tecniche in grado di migliorare le prestazioni, è più probabile che queste abbiano impatto anche sugli altri programmi.

L'uso di benchmark le cui prestazioni dipendono da piccoli segmenti di codice incoraggia l'ottimizzazione sia delle architetture che dei compilatori che trattano questi segmenti. La procedura di ottimizzazione dei compilatori può riconoscere determinati segmenti di codice e generare una sequenza di istruzioni particolarmente efficiente per quei segmenti; allo stesso modo un progettista può cercare di far sì che determinate sequenze di istruzioni siano eseguite in modo particolarmente veloce perché compaiono in un benchmark. Recentemente, molte aziende hanno introdotto compilatori con ottimizzazioni particolari orientate a benchmark specifici; spesso tali ottimizzazioni devono essere abilitate con specifiche opzioni del compilatore, non usate nella compilazione degli altri programmi. Non è ben chiaro se il

compilatore produca codice efficiente o addirittura *codice corretto*, quando l'opzione è usata con un'applicazione reale; talvolta cercando di riprodurre codice altamente ottimizzato per i benchmark, gli ingegneri introducono ottimizzazioni errate. Ad esempio, alla fine del 1995, la Intel pubblicò una nuova misura delle prestazioni di un processore Pentium con un compilatore proprietario, non usato fuori dall'Intel, ottenuta con i benchmark SPEC per l'elaborazione di interi (vedere i Paragrafi 2.6 e 2.9 per un approfondimento su SPEC). Sfortunatamente il codice prodotto per uno dei benchmark era sbagliato; il fatto fu scoperto quando uno dei concorrenti esaminò il codice binario per capire come avesse potuto l'Intel rendere così veloce l'esecuzione di uno dei benchmark. Nel gennaio del 1996 l'Intel ammise l'errore e pubblicò nuovamente le prestazioni.

I programmi con un numero ridotto di istruzioni o quelli che spendono gran parte del tempo di esecuzione in un frammento di codice molto ridotto sono particolarmente vulnerabili a questo tipo di tentativi. L'insieme dei benchmark SPEC per la valutazione dei processori, ad esempio, fu orientata principalmente alle applicazioni reali; sfortunatamente la prima versione includeva un benchmark chiamato matrix300, che consisteva soltanto di una serie di moltiplicazioni fra matrici, in cui in realtà il 99% del tempo di esecuzione era relativo ad una sola linea di codice del benchmark. Il fatto che così tanto tempo fosse speso su una singola linea, facendo molte volte lo stesso calcolo, ha portato molte aziende a comprare o sviluppare una particolare tecnologia di compilatori con l'obiettivo di migliorare il tempo di esecuzione di questo benchmark. La figura 2.3 mostra il rapporto fra le prestazioni (l'inverso del tempo di esecuzione) per una macchina con due diversi compilatori: l'utilizzo del compilatore avanzato non ha praticamente effetto sul tempo di esecuzione di 8 dei 10 benchmark, ma migliora le prestazioni di matrix300 di un fattore maggiore di nove. Usando il compilatore avanzato, il programma matrix300 viene eseguito 729,8 volte più velocemente rispetto al tempo di riferimento ottenuto con un VAX-11/780, ma in generale le prestazioni della macchina sono molto inferiori: gli altri programmi sono eseguiti soltanto da 30 a 140 volte più velocemente. L'utente che si aspetta che il proprio programma venga eseguito 700 volte più rapidamente di quanto farebbe su un VAX-11/780 ha molte probabilità di rimanere deluso! Nella versione del 1992, il programma matrix300 fu escluso dall'insieme dei benchmark SPEC.

Allora perché nessuno esegue programmi reali per misurare le prestazioni? Una ragione è che piccoli benchmark sono attraenti quando si dà inizio ad un progetto, dato che sono sufficientemente ridotti da poter essere compilati e simulati facilmente, qualche volta anche a mano; sono particolarmente allettanti quando i progettisti lavorano su una nuova macchina, dato che i compilatori potrebbero non essere disponibili fino a progetto avanzato. Benchmark ridotti sono inoltre più facili da standardizzare rispetto ai grossi programmi, quindi è facile avere accesso alla relativa documentazione.

Benché l'utilizzo di benchmark ridotti all'inizio del processo di progettazione sia giustificato, non esiste una ragione valida per usarli per valutare sistemi di calcolo funzionanti. Nel passato era difficile ottenere applicazioni di grandi dimensioni facilmente portabili da una macchina all'altra, ma oggi questo non è più vero: usare piccoli programmi come benchmark era un tentativo per fare paragoni corretti fra macchine diverse, ma non usare applicazioni reali dopo gli studi di progetto iniziali aumenta la probabilità di fornire risultati fuorvianti e di portare fuori strada il progettista.

Una volta scelto un insieme di benchmark adeguati e dopo aver ottenuto le misure, si può scrivere un rapporto sulle prestazioni. Quando si descrive l'insieme

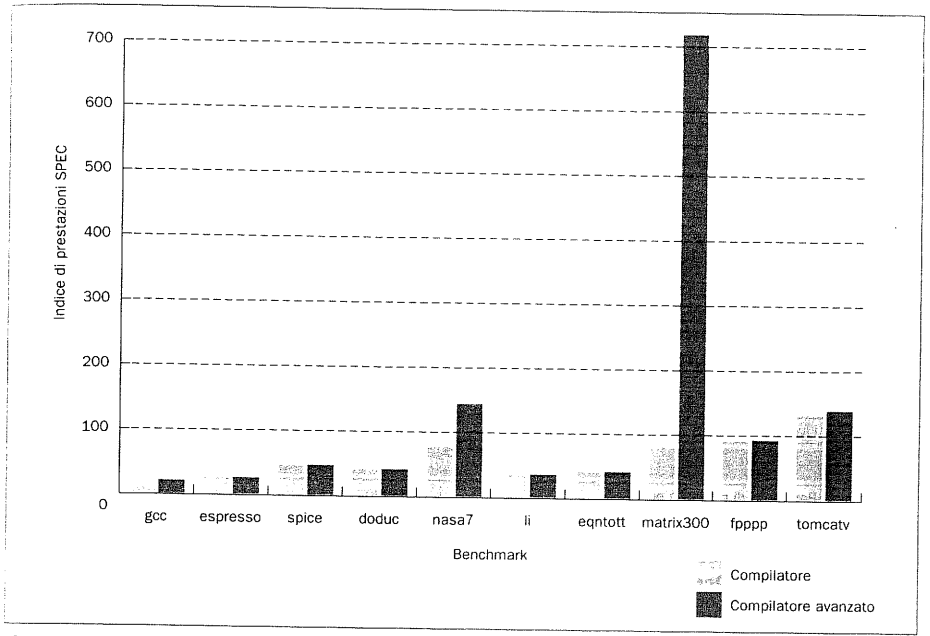


Figura 2.3 Le prestazioni relative SPEC89 riportate per la IBM Powerstation 550, utilizzando due diversi compilatori. I valori molto elevati su matrix300 (e su nasa7) risultano dall'applicazione di una tecnica di ottimizzazione rivolta a questi due kernel benchmark. Particolari opzioni vengono fornite al compilatore avanzato sia durante l'esecuzione di nasa7 che di matrix300, opzioni non utilizzate per gli altri benchmark. In entrambi i casi, il compilatore modifica i programmi trattando a blocchi le operazioni matriciali che si trovano nei cicli più interni. Queste trasformazioni diminuiscono in modo sostanziale il numero di accessi in memoria richiesti e fanno sì che dentro i cicli interni la probabilità che i dati richiesti siano nella cache passi da trascurabile a quasi certa. È interessante sapere che la motivazione originale per includere il programma matrix300 era quella di mettere alla prova la memoria del sistema di calcolo; al contrario nella pratica l'ottimizzazione riorganizza il programma in modo da minimizzare l'utilizzo della memoria. Questi dati comparvero in due rapporti SPEC durante l'autunno e l'inverno del 1991. La dipendenza di questi benchmark dall'ottimizzazione del compilatore e la relativa scarsa rilevanza dei risultati ottenuti dopo l'ottimizzazione hanno condotto all'esclusione di matrix300 dalla versione dei benchmark SPEC del 1992.

delle misure prestazionali effettuate è fondamentale considerare la *riproducibilità* delle stesse: deve essere illustrato tutto il necessario affinché un altro sperimentatore possa ripetere le prove e verificare il risultato. Si deve includere in particolare la versione del sistema operativo, i compilatori e i dati di ingresso, insieme alla configurazione della macchina. Come esempio si riporta in figura 2.4 la parte di un rapporto, riguardante la descrizione del sistema, relativo al benchmark SPEC.

2.5 Il confronto e la sintesi di prestazioni

Una volta che i programmi da utilizzare come benchmark sono stati scelti e si è deciso se si intende misurare il tempo di risposta o il throughput, si potrebbe pensare che il più sia cosa fatta. Invece si deve ancora decidere come esprimere in modo sintetico i risultati di un gruppo di benchmark.

Hardware	
Numero del modello	Powerstation 550
CPU	41,67-MHz POWER 4164
FPU	Integrata
Numero di CPU	1
Dimensione della cache per CPU	64K dati / 8K istruzioni
Memoria	64 MB
Sottosistema dischi	2 400-MB SCSI
Interfaccia di rete	–
Software	
Tipo e versione del sistema operativo	AIX v3.1.5
Versione del compilatore	AIX XL C/6000 Ver. 1.1.5 AIX XL Fortran Ver. 2.2
Altro software	Nessuno
Tipo del file system	AIX
Livello di firmware	–
Sistema	
Parametri di messa a punto	Nessuno
Carico di background	Nessuno
Stato del sistema	Multiutente (login di singoli utenti)

Figura 2.4 Descrizione del sistema usato per ottenere le migliori prestazioni di figura 2.3. Una nota allegata alla voce relativa al compilatore Fortran dice: "AIX XL Fortran Alpha versione 2.2 usato per il test." Benché non siano indicati parametri per la messa a punto, note aggiuntive descrivono alcune opzioni particolari fornite al compilatore nel caso particolare del benchmark.

Benché esprimere sinteticamente un insieme di misure comporti in genere una perdita di informazione, i venditori e anche gli utenti spesso preferiscono disporre di un solo valore sulla base del quale poter confrontare le prestazioni. Il punto chiave è allora come esprimere sinteticamente un insieme di misure prestazionali; la figura 2.5, estratta da un articolo sulla sintesi delle prestazioni, illustra alcune delle difficoltà che si incontrano quando si affronta tale problema.

Utilizzando la precedente definizione di *più veloce*, per le misure sui programmi della figura 2.5 si può affermare quanto segue:

- A è 10 volte più veloce di B sul programma 1;
- B è 10 volte più veloce di A sul programma 2.

Presa singolarmente, ciascuna delle due affermazioni è vera; prese insieme, invece, esse forniscono un'immagine confusa, sulla base della quale non sono affatto chiare le prestazioni relative dei calcolatori A e B.

	Calcolatore A	Calcolatore B
Programma 1(secondi)	1	10
Programma 2 (secondi)	1000	100
Tempo totale (secondi)	1001	110

Figura 2.5 Tempi di esecuzione di due programmi su due macchine diverse. Tratto dalla figura 1 di Smith [1988].

Il tempo totale di esecuzione: una misura coerente e sintetica

Il modo più semplice per esprimere in maniera sintetica le prestazioni relative di due sistemi è di riferirsi al tempo totale di esecuzione dei due programmi. Di conseguenza

$$\frac{\text{prestazioni}_B}{\text{prestazioni}_A} = \frac{\text{tempo di esecuzione}_A}{\text{tempo di esecuzione}_B} = \frac{1001}{110} = 9,1$$

In pratica, B è 9,1 volte più veloce di A se si considerano insieme sia il programma 1 che il programma 2.

Il valore di sintesi così introdotto è proporzionale al tempo di esecuzione, ossia alla misura di prestazioni prescelta. Se il carico di lavoro consiste nell'esecuzione dei programmi 1 e 2 per un numero uguale di volte, essa permette di predire per ciascuna macchina i tempi di esecuzione relativi al carico di lavoro.

Una media dei tempi di esecuzione che risulta direttamente proporzionale al tempo totale di esecuzione è la *media aritmetica* (MA):

$$\text{MA} = \frac{1}{n}\sum_{i=1}^{n} \text{tempo}_i$$

dove tempo_i è il tempo di esecuzione dell'i-esimo programma tra gli n che costituiscono il carico di lavoro. Dal momento che MA è la media dei tempi di esecuzione, un valore più basso indica un valore inferiore del tempo di esecuzione medio, quindi migliori prestazioni.

La media aritmetica è proporzionale al tempo di esecuzione se si assume che tutti i programmi inclusi nel carico di lavoro siano eseguiti lo stesso numero di volte. Se questo non corrisponde al carico di lavoro reale, si può assegnare un peso w_i a ciascun programma per indicare la frequenza di quel programma nel carico di lavoro prescelto. Se, per esempio, il 20% del carico di lavoro fosse costituito dal programma 1 e l'80% dal programma 2, allora i pesi sarebbero 0,2 e 0,8. Sommando tra loro i prodotti dei pesi per i tempi di esecuzione si ottiene un'immagine chiara delle prestazioni rispetto ad un certo carico di lavoro. Tale somma è denominata *media aritmetica pesata*. Per definire i pesi dei programmi si può fare in modo che il tempo di esecuzione dei benchmark sia uguale sulla macchina di riferimento. La media aritmetica standard rappresenta un caso particolare della media aritmetica pesata, nel quale tutti i pesi sono uguali; la media pesata sarà analizzata in maggior dettaglio negli Esercizi 2.29 e 2.30.

2.6 Un caso reale: i benchmark SPEC95 e le prestazioni dei processori moderni

L'insieme di benchmark più diffuso e di più vasta portata è la serie denominata SPEC (System Performance Evaluation Cooperative), creata per iniziativa di alcune industrie nel 1989 al fine di migliorare la misurazione e la pubblicazione dei dati sulle prestazioni delle CPU per mezzo di un processo di misura più controllato e dell'utilizzo di benchmark più realistici; una panoramica storica più dettagliata è riportata nel paragrafo 2.9.

La versione più recente dei benchmark SPEC è la serie SPEC95 (per i dettagli si veda la figura 2.6), che consiste di otto programmi per elaborazioni intere e dieci in

Benchmark	Descrizione
go	Intelligenza artificiale: gioco del Go (detto anche Otello o Reversi)
m88ksim	Simulatore del chip Motorola 88K: esecuzione di un programma
gcc	Compilatore Gnu C che genera codice SPARC
compress	Compressione e decompressione di un file in memoria
li	Interprete LISP
ijpeg	Compressione e decompressione di immagini grafiche
perl	Manipolazione di stringhe e numeri primi nel linguaggio di programmazione dedicato Perl
vortex	Programma di gestione di una base di dati
tomcatv	Programma per generazione di griglie
swim	Modello per acqua poco profonda con una griglia 513x513
su2cor	Fisica quantistica: simulazione Monte Carlo
hydro2d	Astrofisica: equazioni idrodinamiche di Naiver Stokes
mgrid	Risolutore multi-griglia in campo di potenziale 3-D
applu	Equazioni alle differenze parziali paraboliche/ellittiche
turb3d	Simulazione di turbolenza isotropica e omogenea in un cubo
apsi	Risoluzione di problemi di temperatura, velocità del vento e diffusione di agenti inquinanti
fpppp	Chimica quantistica
wave5	Fisica dei plasmi: simulazione di particelle elettromagnetiche

Figura 2.6 I benchmark per la CPU SPEC95. Gli otto benchmark per l'elaborazione intera riportati nella metà superiore della tabella sono scritti in C, mentre i dieci per l'elaborazione in virgola mobile sono in Fortran 77. Per ulteriori informazioni sul consorzio SPEC e sui suoi benchmark, si segua il collegamento alle relative pagine Web all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*.

virgola mobile, i cui risultati sono riportati separatamente. Le misure dei tempi di esecuzione sono innanzitutto normalizzate dividendo il tempo di esecuzione su una Sun SPARCstation 10/40 per il tempo di esecuzione sulla macchina in valutazione: tale normalizzazione fornisce una misura, detta *indice SPEC* (*SPEC ratio*), il cui vantaggio consiste nel fatto che valori numerici maggiori corrispondono a prestazioni migliori (in altre parole il rapporto SPEC è l'inverso del tempo di esecuzione). Le misure sintetiche SPECint95 o SPECfp95 si ottengono dalla media geometrica dei rapporti SPEC (si veda anche l'errore descritto nel paragrafo seguente per una valutazione della bontà delle medie geometriche).

I miglioramenti delle prestazioni della CPU per un determinato set di istruzioni possono derivare da tre cause:

1. Aumento della frequenza del clock.
2. Miglioramenti dell'organizzazione del processore mirati a ridurre il CPI.
3. Miglioramenti dei compilatori mirati a ridurre il numero di istruzioni o che generano istruzioni con un CPI medio inferiore (ad esempio utilizzando istruzioni più semplici).

Per meglio comprendere l'effetto di questi fattori, le figure 2.7 e 2.8 riportano le misure SPECint95 e SPECfp95 di una serie di processori Intel Pentium (montati nel sistema Intel XXpress) e Pentium Pro (montati nel sistema Intel Alder). Tenendo presente che SPEC richiede di eseguire i benchmark su un sistema reale e che il sottosistema di memoria ha un notevole impatto sulle prestazioni, se ne deduce che altri calcolatori basati sugli stessi processori possano avere livelli di prestazioni diversi. I sistemi Intel che sono stati qui valutati sono dotati di memorie e compila-

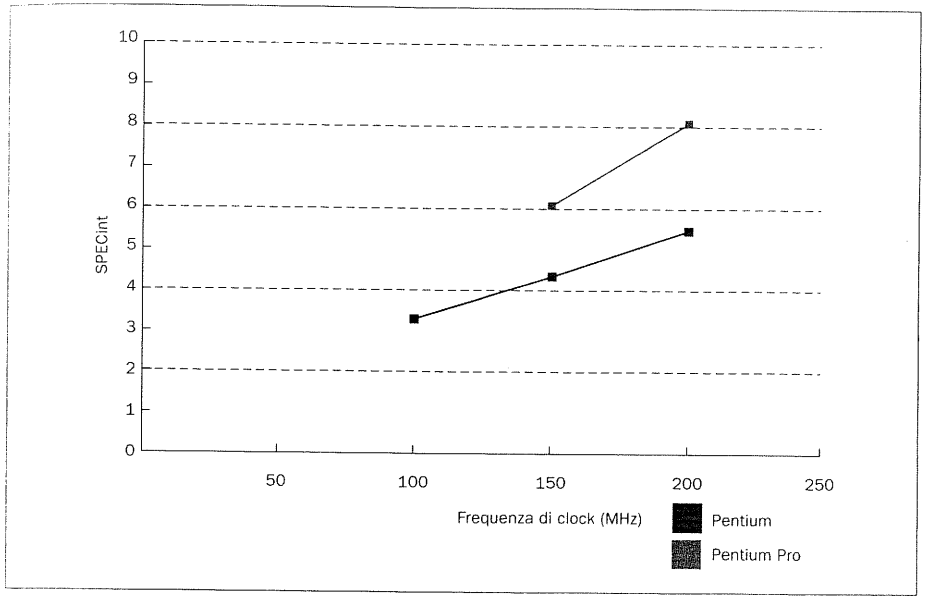


Figura 2.7 Valori SPECint95 per i processori Pentium e Pentium Pro con diverse frequenze di clock. Lo standard SPEC richiede due gruppi di misure: nella prima sono permesse ottimizzazioni spinte con opzioni specifiche per ciascun benchmark, nella seconda (detta SPECint_base95) sono permesse solo le opzioni di ottimizzazione standard; per i benchmark interi, i risultati su questi processori sono identici. I risultati sono reperibili seguendo il collegamento all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*.

tori fortemente ottimizzati (al contrario dei compilatori standard, forniti da terze parti, comunemente usati): la maggior parte dei sistemi che montano questi processori avranno prestazioni inferiori nei benchmark SPEC.

Sulla base di questi due grafici si possono fare diverse osservazioni importanti. Si nota immediatamente il miglioramento di prestazioni offerto dal Pentium Pro rispetto al Pentium: a parità di frequenza di clock la misura SPECint95 indica che il Pentium Pro è più veloce di 1,4-1,5 volte, mentre la misura SPECfp95 indica un valore di 1,7-1,8 volte. Sebbene i compilatori effettuino delle ottimizzazioni peculiari per ciascun processore, la maggior parte dell'incremento di prestazioni è dovuto ad una migliore organizzazione del Pentium Pro: alcuni di questi miglioramenti saranno illustrati nel Capitolo 6 a proposito della pipeline e nel Capitolo 7 a proposito del sottosistema di memoria.

Una seconda osservazione importante è che, quando viene aumentata la frequenza di clock di un certo fattore, il corrispondente fattore di incremento delle prestazioni è sempre minore. Ad esempio, al raddoppiare della frequenza di clock del Pentium da 100 MHz a 200 MHz, le prestazioni SPECint95 migliorano di un fattore 1,7 e quelle SPECfp95 di un mero 1,4! La causa di tale perdita di prestazioni va ricercata nella memoria: in generale, dato che la velocità della memoria principale non è aumentata, l'incremento della velocità del processore non fa che rendere più critico il collo di bottiglia causato dalla memoria; tale effetto è ancor più evi-

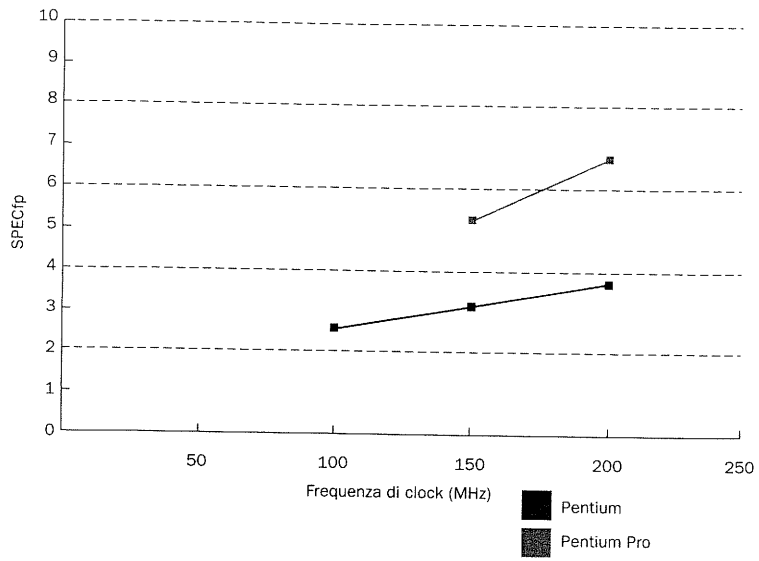


Figura 2.8 Valori SPECfp95 per i processori Pentium e Pentium Pro con diverse frequenze di clock. I risultati sono reperibili attraverso il collegamento al sito SPEC all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*.

dente sui benchmark per l'elaborazione in virgola mobile, poiché hanno dimensioni maggiori. Questo comportamento è un esempio di un principio generale, detto *legge di Amdahl*, che verrà esaminato in maggior dettaglio nel prossimo paragrafo.

In confronto al Pentium, le prestazioni del Pentium Pro sono maggiormente scalabili all'aumentare della frequenza di clock, sebbene non crescano mai quanto la velocità di clock stessa. Ad esempio, il miglioramento di SPECfp95 al passare da 150 MHz a 200 MHz è pari a 1,24, mentre il Pentium a parità di clock migliora di un fattore 1,18.

Nel Capitolo 7 si vedrà come sono progettati i sottosistemi di memoria di questi processori e come i progettisti possano ridurre l'effetto negativo di una memoria lenta al crescere della frequenza di clock. In particolare il Paragrafo 7.6 descrive alcune tecniche utilizzate dal Pentium Pro, che giustificano tra l'altro la miglior scalabilità rispetto al Pentium.

2.7 Errori e trabocchetti

Errori e trabocchetti che riguardano il rapporto costo/prestazioni hanno tratto in inganno molti progettisti di architetture dei calcolatori, autori compresi; di conseguenza questo paragrafo non soffre della mancanza di esempi significativi. Il

primo trabocchetto ha fatto cadere molti progettisti e rivela una relazione importante che riguarda la progettazione dei calcolatori.

Trabocchetto: aspettarsi che il miglioramento di una delle componenti di una macchina produca un aumento delle prestazioni proporzionale al miglioramento.

In questo trabocchetto sono incorsi progettisti sia di hardware che di software; un semplice problema di progetto lo illustra in modo efficace. Si supponga che un programma venga eseguito in 100 secondi su una macchina e che 80 di questi secondi siano impiegati nelle operazioni di moltiplicazione. Di quanto bisogna migliorare la velocità di esecuzione delle moltiplicazioni se si vuole che il programma venga eseguito 5 volte più rapidamente?

Il tempo di esecuzione del programma dopo il miglioramento è dato dalla seguente semplice equazione:

$$\text{tempo di esecuzione dopo il miglioramento} = \left(\frac{\text{tempo di esecuzione influenzato dal miglioramento}}{\text{ammontare del miglioramento}} + \text{tempo di esecuzione non influenzato}\right)$$

In questo problema si ha

$$\text{tempo di esecuzione dopo il miglioramento} = \left(\frac{80 \text{ secondi}}{n} + (100 - 80) \text{ secondi}\right)$$

Dal momento che si vuole che le prestazioni siano cinque volte migliori, il nuovo tempo di esecuzione deve essere di 20 secondi, per cui

$$20 \text{ secondi} = \frac{80 \text{ secondi}}{n} + 20 \text{ secondi}$$

$$0 = \frac{80 \text{ secondi}}{n}$$

In altre parole, non esiste un miglioramento quantificabile delle moltiplicazioni che permetta di ottenere un aumento delle prestazioni di un fattore cinque, se le moltiplicazioni rappresentano solo l'80% del carico di lavoro: dato un miglioramento, il possibile incremento delle prestazioni è limitato dall'ammontare dell'utilizzo dell'aspetto in questione. In informatica questo concetto è chiamato *legge di Amdahl* o più comunemente legge dei rendimenti decrescenti; altre implicazioni di questa relazione saranno esaminate negli esercizi da 2.41 a 2.46.

Un tema comune nella progettazione dell'hardware è un corollario della legge di Amdahl: *rendere veloce l'evento più frequente*. Questa semplice linea guida invita a tenere conto che in molti casi la frequenza con cui accade un evento può essere molto più elevata di un'altra. La legge di Amdahl ricorda che l'opportunità di operare un miglioramento dipende dalla durata dell'evento: rendere veloce l'evento più frequente contribuisce a migliorare le prestazioni molto di più che ottimizzare gli eventi rari. Ironicamente l'evento più frequente è spesso più facile da ottimizzare di quello raro.

Errore: le metriche indipendenti dall'hardware possono predire le prestazioni.

Dal momento che è molto difficile sia predire in modo accurato le prestazioni, sia operare confronti, molti progettisti e ricercatori hanno cercato di escogitare metodi

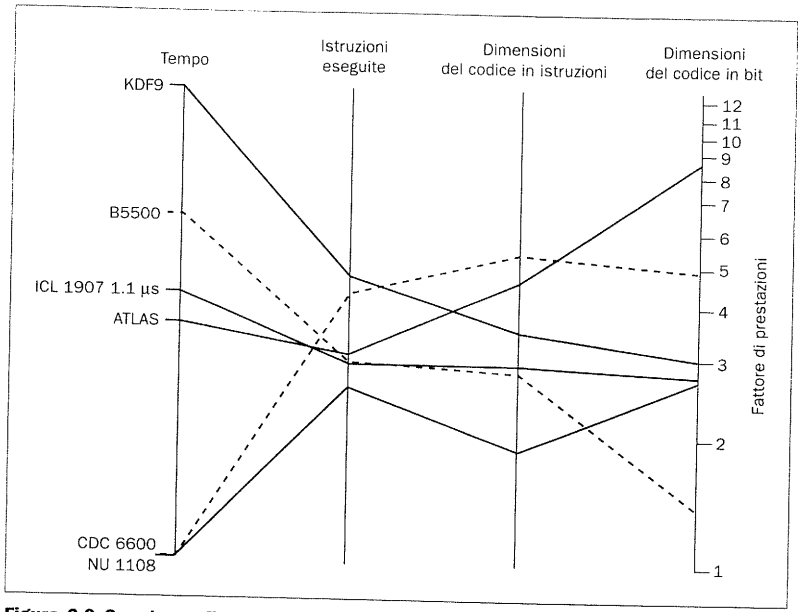


Figura 2.9 Questo grafico è tratto dalla copertina di *Algol 60 Compilation and Assessment* di B. A. Wichmann (pubblicato nel 1973). Il grafico mostra il tempo di esecuzione relativo, le istruzioni eseguite e la dimensione del codice sia in istruzioni che in bit per un insieme di programmi scritti in Algol 60 e fatti eseguire su sei macchine differenti. I risultati sono normalizzati ad una macchina di riferimento, dove un numero più elevato indica rispettivamente un maggiore tempo di esecuzione, un maggior numero di istruzioni eseguite ed una dimensione maggiore del codice. Il grafico mostra chiaramente che misure astratte come la dimensione del codice sono in scarsa relazione con misure di prestazioni quali il tempo di esecuzione o il numero di istruzioni eseguite. A dispetto dell'evidenza che dimensione del codice e tempo di esecuzione possano essere totalmente scorrelate, molti progettisti continuarono a dare rilievo alle dimensioni del codice per tutti gli anni '80. Grafico riprodotto per concessione della Academic Press.

per valutare le prestazioni indipendentemente dalla misura del tempo di esecuzione. Tali metodi sono usati frequentemente quando i progettisti confrontano più set di istruzioni per capirne gli effetti su implementazioni o sistemi software diversi; da questo sono in grado di ricavare conclusioni relative alle prestazioni ottenibili per i diversi set di istruzioni.

Uno di questi metodi, usato in passato, consisteva nell'utilizzare le dimensioni del codice come misura della velocità; secondo tale metodo il set di istruzioni che rende il programma più snello è il più veloce. Naturalmente la dimensione del programma compilato è importante quando lo spazio in memoria è una risorsa critica, ma non succede lo stesso per le prestazioni: infatti oggi le macchine più veloci tendono ad avere set di istruzioni che portano a programmi di dimensioni maggiori, ma che possono essere eseguiti più velocemente e con minor hardware.

L'evidenza dell'errore di usare le dimensioni del codice per misurarne la velocità di esecuzione si trova sulla copertina, mostrata in figura 2.9, di un libro pubblicato nel 1973. La figura mostra chiaramente la mancanza di una relazione diretta fra le dimensioni del codice e il tempo di esecuzione: ad esempio, i programmi del CDC 6600 sono oltre tre volte più grandi di quelli del Burroughs B5500, ma la macchina

CDC esegue i programmi Algol 60 quasi sei volte più velocemente del B5500, una macchina progettata specificamente per l'Algol 60.

Talvolta chi scrive i compilatori usa la dimensione del codice per scegliere fra due diversi segmenti di codice relativi alla stessa architettura. Benché questo sia meno fuorviante di cercare di confrontare le dimensioni del codice per architetture diverse, l'accuratezza con cui si possono predire le prestazioni partendo dalle dimensioni del codice è ampiamente variabile.

Trabocchetto: usare i MIPS come unità di misura per le prestazioni.

Sono state proposte numerose unità di misura con l'obiettivo di creare un sistema di riferimento per le prestazioni standard e di facile utilizzo; un risultato è stato che metriche semplici, valide in un contesto limitato, sono state pesantemente usate a sproposito. Tutte le proposte alternative all'utilizzo del tempo come misura delle prestazioni alla fine hanno condotto ad affermazioni fuorvianti, risultati distorti o interpretazioni scorrette.

Un'alternativa alle misure basate sul tempo per valutare le prestazioni sono i MIPS (milioni di istruzioni al secondo); dato un programma, i MIPS sono semplicemente

$$\text{MIPS} = \frac{\text{numero di istruzioni}}{\text{tempo di esecuzione} \times 10^6}$$

Questi MIPS sono anche chiamati *MIPS nativi*, per distinguerli da definizioni alternative che saranno discusse nel Paragrafo 2.9.

Dal momento che i MIPS sono una misura delle istruzioni eseguite, fissano le prestazioni in modo inverso al tempo di esecuzione: le macchine più veloci hanno un valore più elevato in termini di MIPS; un aspetto positivo dei MIPS è che rappresentano un concetto facile da comprendere, e il fatto che le macchine più veloci abbiano un valore in MIPS più elevato è particolarmente intuitivo.

Ci sono tre problemi nell'utilizzo dei MIPS come unità di misura per confrontare le macchine. Per prima cosa esso specifica una misura delle istruzioni eseguite, ma non tiene conto delle caratteristiche delle istruzioni: non si possono confrontare calcolatori con set di istruzioni diversi usando i MIPS, dal momento che il conteggio delle istruzioni sarebbe sicuramente diverso. In secondo luogo nello stesso calcolatore i MIPS variano a seconda del programma, quindi una macchina non ha un'unica misura di MIPS valida per tutti i programmi. Infine l'ultimo punto, il più importante, è che i MIPS possono variare in modo inverso alle prestazioni! Ci sono molti esempi di questo comportamento anomalo e uno di questi è riportato nel seguito.

I MIPS come unità di misura delle prestazioni

Si consideri la macchina con le tre classi di istruzioni e le misure di CPI dell'ultimo esempio del Paragrafo 2.3. Si supponga di misurare il codice dello stesso programma generato da due diversi compilatori e di ottenere i seguenti dati:

Codice — Numero di istruzioni (in miliardi) per ciascuna classe di istruzioni

	A	B	C
Compilatore 1	5	1	1
Compilatore 2	10	1	1

Si supponga che la frequenza di clock della macchina sia 500 MHz. Quale sequenza di codice sarà seguita più velocemente, misurata in MIPS? E quale, misurata secondo il tempo di esecuzione?

Soluzione

Per prima cosa si trova il tempo di esecuzione dei due diversi compilatori usando la seguente equazione:

$$\text{tempo di esecuzione} = \frac{\text{cicli di clock della CPU}}{\text{frequenza di clock}}$$

Per ricavare i cicli di clock della CPU si può usare una formula esaminata precedentemente:

$$\text{cicli di clock della CPU} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

$$\text{cicli di clock della CPU}_1 = (5 \cdot 1 + 1 \cdot 2 + 1 \cdot 3) \cdot 10^9 = 10 \cdot 10^9$$

$$\text{cicli di clock della CPU}_2 = (10 \cdot 1 + 1 \cdot 2 + 1 \cdot 3) \cdot 10^9 = 15 \cdot 10^9$$

A questo punto si può trovare il tempo di esecuzione per i due compilatori:

$$\text{tempo di esecuzione}_1 = \frac{10 \cdot 10^9}{500 \cdot 10^6} = 20 \text{ secondi}$$

$$\text{tempo di esecuzione}_2 = \frac{15 \cdot 10^9}{500 \cdot 10^6} = 30 \text{ secondi}$$

Si può quindi concludere che il compilatore 1 genera il programma più veloce, tenendo conto del tempo di esecuzione. Si calcoli ora il valore in MIPS di ciascuna versione del programma, usando l'equazione

$$\text{MIPS} = \frac{\text{numero di istruzioni}}{\text{tempo di esecuzione} \times 10^6}$$

$$\text{MIPS}_1 = \frac{(5+1+1)\ 10^9}{20 \cdot 10^6} = 350$$

$$\text{MIPS}_2 = \frac{(10+1+1) \cdot 10^9}{30 \cdot 10^6} = 400$$

Il codice generato dal compilatore 2 ha un valore di MIPS più elevato, ma il codice generato dal compilatore 1 viene eseguito più velocemente!

Come mostrano esempi di questo tipo, i MIPS possono fallire nel dare una reale misura delle prestazioni, anche quando si confrontano due versioni dello stesso programma sulla stessa macchina. Nel Paragrafo 2.9 verranno discussi altri usi del termine *MIPS*, che possono essere altrettanto fuorvianti.

Errore: i benchmark sintetici sono in grado di predire le prestazioni.

I benchmark sintetici sono programmi artificiosi costruiti nel tentativo di confrontare le caratteristiche di vasti insiemi di programmi. L'obiettivo è creare un singolo programma benchmark in cui la frequenza con cui viene eseguito un costrutto sia pari a quella verificata per lo stesso costrutto in un vasto insieme di programmi; Whetstone e Dhrystone sono i benchmark artificiali più conosciuti. Whetstone era basato su misurazioni di programmi Algol in ambiente scientifico e ingegneristico; fu poi tradotto in Fortran diffondendosi moltissimo. Ispirandosi a Whetstone, Dhrystone fu creato come benchmark in ambienti di programmazione di sistemi e si basava su un insieme di misure di frequenza rese pubbliche in precedenza. Dhrystone inizialmente fu scritto in Ada e in seguito convertito in C, diffondendosi ulteriormente.

Uno dei maggiori inconvenienti dei benchmark artificiali è che nella realtà nessun utente eseguirà mai uno di essi come applicazione, dal momento che programmi di questo tipo non calcolano nulla che un utente possa considerare anche solo lontanamente interessante. In più, dal momento che non sono veri programmi, di solito non rispecchiano il comportamento del programma al di là di quanto è stato considerato durante la loro creazione. Infine le ottimizzazioni del compilatore e dell'hardware possono gonfiare le prestazioni di questi benchmark al di là di quanto le stesse ottimizzazioni otterrebbero sui programmi reali: naturalmente, dato che non sono programmi normali, possono non premiare ottimizzazioni ricorrenti nei programmi reali. Di seguito sono riportati alcuni esempi di come Dhrystone possa distorcere l'importanza di alcune ottimizzazioni:

- L'ottimizzazione dei compilatori porta facilmente a scartare il 25% del codice Dhrystone; in questa situazione ricadono ad esempio i cicli eseguiti solo una volta, che rendono inutili le istruzioni aggiuntive relative alla gestione dei cicli. Riferendosi a questi problemi, gli autori del benchmark "richiedono" che vengano riportati i risultati sia sul codice ottimizzato che su quello non ottimizzato. In più essi "proibiscono" la pratica dell'ottimizzazione attraverso l'espansione in linea delle procedure, dal momento che la semplice struttura delle procedure inserite in Dhrystone permette l'eliminazione di tutte le chiamate a procedura senza quasi aumentare la dimensione del codice.
- Un compilatore C sembra includere ottimizzazioni mirate proprio a Dhrystone: se durante la compilazione è inserita l'opzione appropriata, il compilatore converte la porzione della versione in C del benchmark che copia una stringa di byte di lunghezza variabile (terminata da un simbolo di fine stringa) in un ciclo che trasferisce un numero fisso di parole. Il compilatore assume inoltre che la sorgente e la destinazione della stringa in memoria siano allineate a parole. Nonostante sia stimato che un numero variabile fra il 99,70% e il 99,98% delle tipiche operazioni di copia di stringhe non possano usare tale ottimizzazione, questo semplice accorgimento può condurre ad un miglioramento dal 20% al 30% nelle prestazioni complessive di Dhrystone.

Le piccole dimensioni e la struttura semplicistica dei benchmark artificiali li rendono particolarmente vulnerabili ad operazioni di questo genere.

Trabocchetto: usare la media aritmetica dei tempi di esecuzione normalizzati per predire le prestazioni.

In questo trabocchetto sono caduti molti ricercatori, incluso uno degli autori di questo libro. Un metodo allettante per presentare le prestazioni di una macchina è normalizzare il tempo di esecuzione rispetto ad una macchina di riferimento, come è stato fatto per ottenere l'indice SPEC, e poi prendere la media dei tempi di esecuzione normalizzati. Se però la media utilizzata è quella aritmetica il risultato dipenderà dalla scelta della macchina di riferimento. Ad esempio la figura 2.10 riporta i tempi di esecuzione della figura 2.5 normalizzati sia su A che su B e il calcolo della media aritmetica; quando i tempi di esecuzione sono normalizzati ad A, la media aritmetica indica che A è più veloce di B di un fattore 5,05 a 1, che è l'inverso del rapporto fra le prestazioni. Quando i tempi di esecuzione sono normalizzati a B, si vede che *B è più veloce esattamente dello stesso fattore*. Chiaramente i due risultati non possono essere contemporaneamente corretti!

Il problema nasce dall'aver usato la media aritmetica in un rapporto. I risultati normalizzati devono invece essere combinati utilizzando la media geometrica; la formula della media geometrica è

$$\sqrt[n]{\prod_{i=1}^{n} \text{rapporto fra i tempi di esecuzione}_i}$$

dove il rapporto fra i tempi di esecuzione$_i$ è il tempo di esecuzione, normalizzato rispetto alla macchina di riferimento, dell'i-esimo degli n programmi del carico di lavoro e

$$\prod_{i=1}^{n} a_i \quad \text{indica il prodotto} \quad a_1 \times a_2 \times \mathrm{K} \times a_n$$

La media geometrica è indipendente dalla serie di dati usata per la normalizzazione, godendo della proprietà

$$\frac{\text{media geometrica}(X_i)}{\text{media geometrica}(Y_i)} = \text{media geometrica}\left(\frac{X_i}{Y_i}\right)$$

che dice che il rapporto fra le medie geometriche è uguale alla media geometrica dei rapporti. La media geometrica produce quindi lo stesso risultato relativo sia che si normalizzi rispetto ad A che rispetto a B, come si può vedere nell'ultima riga di figura 2.10. Quando i tempi di esecuzione sono normalizzati, solo la media geometrica può essere usata per sintetizzare i risultati in modo coerente; sfortunatamente, come verrà mostrato negli esercizi, la media geometrica non tiene traccia del tempo totale di esecuzione e non può quindi essere usata per predire il tempo di esecuzione relativo di un carico di lavoro.

	Tempo X	Tempo su B	Normalizzato rispetto ad A		Normalizzato rispetto a B	
			A	B	A	B
Programma 1	1	10	1	10	0,1	1
Programma 2	1000	100	1	0,1	10	1
Media aritmetica dei tempi o dei tempi normalizzati	500,5	55	1	5,05	5,05	1
Media geometrica dei tempi o dei tempi normalizzati	31,6	31,6	1	1	1	1

Figura 2.10 Tempi di esecuzione della figura 2.5 normalizzati rispetto a ciascuna delle due macchine. Per ciascuna colonna è calcolata la media aritmetica. Mentre la media aritmetica varia normalizzando rispetto ad A o a B, la media geometrica è costante, indipendentemente dalla normalizzazione.

Errore: la media geometrica dei rapporti fra i tempi di esecuzione è proporzionale al tempo di esecuzione totale.

Il vantaggio della media geometrica è la sua indipendenza sia dal tempo di esecuzione di un singolo programma che dalla macchina usata per la normalizzazione. L'inconveniente di utilizzare la media geometrica dei tempi di esecuzione è che viola il principio fondamentale della misurazione delle prestazioni: non predice il tempo di esecuzione. Nella figura 2.10 la media geometrica suggerisce che le prestazioni delle macchine A e B sui programmi 1 e 2 siano le stesse, mentre la media aritmetica dei tempi di esecuzione, che si sa essere proporzionale al tempo di esecuzione totale, suggerisce che la macchina B sia 9,1 volte più veloce della macchina A! Se si utilizza il tempo di esecuzione complessivo come misura delle prestazioni, A e B avrebbero le stesse prestazioni solo per un carico di lavoro che facesse eseguire il primo programma 100 volte più frequentemente del secondo!

In generale, nessun carico di lavoro su tre o più macchine rispetta le previsioni ottenute con la media geometrica dei tempi di esecuzione normalizzati. La soluzione ideale consiste nel misurare un carico di lavoro reale e pesare i programmi a seconda della frequenza di esecuzione; se questo non può essere fatto, normalizzare in modo che lo stesso tempo sia speso sullo stesso programma in più macchine rende per lo meno espliciti i relativi pesi e predice il tempo di esecuzione di un carico di lavoro con una combinazione analoga. Se i risultati devono essere normalizzati rispetto ad una macchina specifica, è bene prima sintetizzare le prestazioni con misure pesate in modo appropriato e poi normalizzarle.

Quadro d'insieme

Il tempo di esecuzione è l'unica misura delle prestazioni valida e incontestabile. Molti altri metodi di misura sono stati proposti, ma tutti si sono poi dimostrati carenti. Talvolta questi metodi, non rispecchiando i tempi di esecuzione, sono stati invalidati sin dall'inizio; in altri casi un sistema di misura valido solo in un contesto limitato è stato usato al di là di tale contesto o senza i chiarimenti necessari alla sua convalida.

In modo analogo, qualunque misura che sintetizzi le prestazioni deve rispecchiare i tempi di esecuzione. La somma aritmetica pesata sintetizza le prestazioni tenendo traccia dei tempi di esecuzione; grazie all'utilizzo dei pesi, tale media può compensare tempi di esecuzione diversi, bilanciando i contributi di ciascun benchmark all'interno della valutazione complessiva.

Note conclusive

Benché in questo capitolo ci si sia concentrati sulle prestazioni e sulla loro valutazione, si deve tenere conto che un'attività di progetto finalizzata alle sole prestazioni e che non tenga anche conto dei costi è del tutto improponibile: tutti i progettisti di calcolatori devono trovare un compromesso tra costi e prestazioni. Ovviamente esi-

ste un settore, quello del *progetto per alte prestazioni*, nel quale le prestazioni sono l'obiettivo primario ed il costo quello secondario; gran parte dell'industria dei supercomputer svolge la sua attività di progetto seguendo questo principio. All'altro estremo si trova l'area dei *progetti a basso costo*, nella quale i costi hanno la precedenza sulle prestazioni; appartengono a questa categoria i calcolatori di fascia bassa, come i cloni dei PC IBM e gran parte dei calcolatori incorporati nei prodotti (embedded). Tra questi due estremi va collocato il *progetto costo/prestazioni*, nel quale il progettista cerca un equilibrio tra prestazioni e costi; vari esempi tratti dall'industria delle workstation rappresentano casi tipici del tipo di compromesso con il quale i progettisti di questo settore devono fare quotidianamente i conti.

Si è visto in questo capitolo che esiste un metodo affidabile per determinare e documentare le prestazioni, usando come metrica le prestazioni di programmi reali. Il tempo di esecuzione è correlato ad altre importanti misure dalla seguente equazione:

$$\frac{\text{secondi}}{\text{programma}} = \frac{\text{istruzioni}}{\text{programma}} \times \frac{\text{cicli di clock}}{\text{istruzione}} \times \frac{\text{secondi}}{\text{ciclo di clock}}$$

Si farà uso molte volte di questa equazione e dei termini che la compongono; si ricordi tuttavia che i singoli termini non definiscono le prestazioni: soltanto il prodotto complessivo, corrispondente al tempo di esecuzione, ne è una misura affidabile.

Ovviamente, la semplice conoscenza dell'equazione sopra riportata non è sufficiente per guidare il progetto e la valutazione di un calcolatore: si deve innanzitutto comprendere come i diversi aspetti di un progetto influiscono su ciascuno di questi parametri chiave. Questo comporta un'ampia gamma di considerazioni, dagli effetti della scelta del set di istruzioni sul numero di istruzioni dinamicamente eseguite, all'impatto della pipeline e dei sistemi di memoria sul CPI, all'interazione tra tecnologia ed organizzazione architetturale, sulla base della quale si determina la frequenza di clock. La capacità di progettare un calcolatore non consiste nel sistemare dei numeri in un'equazione che fornisce dati sulle prestazioni, ma nel determinare con precisione come le varie alternative di progetto possano influenzare le prestazioni ed il costo.

Gran parte degli utilizzatori di calcolatori si preoccupano sia del costo che delle prestazioni; la comprensione delle relazioni tra i vari aspetti di un progetto e le sue prestazioni è un'impresa impegnativa, ma individuare il costo delle varie alternative possibili in un progetto è spesso un problema ancora più complesso. Il costo di una macchina è influenzato non solo dal costo delle componenti, ma anche dal costo del lavoro per montare la macchina, da quello per la ricerca e lo sviluppo, da quelli legati agli aspetti commerciali e di marketing e dal margine di profitto desiderato. Infine, a causa dei rapidi cambiamenti nelle tecnologie, la scelta ottimale fatta oggi diviene generalmente subottimale nell'arco di sei mesi o un anno.

Il progetto di un calcolatore verrà sempre valutato in termini di costi e prestazioni e proprio nell'individuazione del miglior compromesso risiederà sempre la capacità di progettare i calcolatori, esattamente come avviene in ogni attività ingegneristica.

2.9 Inquadramento storico e approfondimenti

Sin dalla nascita dell'informatica, i progettisti si sono posti obiettivi riguardanti le prestazioni: l'ENIAC doveva essere 1000 volte più veloce del Mark-I di Harvard, mentre l'IBM Stretch (7030) doveva essere 100 volte più veloce del sistema più

veloce allora esistente. Ciò che non era affatto chiaro, però, è come tali prestazioni dovessero essere misurate.

La prima misura di prestazioni fu il tempo richiesto per eseguire una singola operazione, ad esempio un'addizione; dato che la maggior parte delle istruzioni richiedeva lo stesso tempo per essere eseguita, la durata di ciascuna forniva indicazioni anche per le altre. Via via che i tempi di esecuzione per le varie istruzioni iniziarono a diversificarsi, il tempo richiesto da una sola di esse cessò di essere utile ai fini dei confronti.

Per tenere in debito conto queste differenze, si calcolò una *combinazione di istruzioni* (*instruction mix*) misurando la frequenza relativa delle istruzioni in un calcolatore, all'interno di diversi programmi. Il prodotto del tempo di esecuzione di un'istruzione per il suo peso nella combinazione forniva il *tempo di esecuzione medio per istruzione* (se misurata in cicli di clock, questa quantità è identica al CPI); il confronto dei tempi di esecuzione medi risultò più preciso del confronto tra le singole istruzioni, anche per via della somiglianza tra i diversi insiemi di istruzioni. Il passo dal tempo di esecuzione medio ai MIPS (milioni di istruzioni per secondo) fu breve: avendo il vantaggio di essere facilmente comprensibili ed intuitivi, i MIPS si diffusero rapidamente.

MIPS, MOPS ed altri FLOPS

Una delle definizioni di MIPS che si è rivelata particolarmente fuorviante è quella dei *MIPS di picco*; i MIPS di picco si ottengono scegliendo una combinazione di istruzioni che minimizzi il CPI, anche se tale combinazione si può rivelare del tutto inutile. Ricordando l'esempio precedente, il valore dei MIPS di picco è uguale per entrambi i sistemi: 500 MIPS; per ottenere 500 MIPS con un clock a 500 MHz, il CPI del programma deve essere pari ad 1: un programma con un CPI uguale a 1 deve necessariamente essere composto di sole istruzioni di tipo A!

In pratica spesso si commercializzano i processori attirando i clienti con la misura dei MIPS di picco, che può distorcere notevolmente il quadro reale delle prestazioni. Quando fu annunciato l'Intel i860 nel febbraio del 1989, l'annuncio commerciale citò le prestazioni di picco in confronto ad altri processori: l'i860 era capace di eseguire fino a due operazioni in virgola mobile ed una intera nello stesso ciclo di clock; con una frequenza di clock pari a 50 MHz, si calcolarono 100 MFLOPS (milioni di istruzioni in virgola mobile per secondo) e 150 MOPS (milioni di operazioni per secondo). I primi calcolatori basati su i860 (che utilizzavano dispositivi a 40 MHz) divennero disponibili per una valutazione all'inizio del 1991. Come termine di confronto, un sistema MIPS basato su un processore R3000 a 33 MHz, immesso sul mercato all'incirca nello stesso periodo, aveva prestazioni di picco pari a circa 16 MFLOPS e 33 MOPS. Sulla base delle prestazioni di picco il sistema basato su i860 avrebbe dovuto essere più di 5 volte più veloce del sistema basato su R3000, mentre i benchmark SPEC dimostrarono che in realtà era il sistema R3000 ad essere del 15% più veloce! Sebbene i MIPS di picco siano una misura fondamentalmente inutile, i produttori di calcolatori commercializzano ancora i loro prodotti utilizzandoli come metrica, spesso dimenticando di includere il termine "di picco"!

Un tentativo fatto per continuare ad utilizzare il termine MIPS, rendendolo significativo al variare dei set di istruzioni, fu di individuare una definizione di MIPS relativa ad un elaboratore identificato di comune accordo come termine di riferimento, in modo simile a quanto avviene nelle misure SPEC. I cosiddetti *MIPS relativi* sono quindi stati definiti come segue:

$$\text{MIPS relativi} = \frac{\text{tempo}_{\text{riferimento}}}{\text{tempo}_{\text{in valutazione}}} \times \text{MIPS}_{\text{riferimento}}$$

dove:

$\text{tempo}_{\text{riferimento}}$ = tempo di esecuzione di un programma sul calcolatore di riferimento

$\text{tempo}_{\text{in valutazione}}$ = tempo di esecuzione dello stesso programma sul calcolatore in corso di valutazione

$\text{MIPS}_{\text{riferimento}}$ = valutazione concordata, in MIPS, del calcolatore di riferimento

I MIPS relativi sono proporzionali al tempo di esecuzione *solamente* per un determinato programma e per un determinato valore dei suoi dati di ingresso. Anche quando questi due elementi sono identificati, diviene sempre più difficile reperire calcolatori di riferimento per via del loro invecchiamento: negli anni '80 l'elaboratore di riferimento più utilizzato era il VAX-11/780, a cui fu attribuito il valore di 1 MIPS per ragioni che risulteranno chiare tra poco, ma ora è difficile reperirne uno ancora funzionante. Inoltre sarebbe più corretto eseguire il programma sul vecchio calcolatore con versioni aggiornate del compilatore e del sistema operativo oppure si dovrebbero fissare anche le versioni del software in modo da evitare che il calcolatore di riferimento divenga più veloce al passare del tempo? Si potrebbe anche essere tentati di generalizzare i risultati ottenuti da una misura dei MIPS relativi esprimendo un giudizio sulle prestazioni, sebbene siano possibili ampie variazioni di prestazioni già all'interno di un insieme di benchmark.

La diffusione dei MIPS relativi come misura di prestazioni è un esempio di come la valutazione delle prestazioni non evolva necessariamente secondo logica. Negli anni '70 si utilizzavano i MIPS esclusivamente per confrontare le prestazioni delle implementazioni di sistemi IBM 360/370: dato che queste erano basate su architetture identiche (e quindi su identici numeri di istruzioni) la metrica aveva una sua validità. La nozione di MIPS relativi fu un modo di estendere la definizione, così intuitiva, di MIPS: nel 1987, poco prima dell'annuncio del VAX-11/780, la DEC eseguì alcuni benchmark e confrontò le prestazioni con un IBM 370/158. Dal momento che i canali commerciali dell'IBM indicavano il modello 370/150 come un calcolatore ad 1 MIPS e poiché i benchmark venivano eseguiti alla stessa velocità, anche la DEC pubblicizzò il VAX-11/780 come un calcolatore da 1 MIPS.

La diffusione del VAX-11/780 ne fece una valida macchina di riferimento per i MIPS relativi, soprattutto perché il valore dei MIPS relativi riferiti ad un calcolatore da 1 MIPS era semplice da calcolare: se un elaboratore era cinque volte più veloce del VAX-11/780, la sua valutazione, per quel benchmark, risultava di 5 MIPS. La valutazione di 1 MIPS fu ritenuta valida per quattro anni, fino a quando Joel Elmer della DEC non misurò il VAX-11/780 in ambiente timesharing ed il reale valore in MIPS risultò essere di 0,5. Gli esemplari successivi di VAX, che eseguivano 3 milioni di istruzioni VAX al secondo, vennero quindi valutati a 6 MIPS in quanto erano sei volte più veloci del VAX-11/780.

Gli anni '70 e '80 videro la crescita del mercato dei supercomputer, caratterizzati da elevate prestazioni nel caso di programmi facenti un pesante uso di elaborazione in virgola mobile: in questo caso il tempo di esecuzione medio delle istruzioni ed i MIPS erano misure decisamente inadatte, per cui si definirono i MFLOPS (milioni di operazioni in virgola mobile per secondo). Sfortunatamente, gli utilizzatori si dimenticarono presto del programma utilizzato nelle valutazioni ed i canali commerciali decisero di iniziare a citare i MFLOPS di picco nelle dispute sulle prestazioni dei supercomputer: l'utilizzo dei MFLOPS ed i relativi problemi sono trattati negli esercizi di questo capitolo.

Alla ricerca del programma tipico

Via via che i processori divennero più sofisticati e iniziarono a dipendere dalle gerarchie di memoria (trattate nel Capitolo 7) e dalle pipeline (Capitolo 6), cessò di esistere un singolo valore per il tempo di esecuzione di una singola istruzione: a partire dal manuale di istruzioni e pur conoscendo la combinazione di istruzioni, non era quindi più possibile risalire né al tempo di esecuzione né ai MIPS. Sebbene oggigiorno possa sembrare ovvio che la soluzione risiedeva nell'identificazione di un insieme di applicazioni reali che si potessero utilizzare come benchmark standard, fino a tempi più recenti questo era un compito di notevole difficoltà: la grande variabilità nei sistemi operativi e nei linguaggi di programmazione rendevano ardua la realizzazione di programmi che potessero migrare su diversi calcolatori semplicemente ricompilandoli. Si assisté invece all'utilizzo di benchmark basati sui cosiddetti programmi "sintetici", un esempio dei quali è Whetstone, creato per la valutazione di programmi scientifici scritti in Algol 60 (si veda la descrizione di Curnow e Wichmann [1976]); il programma fu convertito in Fortran ed ampiamente utilizzato per caratterizzare le prestazioni di applicazioni scientifiche. Le prestazioni Whetstone sono tipicamente espresse in termini di Whetstone al secondo, ossia il numero di iterazioni, in un secondo, del benchmark! Più di recente fu sviluppato il benchmark sintetico Dhrystone (la descrizione del programma e della metodologia adottata sono riportati da Weicker [1984]).

Circa in contemporanea allo sviluppo di Whetstone, si diffuse il concetto di *kernel benckmark*. Un kernel (termine inglese che significa nucleo) è un piccolo frammento di programma, il cui tempo di esecuzione è rilevante, che viene estratto ed utilizzato come benchmark: questo approccio fu utilizzato principalmente per la valutazione di sistemi di fascia alta, soprattutto supercomputer. Gli esempi più noti sono i Loop Livermore e Linpack: i Loop Livermore sono composti di una serie di 21 piccoli frammenti di ciclo, mentre Linpack è una parte di una libreria di subroutine per algebra lineare. Gli utilizzi migliori dei kernel si riscontrano nell'isolare le prestazioni di caratteristiche particolari del calcolatore e nello spiegare le ragioni di differenze riscontrabili nelle prestazioni di programmi reali. Dato che molte applicazioni scientifiche spendono molto tempo nell'esecuzione di una parte relativamente ridotta del codice, in questa classe di applicazioni è spesso utilizzata una valutazione tramite kernel. Sebbene i kernel possano giovare alla comprensione, hanno spesso l'effetto di sovrastimare le prestazioni delle applicazioni reali: ad esempio i supercomputer odierni ottengono prestazioni molto vicine a quelle di picco nell'eseguire un kernel, ma in un'applicazione reale le prestazioni diventano una piccola frazione di quelle di picco.

Alla ricerca del programma semplice

Un ulteriore passo falso nella ricerca di metodi di valutazione migliori è rappresentato dall'utilizzo di programmi "giocattolo" come benchmark. Tali programmi sono solitamente composti di 10-100 linee di codice e producono un risultato noto a priori: tipici esempi sono il setaccio di Erastotene, il gioco del 15 e il Quicksort, in quanto sono piccoli, facili da compilare e possono essere eseguiti praticamente su ogni calcolatore. La maggior diffusione di questi programmi si ebbe nei primi anni '80, quando nelle università si progettavano i primi processori RISC: le ridotte

dimensioni di questi programmi facilitavano la loro esecuzione da parte di un simulatore. Purtroppo gli autori di questo testo devono ammettere di aver giocato un ruolo a favore della loro diffusione, avendoli utilizzati per confrontare le prestazioni ed addirittura avendo distribuito raccolte di tali programmi. Ancor più imbarazzante è il fatto che molta gente continui ad utilizzarli: dobbiamo dire che la lezione è stata compresa e che il miglior uso di questi programmi è come semplici esercizi di programmazione.

Sintetizzare è rischioso

Quasi tutte le azioni legate alla misurazione ed alla pubblicazione delle prestazioni hanno dato origine a controversie e discussioni, compresa la modalità di calcolo delle prestazioni complessive. I metodi utilizzati sono stati i più diversi, come la media aritmetica delle prestazioni normalizzate, la media armonica dei rapporti, la media geometrica del tempo di esecuzione normalizzato, il tempo di esecuzione totale. Il problema è discusso in diversi testi citati alla fine di questo paragrafo, tra cui l'articolo di Smith [1988], sul quale è basato l'approccio del Paragrafo 2.5.

SPECulazioni sulle prestazioni

Un progresso notevole nella valutazione delle prestazioni si ebbe con la formazione del gruppo SPEC (System Performance Evaluation Cooperative) nel 1988. SPEC comprende rappresentanti di diverse industrie informatiche (i soci fondatori furono Apollo/Hewlett-Packard, DEC, MIPS e Sun) che si sono accordate su un insieme di programmi reali e di valori di ingresso. Vale la pena di sottolineare che SPEC non sarebbe potuto nascere se non grazie alla realizzazione di sistemi operativi portabili ed alla diffusione dei linguaggi di programmazione ad alto livello. Oggigiorno, infatti, i compilatori sono considerati come facenti parte delle prestazioni di un elaboratore e devono essere considerati in ogni valutazione.

La storia insegna che sebbene SPEC possa essere utile per i calcolatori attuali, così com'è non potrà soddisfare i requisiti della prossima generazione. Nel 1991 fu aggiunta una misura del throughput, basata sull'esecuzione di diverse versioni dello stesso benchkmark, che risulta utile nel valutare l'utilizzo, in ambiente time-sharing, di un singolo processore o di un sistema multi-processore. Furono aggiunti anche altri benchmark specifici per valutare il sistema operativo o il sottosistema di Input/Output. Fu poi deciso, basandosi anche sui risultati mostrati in figura 2.3, di escludere matrix300 e di aumentare il numero di benchmark: la difficoltà nel trovare dei buoni benchmark fece sì che la versione iniziale (detta SPEC89) ne contenesse sei per elaborazioni in virgola mobile, ma solo quattro per elaborazioni intere, ed il valore complessivo calcolato come media geometrica dei tempi di esecuzione normalizzati rispetto al VAX-11/780 favoriva calcolatori con migliori prestazioni sui numeri in virgola mobile.

Nel 1992 fu introdotto un nuovo insieme di benchmark (chiamato SPEC92) che comprendeva programmi nuovi, escludeva matrix300 e forniva valori medi separati (SPECint e SPECfp) per programmi interi e a virgola mobile. Inoltre fu aggiunta la misura SPECbase, in cui si vietano opzioni di ottimizzazione adattate ai benchmark, per permettere di ottenere delle cifre più vicine alle prestazioni dei programmi reali. In particolare, la misura SPECfp tende a variare notevolmente rispetto alla misura base, tipicamente dal 15% al 30%.

Un ulteriore aggiornamento si ebbe nel 1995, quando vennero aggiunti altri benckmark, sia interi che in virgola mobile e se ne eliminarono alcuni che si erano

rivelati errati o che avevano tempi di esecuzione troppo ridotti, visto l'incremento nelle prestazioni di circa un fattore 20 rispetto alla prima versione di SPEC. Anche il calcolatore di riferimento per la normalizzazione venne cambiato, adottando una Sun SPARCstation 10/40, perché iniziava ad essere difficile reperire esemplari funzionanti di VAX-11/780!

Il consorzio SPEC in più ha raccolto altre serie di benchmark, oltre a quelle originarie, che erano dedicate alle prestazioni della CPU. Il benchmark SDM (Systems Development Multitasking) contiene due versioni sintetiche di un carico di lavoro tipico dello sviluppo di software (editing, compilazione, esecuzione, comandi di sistema). La serie di benchmark sintetici SFS (System-level File Server) genera un carico di lavoro adatto a misurare le prestazioni di un file server. Questi due benchmark contengono una parte rilevante dedicata alle operazioni di I/O ed al sistema operativo, a differenza dei benchmark per la CPU. I due benchmark più recenti di SPEC costituiscono la serie SPEChpc96 e mirano a valutare le prestazioni con carichi di lavoro che si avvicinano ad elaborazioni scientifiche di alto livello; nel prossimo futuro, SPEC sta elaborando benchmark per nuove funzioni, come ad esempio i server Web.

La creazione e lo sviluppo di questi insiemi di benchmark è divenuto un compito difficile ed impegnativo: da un consorzio nato come impegno comune, in buona fede, tra alcune aziende, SPEC ha assunto importanza nel settore propagandistico e di marketing comparativo. La scelta dei benchmark e delle norme per eseguirli viene fatta dai rappresentanti delle stesse aziende che sono in concorrenza nella pubblicizzazione dei risultati: inevitabilmente nascono dei conflitti tra le aspettative delle aziende e quelle degli acquirenti. Presumibilmente in futuro le decisioni che coinvolgono i benchmark potranno essere prese da un gruppo più rappresentativo delle diverse esigenze.

Letture consigliate

Curnow, H. J., e B.A. Wichmann [1976]. "A synthetic benchmark," *The Computer J.* 19 (1):80.
Descrive il primo benchmark sintetico di una certa rilevanza, Whetstone, ed il modo in cui è stato creato.

Flemming, P. J., e J.J Wallace [1986]. "How not to lie with statistics: The correct way to summarize benchmark results," *Comm. ACM* 29:3 (Marzo) 218-21.
Descrive alcuni dei principi base nell'uso di medie diverse per il calcolo delle prestazioni globali.

McMahon, F. M. [1986] "The Livermore FORTRAN kernels: A computer test of numerical performance range," Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore (Dicembre).
Descrive i Loop Livermore, un insieme di kernel benchmark scritti in Fortran.

Smith, J. E. [1988]. "Characterizing computer performance with a single number," *Comm. ACM* 31:10 (Ottobre) 1202-06
Descrive le difficoltà nel sintetizzare le prestazioni in un solo numero e propone il tempo di esecuzione totale come unica misura coerente.

SPEC [1989]. *SPEC Benchmark Suite Release 1.0*, SPEC, Santa Clara, CA, Ottobre.
Descrive la serie di benchmark SPEC. Per informazioni aggiornate, si veda la pagina Web di SPEC attraverso il link all'indirizzo www.mkp.com/books_catalog/cod/links.htm.

Weicker, R. P. [1984]. "Dhrystone: A synthetic systems programming benchmark," *Comm. ACM* 27:10 (Ottobre) 1013-30.
Descrive il benchmark Dhrystone e la sua realizzazione.

Parole chiave

Questo capitolo ha introdotto le nozioni basilari sulla valutazione delle prestazioni, la loro misura e la loro analisi. Sono stati introdotti diversi termini nuovi, indicati di seguito e definiti nel Glossario alla fine del libro.

benchmark SPEC (System Performance Evaluation Cooperative)
carico di lavoro
cicli di clock per istruzione (CPI)
ciclo di clock, periodo di clock, clock , ciclo
combinazione di istruzioni
frequenza di clock
kernel benchmark
legge di Amdahl
media aritmetica
media aritmetica pesata
media armonica dei rapporti
media geometrica
milione di istruzioni per secondo (MIPS)
milione di operazioni in virgola mobile per secondo (MFLOPS)
speedup
tempo di CPU relativo al sistema
tempo di CPU relativo all'utente
tempo di esecuzione della CPU o tempo di CPU
tempo di risposta o tempo di esecuzione

Esercizi

2.1 [5] <§2.1> Si vogliano confrontare le prestazioni di due macchine diverse M1 e M2; su queste macchine sono state eseguite le seguenti misure:

Programma	Tempo su M1	Tempo su M2
1	10 secondi	5 secondi
2	3 secondi	4 secondi

Quale macchina è più veloce per ciascun programma e di quanto?

2.2 [5] <§2.1> Si considerino le due macchine e i due programmi dell'esercizio 2.1. Sulle macchine sono state eseguite le seguenti misure aggiuntive:

Programma	Istruzioni eseguite su M1	Istruzioni eseguite su M2
1	200×10^6	160×10^6

Trovare la frequenza di esecuzione delle istruzioni (numero di istruzioni al secondo) di ciascuna macchina durante l'esecuzione del programma 1.

2.3 [5] <§§2.2-2.3> Se le frequenze di clock delle macchine M1 e M2 dell'esercizio 2.1 sono rispettivamente 200 MHz e 300 MHz, trovare i cicli di clock per istruzione (CPI) necessari per eseguire il programma 1 sulle due macchine, usando i dati degli esercizi 2.1 e 2.2.

2.4 [5] <§§2.2-2.3> {Es. 2.3} Assumendo che il CPI di ciascuna delle macchine dell'Esercizio 2.1 relativo al programma 2 sia uguale al CPI trovato nell'Esercizio 2.3 per il programma 1, calcolare il numero di istruzioni necessarie per eseguire il programma 2 su entrambe le macchine, usando i tempi di esecuzione dell'Esercizio 2.1.

2.5 [5] <§2.1> Si supponga che M1 dell'Esercizio 2.1 costi 10.000 dollari e che M2 ne costi 15.000. Dovendo eseguire il programma 1 un numero elevato di volte (essendo ad esempio interessati al throughput piuttosto che al tempo di risposta), quale tipo di macchina converrebbe comprare, ragionando in termini di grossi quantitativi? Perché?

2.6 [10] <§2.1> Si supponga di considerare altre macchine (ciascuna con un dato costo ed un dato tempo di esecuzione in relazione al programma 1, che deve essere eseguito un gran numero di volte) oltre a M1 e M2 descritte negli Esercizi 2.1 e 2.5. È possibile usare il costo diviso per il tempo di esecuzione come misura di riferimento per decidere quale macchina acquistare? Ed il costo moltiplicato per il tempo di esecuzione? Nel caso nessuna delle due misure abbia senso, illustrare un semplice esempio che ne dimostri la ragione.

2.7 [5] <§2.1> {Es. 2.6} Se si volesse ottenere una misura del rapporto prezzo-prestazioni, simile alle prestazioni nel senso che un numero maggiore indichi un miglior livello di tale rapporto, quale formula bisognerebbe usare?

2.8 [5] <§2.1> Un altro utente è interessato al throughput delle macchine dell'Esercizio 2.1, misurato con un carico di lavoro composto dai programmi 1 e 2 in uguale misura. Quale macchina ha le prestazioni migliori con questo carico di lavoro? Di quanto? Quale macchina ha il miglior rapporto prezzo-prestazioni con questo carico di lavoro? Di quanto?

2.9 [10] <§2.1> Un altro utente ancora ha il seguente requisito riguardo alle macchine di cui si è parlato nell'Esercizio 2.1: il programma 1 deve essere eseguito 200 volte ogni ora; tutto il tempo restante può essere usato per eseguire il programma 2. Se la macchina ha prestazioni sufficienti ad eseguire il programma 1 il numero di volte richiesto ogni ora, le prestazioni possono essere misurate come il throughput del programma 2. Qual è la macchina più veloce con questo carico di lavoro? Quale ha il miglior rapporto prezzo-prestazioni?

2.10 [5] <§§2.2-2.3> Si considerino due diverse implementazioni, M1 ed M2, dello stesso set di istruzioni, cui appartengono quattro diverse classi di istruzioni (A, B, C e D).

M1 ha frequenza di clock pari a 500 MHz; per M1 il numero medio di cicli di clock per ciascuna classe di istruzioni è il seguente:

Classe	CPI per classe
A	1
B	2
C	3
D	4

M2 ha frequenza di clock pari a 750 MHz; per M2 il numero medio di cicli di clock per ciascuna classe di istruzioni è il seguente:

Classe	CPI per classe
A	2
B	2
C	4
D	4

Si assuma che le prestazioni di picco siano definite come la velocità massima con cui viene eseguita una sequenza di istruzioni scelta appositamente. Quali sono le prestazioni massime di M1 e M2 espresse in numero di istruzioni al secondo?

2.11 [5] <§§2.2-2.3> Se il numero di istruzioni di un dato programma è suddiviso in parti uguali fra le classi di istruzioni dell'Esercizio 2.10, di quanto M2 è più veloce di M1?

2.12 [5] <§§2.2-2.3> {Es. 2.11} Assumendo i valori di CPI dell'Esercizio 2.10 e la distribuzione delle istruzioni dell'Esercizio 2.11, per quale frequenza di clock M1 avrebbe le stesse prestazioni della versione di M2 a 750 MHz?

2.13 [10] <§§2.2-2.3> Si considerino due diverse implementazioni, M1 ed M2, dello stesso set di istruzioni, cui appartengono tre diverse classi di istruzioni (A, B e C); M1 ha frequenza di clock pari a 400 MHz, mentre la frequenza di M2 è 200 MHz. Il numero medio di cicli di clock per ciascuna classe di istruzioni su M1 e M2 è riportato nella seguente tabella:

Classe	CPI su M1	CPI su M2	Utilizzo da parte di C1	Utilizzo da parte di C2	Utilizzo da parte di C3
A	4	2	30%	30%	50%
B	6	4	50%	20%	30%
C	8	8	20%	50%	20%

La tabella riporta inoltre la sintesi di come tre diversi compilatori utilizzano il set di istruzioni. C1 è un compilatore prodotto dai costruttori di M1, C2 è rilasciato dai costruttori di M2, mentre C3 è prodotto da un terzo costruttore. Si supponga che ciascun compilatore usi lo stesso numero di istruzioni per convertire un dato programma, ma che la tipologia di istruzioni sia quella descritta in tabella. Usando C1 sia su M1 che su M2, i costruttori di M1 possono vantare la maggior velocità di M1 rispetto a M2: di quanto M1 è più veloce in questo caso? Analogamente, usando C2 sia su M1 che su M2, i costruttori di M2 possono vantare la maggior velocità di M2 rispetto a M1: di quanto M2 è più veloce in questo altro caso? Avendo acquistato M1, quale compilatore è meglio usare? Avendo invece acquistato M2, quale compilatore è la scelta migliore? Quale macchina è il miglior acquisto, supponendo che tutti gli altri criteri siano identici, compreso il costo?

2.14 [5] <§§2.2, 2.3, 2.7> Nel seguente insieme di variabili, identificare tutti i sottoinsiemi che possono essere usati per calcolare il tempo di esecuzione. Ciascun sottoinsieme deve essere minimo, cioè non deve contenere nessuna variabile non necessaria.

{CPI, frequenza di clock, periodo di clock, MIPS, numero di istruzioni di un programma, numero di cicli di clock per eseguire un programma}

2.15 [10] <§§2.2, 2.3, 2.7> Si supponga di essere interessati a due implementazioni di una macchina, una dotata di una speciale unità per il trattamento dei numeri in virgola mobile ed una che non la possiede. Si consideri un programma P con la seguente combinazione di operazioni:

moltiplicazioni in virgola mobile	10%
somme in virgola mobile	15%
divisioni in virgola mobile	5%
operazioni su interi	70%

La macchina MVP (Macchina con Virgola Mobile) contiene l'unità per il trattamento dei numeri in virgola mobile e può quindi eseguire direttamente questo tipo di operazioni. Essa richiede il seguente numero di cicli di clock per ciascuna classe di istruzioni:

moltiplicazioni in virgola mobile	6
somme in virgola mobile	4
divisioni in virgola mobile	20
operazioni su interi	2

La macchina MSVM (Macchina Senza Virgola Mobile) non possiede l'unità per il trattamento dei numeri in virgola mobile e quindi deve emulare tali operazioni tramite operazioni fra interi, che richiedono 2 cicli di clock. Il numero di istruzioni necessarie per emulare le operazioni su numeri in virgola mobile è:

moltiplicazioni in virgola mobile	30
somme in virgola mobile	20
divisioni in virgola mobile	50

Entrambe le macchine hanno frequenza di clock pari a 1000 MHz. Si trovi il valore dei MIPS nativi delle due macchine.

2.16 [10] <§§2.2, 2.3, 2.7> Se la macchina MVM dell'Esercizio 2.15 effettuasse 300 milioni di istruzioni per eseguire il programma, quante operazioni su interi dovrebbe fare la macchinaMSVM per eseguire lo stesso programma?

2.17 [5] <§§2.2, 2.3, 2.7> {Es. 2.16} Assumendo valido il numero di istruzioni dell'Esercizio 2.16, quale sarebbe il tempo di esecuzione del programma dell'Esercizio 2.15 sia su MVM che su MSVM?

2.18 [10] <§§2.2-2.3> Il responsabile del progetto di un nuovo processore, dopo il completamento della fase di progettazione del processore e della prima versione del compilatore, deve decidere se mandare in produzione il progetto così com'è o spendere altro tempo per migliorarlo. Per decidere, discute il problema con il gruppo di specialisti hardware e giunge a definire le seguenti possibilità:

a. *Lasciare il progetto così com'è.* La macchina viene chiamata Mbase ed ha frequenza di clock pari a 500 MHz; utilizzando un simulatore vengono ottenute le misure seguenti:

Classe di istruzioni	CPI	Frequenza
A	2	40%
B	3	25%
C	3	25%
D	5	10%

b. *Ottimizzare l'hardware.* Il gruppo di specialisti sostiene di poter migliorare il progetto del processore in modo che la frequenza di clock raggiunga i 600 MHz. La macchina viene chiamata Mopt; utilizzando un simulatore vengono ottenute le misure seguenti:

Classe di istruzioni	CPI	Frequenza
A	2	40%
B	2	25%
C	3	25%
D	4	10%

Qual è il CPI di ciascuna macchina?

2.19 [5] <§§2.2, 2.3, 2.7> {Es. 2.18} Quali sono i valori dei MIPS nativi per le macchine Mbase e Mopt dell'Esercizio 2.18?

2.20 [10] <§§2.2-2.3> {Es. 2.18} Di quanto Mopt è più veloce di Mbase, usando i dati dell'Esercizio 2.18?

2.21 [5] <§§2.2-2.3> Il gruppo che si occupa del compilatore viene a conoscenza della discussione per migliorare la macchina (vedere Esercizi da 2.18 a 2.20) e propone di migliorare il compilatore per aumentarne ulteriormente le prestazioni. La combinazione macchina base più compilatore avanzato viene chiamata Mcomp. Per quanto riguarda le istruzioni, il miglioramento apportato dal compilatore è stimato nel modo seguente:

Classe (di istruzioni)	Percentuale di istruzioni della classe dopo ottimizzazione
A	90%
B	90%
C	85%
D	95%

Se ad esempio in un programma la macchina base esegue 500 istruzioni di classe A, Mcomp ne esegue $0{,}9 \times 500 = 450$ dello stesso tipo. Quanto vale il CPI di Mcomp?

2.22 [5] <§§2.2-2.3> {Es. 2.18, 2.21} Usando i dati dell'Esercizio 2.18, di quanto Mcomp è più veloce di Mbase?

2.23 [10] <§§2.2-2.3> {Es. 2.18, 2.21, 2.22} Il gruppo che si occupa del compilatore mette in evidenza la possibilità di implementare i miglioramenti hardware suggeriti nell'Esercizio 2.18 e contemporaneamente di utilizzare il compilatore avanzato descritto nell'Esercizio 2.21. Se entrambe le modifiche sono apportate, ottenendo la macchina Mboth, di quanto Mboth è più veloce di Mbase?

2.24 [10] <§§2.2-2.3> {Es. 2.18, 2.21, 2.22, 2.23} Dovendo decidere se introdurre nella macchina base il miglioramento hardware suggerito nell'Esercizio 2.18, il compilatore avanzato dell'Esercizio 2.21 o entrambi, si stima che il tempo richiesto per apportare le ottimizzazioni descritte negli Esercizi 2.18, 2.21 e 2.23 sia:

Ottimizzazione	Tempo richiesto	Macchina
Hardware	6 mesi	Mopt
Compilatore	6 mesi	Mcomp
Entrambe	8 mesi	Mboth

Ricordando dal Capitolo 1 che le prestazioni della CPU crescono approssimativamente del 50% ogni anno, ossia del 3,4% ogni mese, ed assumendo che la macchina base abbia le stesse prestazioni delle macchine concorrenti, quale eventuale ottimizzazione conviene implementare?

2.25 [10] <§§2.4, 2.6> Si esamini la lista dei programmi dell'attuale versione di SPEC, riportata in figura 2.6. Sono presenti le applicazioni che usate tipicamente sul vostro calcolatore? Quali classi di programmi sono irrilevanti oppure mancano? Perché ritenete che siano (o non siano) state incluse in SPEC? Che cosa si dovrebbe fare per includere / escludere tali programmi nella prossima versione di SPEC?

2.26 [5] <§2.5> La tabella seguente mostra il numero di operazioni in virgola mobile eseguite in due programmi diversi ed il tempo di esecuzione di questi programmi su tre macchine differenti:

Programma	Operazioni	Macchina A	Macchina B	Macchina C
Programma 1	10000000	1	10	20
Programma 2	100000000	1000	100	20

Qual è la macchina più veloce, tenendo conto del tempo di esecuzione? Di quanto è più veloce rispetto alle altre due macchine?

2.27 [5] <§§2.5, 2.7> Come potrebbero essere confrontate le prestazioni delle tre macchine dell'Esercizio 2.26, usando altri strumenti per normalizzare le prestazioni? Qual è la macchina più veloce tenendo conto della media geometrica?

2.28 [15] <§§2.5, 2.7> {Es. 2.27} Trovare un carico di lavoro relativo ai due programmi dell'Esercizio 2.26 che produca lo stesso valore complessivo delle prestazioni, usando come tempo di esecuzione totale la media geometrica delle prestazioni come calcolata nell'Esercizio 2.27. Fornire il carico di lavoro come percentuale del numero di esecuzioni di ciascun programma per le coppie di macchine: A e B, B e C, A e C.

2.29 [15] <§§2.5, 2.7> Un utente sostiene che i due programmi dell'Esercizio 2.26 costituiscono la maggior parte del suo carico di lavoro, ma che non vengono eseguiti lo stesso numero di volte. L'utente vuole confrontare le tre macchine con un carico di lavoro composto dai due programmi in proporzioni differenti (si può usare la media aritmetica per calcolare le prestazioni relative).
Si supponga che il numero totale di operazioni in virgola mobile eseguite dal carico di lavoro sia ugualmente diviso fra i due programmi, cioè che il programma 1 sia eseguito 10 volte più frequentemente del programma 2. Quale macchina è più veloce con questo carico di lavoro e di quanto? Qual è il confronto con il tempo totale di esecuzione di un carico di lavoro che comprende un ugual numero di esecuzioni dei due programmi?

2.30 [15] <§§2.5, 2.7> In alternativa all'esercizio 2.29, si assuma che per una data macchina il tempo speso per l'esecuzione di ciascun programma sia lo stesso. Quale macchina è la più veloce, usando i dati dell'Esercizio 2.26 ed assumendo che la macchina per cui i due benchmark sono eseguiti nello stesso tempo sia A? Quale macchina è la più veloce assumendo invece che la macchina per cui i due benchmark sono eseguiti nello stesso tempo sia B? Si confrontino questi risultati con quelli ottenuti senza l'assunzione di partenza.

2.31 [5] <§2.7> Si assuma che le operazioni di moltiplicazione vengano eseguite in 12 cicli di clock e che in un programma tipico costituiscano il 10% delle istruzioni, mentre il restante 90% delle istruzioni richiede in media 4 cicli di clock. Qual è la percentuale di tempo che la CPU utilizza per eseguire le moltiplicazioni?

2.32 [5] <§2.7> {Es. 2.31} Gli specialisti hardware sostengono che è possibile ridurre a 6 il numero di cicli di clock con cui una moltiplicazione viene eseguita, rispetto ai 12 dell'Esercizio 2.31, ma che questo comporta un aumento del 20% del periodo di clock; nessun'altra variabile viene influenzata dal cambiamento. La modifica deve essere effettuata?

2.33 [10] <§§2.1-2.7> Si consideri la seguente notizia ipotetica:

"L'azienda rilascerà la prima versione industriale a 800 MHz del chip, che offre un incremento del 20% rispetto alle prestazioni del precedente prodotto di punta dell'azienda, la versione a 666 MHz. Il nuovo chip può essere inserito nella scheda di sistema al posto della versione originale (a 400 Hz) per ottenere un aumento delle prestazioni del 70%."

Si commenti la definizione (o le definizioni) delle prestazioni che l'azienda sembra utilizzare. La notizia riportata può trarre in inganno?

2.34 [3 ore] <§2.5> Dati due calcolatori A e B, si eseguano il benchmark Dhrystone ed un altro programma in linguaggio C di una certa complessità (ad esempio il compilatore C), chiamando quest'ultimo programma P. Provando ad eseguire i due programmi senza nessuna ottimizzazione e con il livello massimo di ottimizzazioni si calcolino i seguenti rapporti di prestazioni:

a. Dhrystone non ottimizzato sulla macchina A confrontato con Dhrystone non ottimizzato sulla macchina B.

b. P non ottimizzato su A confrontato con P non ottimizzato su B.

c. Dhrystone ottimizzato su A confrontato con Dhrystone ottimizzato su B.

d. P ottimizzato su A confrontato con P ottimizzato su B.

e. Dhrystone non ottimizzato su A confrontato con Dhrystone ottimizzato su A.

f. P non ottimizzato su A confrontato con P ottimizzato su A.

g. Dhrystone non ottimizzato su B confrontato con Dhrystone ottimizzato su B.

h. P non ottimizzato su B confrontato con P ottimizzato su B.

Si vuole capire se Dhrystone predica in modo accurato le prestazioni per altri programmi scritti in linguaggio C. Se Dhrystone fosse in grado di predire le prestazioni, allora le seguenti uguaglianze dovrebbero essere verificate:

(a) = (b) e (c) = (d)

Se Dhrystone fosse in grado di predire in modo accurato l'utilità delle ottimizzazioni del compilatore nel caso di applicazioni reali, allora

(e) = (f) e (g) = (h)

Si determinino quali delle relazioni sopra riportate sono verificate e nelle situazioni in cui non lo sono si cerchi di spiegarne le ragioni. Le caratteristiche delle macchine, le ottimizzazioni del compilatore o la differenza fra P e Dhrysrtone spiegano tutto ciò?

2.35 [3 ore] <§2.5> Si esegua lo stesso esperimento dell'Esercizio 2.34, sostituendo Dhrystone con Whetstone e scegliendo un programma scritto in Fortran che tratti numeri in virgola mobile al posto di P.

2.36 [4 ore] <§§2.4, 2.7> Si costruisca un programma in C o in Pascal che determini il valore dei MIPS di picco di un calcolatore e lo si esegua su due macchine per ricavare tale valore. Si esegua quindi un'applicazione reale, in C o in Pascal (ad esempio un compilatore) sulle due macchine: quanto accuratamente i MIPS di picco predicono le prestazioni dell'applicazione reale?

2.37 [Non definito] <§§2.1-2.7> Si raccolga una serie di articoli che contengano analisi di prestazioni non corrette o che utilizzino metriche fuorvianti nel tentativo

di influenzare il lettore. Ad esempio un articolo del *New York Times* (20 aprile 1994, pag. D1) descriveva un sistema per videogiochi "superiore per capacità di calcolo anche al più potente dei personal computer" e riportava la tabella seguente per sostenere che "i sistemi per videogiochi saranno forse i supercalcolatori di domani":

Macchina	Numero approssimato di istruzioni al secondo	Prezzo
1975 IBM Mainframe	10000000	$10000000
1976 Cray-1	160000000	$20000.00
1979 Digital VAX	1000000	$200000
1981 IBM PC	250000	$3000
1984 Sun 2	1000000	$10000
1994 Pentium-chip PC	66000000	$3000
1995 Sony PCX video game	500000000	$500
1995 Microunity set-top	1000000000	$500

L'articolo non discute mai di come la natura delle istruzioni influenzi la definizione di "capacità di calcolo". Per ciascun articolo raccolto, si descriva il motivo per cui è stato ritenuto non corretto o fuorviante. Materiale di questo tipo può essere facilmente trovato nelle pagine di giornali o riviste dedicate all'economia o alle tecnologie (all'interno di articoli o di inserzioni pubblicitarie) e su Internet (Web e gruppi di discussione).

In maggior dettaglio

I MFLOPS come misura delle prestazioni

Un'alternativa molto diffusa al tempo di esecuzione per misurare le prestazioni è il *milione di operazioni in virgola mobile eseguite in un secondo*, abbreviato in megaFLOPS o MFLOPS (million floating-point operations per seconds), ma sempre pronunciato "megaflops." La formula che definisce i MFLOPS si ricava semplicemente dalla definizione dell'acronimo:

$$\text{MFLOPS} = \frac{\text{numero di operazioni in virgola mobile eseguite in un programma}}{\text{tempo di esecuzione} \times 10^6}$$

Un'*operazione in virgola mobile* è una somma, una sottrazione, una moltiplicazione o una divisione eseguita su un numero rappresentato in virgola mobile in singola o doppia precisione. Dati di questo tipo sono usati in modo massiccio nei calcoli scientifici; nei linguaggi di programmazione sono definiti con parole chiave come *float*, *real*, *double* o *double precision*.

Chiaramente una misurazione in MFLOPS è dipendente dal programma: programmi diversi richiedono l'esecuzione di un numero diverso di operazioni in virgola mobile (come esempio si veda l'Esercizio 2.38). Dal momento che i MFLOPS intendono espressamente misurare le prestazioni riguardanti le operazioni in virgola mobile, non possono essere applicati al di fuori di questo insieme; come caso estremo i compilatori hanno valori in MFLOPS vicini allo zero, dal momento che usano molto raramente l'aritmetica in virgola mobile.

Basandosi sulle istruzioni di un programma piuttosto che sulle istruzioni in generale, i MFLOPS hanno migliori prerogative dei MIPS di confrontare in modo corretto macchine diverse. Chi sostiene ciò lo giustifica dicendo che lo stesso programma eseguito su macchine diverse può richiedere un diverso numero di istruzioni, ma necessita sempre dello stesso numero di operazioni in virgola mobile. Sfortunatamente i MFLOPS non sono attendibili in quanto il set di istruzioni in virgola mobile non è omogeneo in tutte le macchine ed il numero di operazioni in virgola mobile effettivamente svolte può variare. Ad esempio il Cray-2 non ha l'istruzione di divisione, mentre il Motorola 68882 ha la divisione, la radice quadrata, il seno ed il coseno; quindi il Cray-2 necessita di molte operazioni in virgola mobile per eseguire una divisione, mentre nel Motorola 68882 una chiamata alla procedura che calcola il seno di una funzione richiede una sola operazione, quando la stessa chiamata nella maggior parte delle macchine genera parecchie operazioni.

Un altro potenziale problema è relativo al fatto che il valore dei MFLOPS cambia non soltanto in funzione del rapporto fra le operazioni intere e quelle in virgola mobile, ma anche in funzione del rapporto fra operazioni in virgola mobile veloci e operazioni in virgola mobile lente. Ad esempio un programma con una percentuale pari al 100% di somme in virgola mobile avrà un valore di MFLOPS più elevato di un programma che ha il 100% di divisioni. La soluzione per entrambi i problemi consiste nel definire un metodo per contare il numero di operazioni in virgola mobile all'interno di un programma scritto in un linguaggio ad alto livello. Tale conteggio potrà anche pesare in modo diverso le operazioni, dando un peso più elevato alle operazioni più complesse, in modo che una macchina possa raggiungere un buon valore di MFLOPS anche se il programma contiene molte divisioni fra numeri in virgola mobile; questi MFLOPS potrebbero essere chiamati *MFLOPS normalizzati*. Naturalmente, a causa del conteggio e dell'attribuzione dei pesi, i MFLOPS normalizzati possono differire notevolmente dall'effettiva velocità con cui una macchina esegue le operazioni in virgola mobile.

Come qualunque altra misura di prestazioni, il valore in MFLOPS di un singolo programma non può essere generalizzato per stabilire una metrica di valutazione di un calcolatore; l'uso dello stesso termine per far riferimento a concetti diversi, dalle prestazioni di picco (il massimo valore di MFLOPS possibile per un qualsiasi segmento di codice), al valore in MFLOPS di un benchmark, alla misura in MFLOPS normalizzati, non fa altro che aumentare la confusione. La peggiore di queste definizioni di MFLOPS, il MFLOPS di picco, non è correlata alle effettive prestazioni di una macchina; la migliore è ridondante rispetto al tempo di esecuzione, che rappresenta la misura più efficace delle prestazioni. In più, a differenza di quanto avviene nel caso dei tempi di esecuzione, si rischia di cadere nella tentazione di caratterizzare una macchina con il semplice valore in MFLOPS, senza fornire dettagli sul programma utilizzato e sui dati in ingresso.

2.38 [5] <§2.2> Si trovi il valore in MFLOPS dei due programmi su ciascuna delle macchine descritte nell'Esercizio 2.26, assumendo che ogni operazione in virgola mobile contribuisca per 1 FLOP, e si confrontino tutti valori ottenuti. L'esempio illustra uno dei problemi descritti in precedenza?

2.39 [15] <§2.5> Se le prestazioni sono espresse tramite un rapporto, come nel caso dei MFLOPS, allora un valore più elevato ed una media più alta indicano

prestazioni migliori. In questo caso la media che mantiene traccia del tempo di esecuzione è la *media armonica* (MH):

$$\mathrm{MH} = \frac{n}{\sum_{i=1}^{n} \frac{1}{\text{rapporto}_i}}$$

Ciascun rapporto_i vale $1/\text{tempo}_i$, dove tempo_i è il tempo di esecuzione dell'i-esimo degli n programmi del carico di lavoro. Si provi che la media armonica di un insieme di rapporti mantiene traccia del tempo di esecuzione, dimostrando che è l'inverso della media aritmetica dei corrispondenti tempi di esecuzione.

2.40 [4 ore] <§2.4> Si costruisca un programma in C o in Fortran che determini il valore dei MFLOPS di picco di un calcolatore e lo si esegua su due macchine per ricavare tale valore. Si esegua quindi sulle due macchine un'applicazione reale (che utilizzi numeri in virgola mobile): quanto accuratamente i MFLOPS di picco predicono le prestazioni dell'applicazione reale?

In maggior dettaglio

La legge di Amdahl

Talvolta la legge di Amdahl è espressa in modo diverso rispetto alla sua formulazione originale, introducendo il fattore di speedup (incremento delle prestazioni). Lo *speedup* è la misura delle prestazioni di una macchina in seguito ad un miglioramento rispetto alle prestazioni originali; di conseguenza, se una modifica conduce ad uno speedup pari a 2, le prestazioni della macchina raddoppiano rispetto alla situazione iniziale. Si può quindi scrivere

$$\text{speedup} = \frac{\text{prestazioni dopo il miglioramento}}{\text{prestazioni prima del miglioramento}} = \frac{\text{tempo di esecuzione prima del miglioramento}}{\text{tempo di esecuzione dopo il miglioramento}}$$

La versione della legge di Amdahl fornita precedentemente era

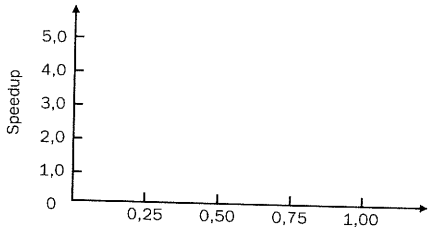
$$\text{tempo di esecuzione dopo il miglioramento} = \left(\frac{\text{tempo di esecuzione influenzato dal miglioramento}}{\text{ammontare del miglioramento}} + \text{tempo di esecuzione non influenzato}\right)$$

2.41 [5] <§2.7> Si supponga di aver migliorato una macchina in modo da far sì che esegua tutte le operazioni in virgola mobile cinque volte più velocemente e si analizzi come si comporta lo speedup quando la modifica viene introdotta. Se prima del miglioramento il tempo di esecuzione di un dato benchmark è di 10 secondi, quale sarà lo speedup nel caso in cui metà dei 10 secondi sono spesi per l'esecuzione delle operazioni in virgola mobile?

2.42 [10] <§2.7> Per mettere in risalto la nuova unità di elaborazione per i numeri in virgola mobile descritta nell'Esercizio 2.41, ci si pone l'obiettivo di tro-

vare un benchmark che mostri uno speedup pari a 3. Uno dei benchmark presi in considerazione viene eseguito in 100 secondi dal vecchio hardware: che peso nell'esecuzione del programma devono avere le operazioni in virgola mobile perché sia possibile mostrare uno speedup complessivo pari a 3?

2.43 [10] <§2.7> Assumendo di migliorare l'unità di elaborazione per i numeri in virgola mobile come descritto nell'Esercizio 2.41, si riportino i dati relativi allo speedup ottenuto in funzione della frazione del tempo di esecuzione impiegata nelle operazioni in virgola mobile; il grafico è del tipo seguente:



Frazione del tempo di esecuzione impiegata nelle operazioni in virgola mobile prima della modifica

2.44 [5] <§2.7> Volendo migliorare le prestazioni di una macchina si individuano due possibili strade: rendere le operazioni di moltiplicazione quattro volte più veloci oppure far sì che le istruzioni di accesso in memoria siano due volte più rapide. Facendo eseguire ripetutamente un programma si ricava che il tempo di esecuzione è di 100 secondi, di cui il 20% è usato per le moltiplicazioni, il 50% per le istruzioni di accesso in memoria ed il restante 30% per altri tipi di operazione. Quale sarà lo speedup se il miglioramento prescelto riguarda soltanto le operazioni di moltiplicazione? Quale se si migliora l'accesso in memoria? Quale se vengono effettuate entrambe le modifiche?

2.45 [5] <§2.7> {Es. 2.44} Volendo modificare il programma descritto nell'Esercizio 2.44 in modo tale che le percentuali dei vari tipi di istruzione non siano più rispettivamente del 20%, del 50% e del 30% ed assumendo che nessuna delle nuove percentuali possa essere nulla, quale tipo di programma fa sì che lo speedup ottenuto con le due modifiche sia lo stesso? Si riportino sia la formula che alcuni esempi significativi.

2.46 [20] <§2.7> La legge di Amdahl che determina lo speedup complessivo è spesso espressa in funzione di due variabili: l'entità del miglioramento e la frazione del tempo di esecuzione originale influenzata dal miglioramento. Si derivi tale formula a partire dalle due riportate in precedenza.

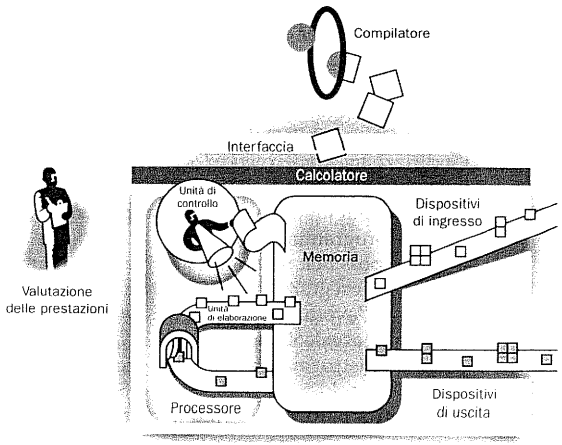
3

Le istruzioni: il linguaggio dei calcolatori

Parlo spagnolo con Dio, italiano con le donne, francese con gli uomini e tedesco col mio cavallo.

Carlo V, re di Francia
1337-1380

Le cinque componenti classiche del calcolatore



3.1 Introduzione

Per poter gestire l'hardware di un calcolatore bisogna saper parlare il suo linguaggio; le parole del linguaggio del calcolatore si chiamano *istruzioni* ed il vocabolario è il *set di istruzioni*. In questo capitolo verrà descritto il set di istruzioni di un calcolatore reale, sia nella forma scritta dall'uomo che in quella letta dall'elaboratore: a partire da una notazione simile ad un linguaggio di programmazione evoluto, seppur con molte limitazioni, si arriva passo dopo passo al vero linguaggio di un calcolatore reale.

Si potrebbe pensare che i linguaggi dei calcolatori siano molti e variegati come quelli umani. In realtà sono abbastanza simili fra loro; si può paragonarli più a dialetti regionali derivati da un'unica radice che non a lingue diverse: di conseguenza avendo imparato un linguaggio è molto semplice apprenderne altri. Le somiglianze fra i linguaggi sono dovute al fatto che tutti i calcolatori sono costruiti secondo tecnologie hardware che si basano sugli stessi principi fondamentali e al fatto che esistono alcune operazioni base che tutti i calcolatori devono poter eseguire. In più i progettisti di calcolatori hanno un obiettivo comune: trovare un linguaggio che renda semplice implementare l'hardware e costruire il compilatore, massimizzando le prestazioni e minimizzando i costi. Questo obiettivo è ben assestato nel tempo: la citazione seguente fu scritta ben prima che fosse possibile acquistare un calcolatore ed è vera oggi tanto quanto nel 1947.

> *Mediante i metodi della logica formale è facile verificare che esiste un certo numero di [set di istruzioni] che, in astratto, si prestano a controllare l'hardware per eseguire qualsiasi sequenza di operazioni... Le considerazioni davvero decisive nella scelta di un set di istruzioni sono di natura molto più pratica: la semplicità dei dispositivi richiesti dal [set di istruzioni], la chiarezza della sua applicazione nel caso di problemi realmente importanti e la velocità con cui questi problemi sono gestiti.*
>
> Burks, Goldstine e von Neumann, 1947

La "semplicità dei dispositivi" è un parametro di grande valore per i calcolatori del 2000 come per quelli degli anni '50: l'obiettivo del presente capitolo è insegnare un set di istruzioni che segue queste indicazioni, mostrandone la realizzazione hardware e la relazione fra i linguaggi ad alto livello e questo linguaggio più primitivo. Nel seguito sarà utilizzato il linguaggio C; nel caso il lettore abbia maggiore familiarità con il Pascal, è invitato a consultare l'estensione Web III, disponibile all'indirizzo *www.mkp.com/cod2e.htm*, in cui viene fatto un breve confronto fra il C ed il Pascal.

Imparare il modo in cui sono rappresentate le istruzioni aiuta a scoprire il segreto della programmazione, cioè il concetto della memorizzazione dei programmi. Inoltre è possibile fare pratica scrivendo programmi nel linguaggio del calcolatore e facendoli eseguire dal simulatore allegato al libro. Il capitolo si conclude con uno sguardo all'evoluzione storica dei set delle istruzioni e con un accenno ad altri dialetti usati dai calcolatori.

Il set di istruzioni prescelto proviene dal processore MIPS, usato fra gli altri da NEC, Nintendo, Silicon Graphics e Sony; è un tipico esempio dei set di istruzioni progettati a partire dai primi anni '80. Il set di istruzioni MIPS sarà presentato in

modo graduale, spiegandone ogni volta i fondamenti logici e le relative strutture del calcolatore; questo approccio rende il linguaggio assembler più facilmente comprensibile. Per mantenere un quadro generale di riferimento, al termine di ciascun paragrafo viene riportata una figura che riassume le istruzioni MIPS introdotte fino a quel momento, ponendo in evidenza quelle presentate nel paragrafo corrente.

3.2 Le operazioni svolte dall'hardware del calcolatore

> *Devono certamente essere previste istruzioni per il calcolo delle operazioni aritmetiche fondamentali.*
>
> Burks, Goldstine e von Neumann, 1947

Qualsiasi calcolatore deve saper eseguire le operazioni aritmetiche. La notazione del linguaggio assembler MIPS

```
add   a, b, c
```

indica al calcolatore di sommare le due variabili `b` e `c` e di memorizzare la loro somma in `a`.

Questa notazione è rigida, nel senso che ciascuna istruzione aritmetica MIPS esegue una sola operazione e contiene esattamente tre variabili. Si supponga ad esempio di voler memorizzare in `a` la somma delle variabili `b`, `c`, `d` ed `e` (in questo paragrafo il termine "variabile" sarà lasciato deliberatamente vago, rimandando al paragrafo successivo una trattazione più dettagliata e precisa); la seguente sequenza di istruzioni somma le tre variabili:

```
add a, b, c   # la somma di b e c è posta in a
add a, a, d   # ora in a è presente la somma di b, c e d
add a, a, e   # ora in a è presente la somma di b, c, d ed e
```

Occorrono quindi tre istruzioni per sommare quattro variabili.

Su ciascuna linea di codice le parole a destra del simbolo # rappresentano i *commenti* rivolti a chi legge il programma e sono ignorate dal calcolatore. Si noti che, a differenza di altri linguaggi di programmazione, in questo linguaggio ciascuna linea può contenere al massimo un'istruzione; un'altra differenza è che i commenti terminano sempre con la fine della linea.

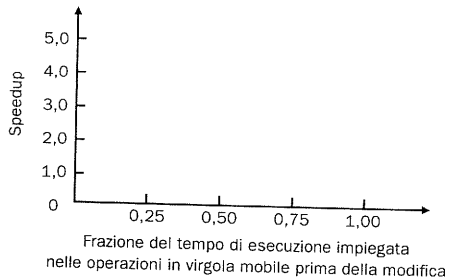
Il numero di operandi più naturale per un'operazione come la somma è pari a tre: i due numeri da sommare ed il riferimento alla locazione in cui memorizzare il risultato; il fatto di richiedere che tutte le istruzioni abbiano esattamente tre operandi, non uno di più né uno di meno, è conforme alla filosofia di mantenere l'hardware semplice: l'hardware necessario nel caso di un numero variabile di operandi è più complesso di quello richiesto per un numero fisso di operandi. Questa situazione illustra il primo di quattro principi fondamentali per il progetto dell'hardware:

Principio di progetto numero 1: semplicità e regolarità sono strettamente correlate.

Nei due esempi che seguono verrà mostrata la relazione fra programmi scritti in linguaggi ad alto livello e programmi scritti in questa notazione più semplice; la figura 3.1 riassume le parti di linguaggio assembler MIPS finora descritte.

vare un benchmark che mostri uno speedup pari a 3. Uno dei benchmark presi in considerazione viene eseguito in 100 secondi dal vecchio hardware: che peso nell'esecuzione del programma devono avere le operazioni in virgola mobile perché sia possibile mostrare uno speedup complessivo pari a 3?

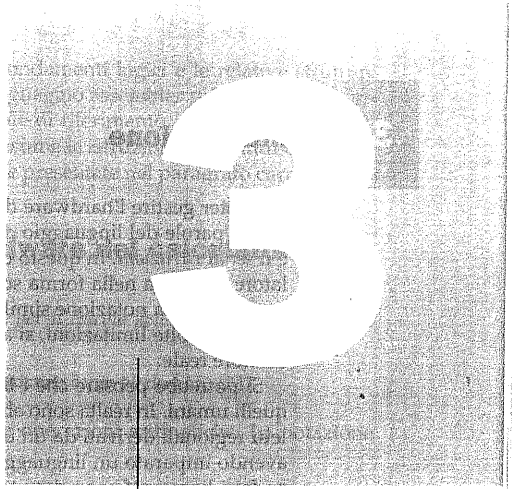
2.43 [10] <§2.7> Assumendo di migliorare l'unità di elaborazione per i numeri in virgola mobile come descritto nell'Esercizio 2.41, si riportino i dati relativi allo speedup ottenuto in funzione della frazione del tempo di esecuzione impiegata nelle operazioni in virgola mobile; il grafico è del tipo seguente:



2.44 [5] <§2.7> Volendo migliorare le prestazioni di una macchina si individuano due possibili strade: rendere le operazioni di moltiplicazione quattro volte più veloci oppure far sì che le istruzioni di accesso in memoria siano due volte più rapide. Facendo eseguire ripetutamente un programma si ricava che il tempo di esecuzione è di 100 secondi, di cui il 20% è usato per le moltiplicazioni, il 50% per le istruzioni di accesso in memoria ed il restante 30% per altri tipi di operazione. Quale sarà lo speedup se il miglioramento prescelto riguarda soltanto le operazioni di moltiplicazione? Quale se si migliora l'accesso in memoria? Quale se vengono effettuate entrambe le modifiche?

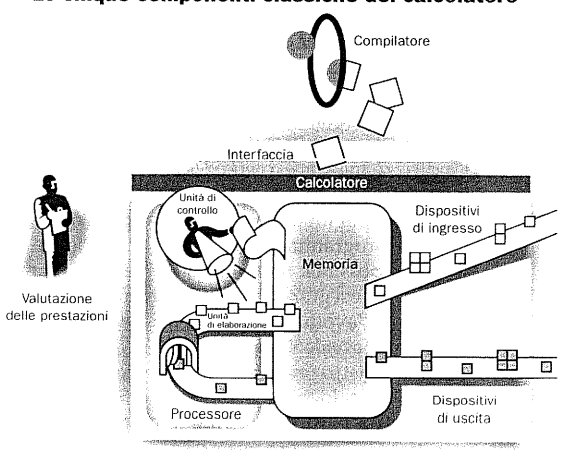
2.45 [5] <§2.7> {Es. 2.44} Volendo modificare il programma descritto nell'Esercizio 2.44 in modo tale che le percentuali dei vari tipi di istruzione non siano più rispettivamente del 20%, del 50% e del 30% ed assumendo che nessuna delle nuove percentuali possa essere nulla, quale tipo di programma fa sì che lo speedup ottenuto con le due modifiche sia lo stesso? Si riportino sia la formula che alcuni esempi significativi.

2.46 [20] <§2.7> La legge di Amdahl che determina lo speedup complessivo è spesso espressa in funzione di due variabili: l'entità del miglioramento e la frazione del tempo di esecuzione originale influenzata dal miglioramento. Si derivi tale formula a partire dalle due riportate in precedenza.



Le istruzioni: il linguaggio dei calcolatori

Le cinque componenti classiche del calcolatore



Parlo spagnolo con Dio,
italiano con le donne,
francese con gli uomini
e tedesco col mio cavallo.

Carlo V, re di Francia
1337-1380

3.1 Introduzione

Per poter gestire l'hardware di un calcolatore bisogna saper parlare il suo linguaggio; le parole del linguaggio del calcolatore si chiamano *istruzioni* ed il vocabolario è il *set di istruzioni*. In questo capitolo verrà descritto il set di istruzioni di un calcolatore reale, sia nella forma scritta dall'uomo che in quella letta dall'elaboratore: a partire da una notazione simile ad un linguaggio di programmazione evoluto, seppur con molte limitazioni, si arriva passo dopo passo al vero linguaggio di un calcolatore reale.

Si potrebbe pensare che i linguaggi dei calcolatori siano molti e variegati come quelli umani. In realtà sono abbastanza simili fra loro; si può paragonarli più a dialetti regionali derivati da un'unica radice che non a lingue diverse: di conseguenza avendo imparato un linguaggio è molto semplice apprenderne altri. Le somiglianze fra i linguaggi sono dovute al fatto che tutti i calcolatori sono costruiti secondo tecnologie hardware che si basano sugli stessi principi fondamentali e al fatto che esistono alcune operazioni base che tutti i calcolatori devono poter eseguire. In più i progettisti di calcolatori hanno un obiettivo comune: trovare un linguaggio che renda semplice implementare l'hardware e costruire il compilatore, massimizzando le prestazioni e minimizzando i costi. Questo obiettivo è ben assestato nel tempo: la citazione seguente fu scritta ben prima che fosse possibile acquistare un calcolatore ed è vera oggi tanto quanto nel 1947.

> *Mediante i metodi della logica formale è facile verificare che esiste un certo numero di [set di istruzioni] che, in astratto, si prestano a controllare l'hardware per eseguire qualsiasi sequenza di operazioni... Le considerazioni davvero decisive nella scelta di un set di istruzioni sono di natura molto più pratica: la semplicità dei dispositivi richiesti dal [set di istruzioni], la chiarezza della sua applicazione nel caso di problemi realmente importanti e la velocità con cui questi problemi sono gestiti.*
>
> Burks, Goldstine e von Neumann, 1947

La "semplicità dei dispositivi" è un parametro di grande valore per i calcolatori del 2000 come per quelli degli anni '50: l'obiettivo del presente capitolo è insegnare un set di istruzioni che segue queste indicazioni, mostrandone la realizzazione hardware e la relazione fra i linguaggi ad alto livello e questo linguaggio più primitivo. Nel seguito sarà utilizzato il linguaggio C; nel caso il lettore abbia maggiore familiarità con il Pascal, è invitato a consultare l'estensione Web III, disponibile all'indirizzo *www.mkp.com/cod2e.htm*, in cui viene fatto un breve confronto fra il C ed il Pascal.

Imparare il modo in cui sono rappresentate le istruzioni aiuta a scoprire il segreto della programmazione, cioè il concetto della memorizzazione dei programmi. Inoltre è possibile fare pratica scrivendo programmi nel linguaggio del calcolatore e facendoli eseguire dal simulatore allegato al libro. Il capitolo si conclude con uno sguardo all'evoluzione storica dei set delle istruzioni e con un accenno ad altri dialetti usati dai calcolatori.

Il set di istruzioni prescelto proviene dal processore MIPS, usato fra gli altri da NEC, Nintendo, Silicon Graphics e Sony; è un tipico esempio dei set di istruzioni progettati a partire dai primi anni '80. Il set di istruzioni MIPS sarà presentato in

modo graduale, spiegandone ogni volta i fondamenti logici e le relative strutture del calcolatore; questo approccio rende il linguaggio assembler più facilmente comprensibile. Per mantenere un quadro generale di riferimento, al termine di ciascun paragrafo viene riportata una figura che riassume le istruzioni MIPS introdotte fino a quel momento, ponendo in evidenza quelle presentate nel paragrafo corrente.

3.2 Le operazioni svolte dall'hardware del calcolatore

> *Devono certamente essere previste istruzioni per il calcolo delle operazioni aritmetiche fondamentali.*
>
> Burks, Goldstine e von Neumann, 1947

Qualsiasi calcolatore deve saper eseguire le operazioni aritmetiche. La notazione del linguaggio assembler MIPS

```
add   a, b, c
```

indica al calcolatore di sommare le due variabili `b` e `c` e di memorizzare la loro somma in `a`.

Questa notazione è rigida, nel senso che ciascuna istruzione aritmetica MIPS esegue una sola operazione e contiene esattamente tre variabili. Si supponga ad esempio di voler memorizzare in `a` la somma delle variabili `b`, `c`, `d` ed `e` (in questo paragrafo il termine "variabile" sarà lasciato deliberatamente vago, rimandando al paragrafo successivo una trattazione più dettagliata e precisa); la seguente sequenza di istruzioni somma le tre variabili:

```
add a, b, c   # la somma di b e c è posta in a
add a, a, d   # ora in a è presente la somma di b, c e d
add a, a, e   # ora in a è presente la somma di b, c, d ed e
```

Occorrono quindi tre istruzioni per sommare quattro variabili.

Su ciascuna linea di codice le parole a destra del simbolo # rappresentano i *commenti* rivolti a chi legge il programma e sono ignorate dal calcolatore. Si noti che, a differenza di altri linguaggi di programmazione, in questo linguaggio ciascuna linea può contenere al massimo un'istruzione; un'altra differenza è che i commenti terminano sempre con la fine della linea.

Il numero di operandi più naturale per un'operazione come la somma è pari a tre: i due numeri da sommare ed il riferimento alla locazione in cui memorizzare il risultato; il fatto di richiedere che tutte le istruzioni abbiano esattamente tre operandi, non uno di più né uno di meno, è conforme alla filosofia di mantenere l'hardware semplice: l'hardware necessario nel caso di un numero variabile di operandi è più complesso di quello richiesto per un numero fisso di operandi. Questa situazione illustra il primo di quattro principi fondamentali per il progetto dell'hardware:

Principio di progetto numero 1: semplicità e regolarità sono strettamente correlate.

Nei due esempi che seguono verrà mostrata la relazione fra programmi scritti in linguaggi ad alto livello e programmi scritti in questa notazione più semplice; la figura 3.1 riassume le parti di linguaggio assembler MIPS finora descritte.

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	add a, b, c	a = b + c	Sempre tre operandi
	sottrazione	sub a, b, c	a = b - c	Sempre tre operandi

Figura 3.1 L'architettura MIPS mostrata nel Paragrafo 3.2. Gli operandi effettivi saranno descritti nel paragrafo seguente. In queste tabelle le parti evidenziate mostrano le istruzioni MIPS introdotte nel paragrafo corrente; in questa prima figura è tutto nuovo.

Compilare in assembler MIPS due assegnazioni scritte in C

Esempio

Questo segmento di programma C contiene le cinque variabili a, b, c, d ed e:

```
a = b + c;
d = a - e;
```

La traduzione da C a linguaggio assembler MIPS è svolta dal *compilatore*: si mostri il codice MIPS prodotto da un compilatore C.

Soluzione

Un'istruzione MIPS lavora su due operandi sorgente e pone il risultato in un operando destinazione; quindi le due semplici istruzioni C vengono compilate direttamente in queste due istruzioni assembler MIPS:

```
add  a, b, c
sub  d, a, e
```

Compilare in assembler MIPS un'assegnazione complessa scritta in C

Esempio

Un'istruzione C relativamente complessa contiene le cinque variabili f, g, h, i e j:

```
f = (g + h) - (i + j);
```

Che cosa produce in uscita un compilatore C?

Soluzione

Il compilatore deve suddividere questa istruzione C in diverse istruzioni assembler, dal momento che ciascuna istruzione MIPS può eseguire una sola operazione. La prima istruzione MIPS calcola la somma di g ed h; il risultato deve essere memorizzato da qualche parte e quindi il compilatore crea una variabile temporanea, chiamata t0:

```
add t0, g, h  # la variabile temporanea t0 contiene g + h
```

Anche se l'operazione successiva nell'istruzione C è una sottrazione, prima di eseguirla è necessario calcolare la somma di i e j: la seconda istruzione assembler memorizza tale somma in un'altra variabile temporanea creata dal compilatore, chiamata t1:

```
add t1, i, j  # la variabile temporanea t1 contiene i + j
```


Infine l'istruzione di sottrazione sottrae la seconda somma dalla prima e memorizza il risultato nella variabile f, completando il codice compilato:

```
sub f, t0, t1  # f assume il valore t0 - t1, cioè (g + h) - (i + j)
```

Queste istruzioni sono rappresentazioni simboliche di ciò che il MIPS è in grado di comprendere effettivamente; tra breve questa rappresentazione simbolica si trasformerà gradualmente nel vero linguaggio assembler MIPS.

3.3 Gli operandi dell'hardware del calcolatore

A differenza dei programmi scritti in linguaggi ad alto livello, gli operandi delle istruzioni aritmetiche non possono essere variabili qualsiasi: devono essere scelti fra un numero limitato di locazioni particolari, chiamate *registri*. I registri sono i mattoni che costituiscono il calcolatore, dal momento che rappresentano sia le primitive usate nella progettazione dell'hardware che gli elementi visibili al programmatore. La dimensione dei registri nell'architettura MIPS è di 32 bit; gruppi di 32 bit sono così frequenti che prendono il nome di *parola* (word).

La differenza più evidente fra le variabili dei linguaggi di programmazione ed i registri è il numero limitato di questi ultimi, tipicamente 32 nei calcolatori attuali; il MIPS ha 32 registri (si veda il Paragrafo 3.15 per l'evoluzione storica del numero dei registri). In questo paragrafo, avanzando di un altro passo verso la rappresentazione simbolica del linguaggio MIPS, viene introdotto il vincolo che i tre operandi delle istruzioni aritmetiche MIPS siano scelti fra i 32 registri a 32 bit.

La ragione per la limitazione ai 32 registri può essere trovata nel secondo dei quattro principi per il progetto dell'hardware:

Principio di progetto numero 2: minori sono le dimensioni, maggiore è la velocità.

Un numero di registri molto elevato aumenterebbe la durata del ciclo di clock, semplicemente perché i segnali elettrici impiegano un tempo maggiore se devono coprire una distanza maggiore.

I principi guida come "minori sono le dimensioni, maggiore è la velocità" non sono validi in modo assoluto: 31 registri possono anche non essere meglio di 32; i progettisti tengono molto in conto osservazioni di questo tipo. Nel caso specifico, il progettista deve trovare un compromesso fra la possibilità di utilizzare molti registri all'interno dei programmi e la necessità di rendere veloce il ciclo di clock.

I Capitoli 5 e 6 mostrano il ruolo centrale dei registri nella costruzione dell'hardware; in questo capitolo si vedrà invece come un uso efficiente dei registri sia la chiave per ottenere buone prestazioni da un programma.

Pur essendo possibile in linea di principio specificare un registro con un numero compreso fra 0 e 31, la convenzione MIPS utilizza a questo scopo un nome costituito da due caratteri preceduti dal segno di dollaro. Il Paragrafo 3.6 spiegherà la ragione di questa scelta; per il momento verranno usati $s0, $s1, ... per i registri che corrispondono a variabili nei programmi C da compilare e $t0, $t1, ... per i registri temporanei necessari alla compilazione dei programmi in istruzioni MIPS.

Compilare un'assegnazione C usando i registri

Associare i registri alle variabili del programma è compito del compilatore. Si prenda come esempio l'assegnazione C esaminata precedentemente:

```
f = (g + h) - (i + j);
```

Le variabili f, g, h, i e j possono essere fatte corrispondere rispettivamente ai registri $s0, $s1, $s2, $s3 e $s4. Qual è il codice MIPS che si ottiene dalla compilazione?

Il programma compilato è molto simile al caso precedente, ad eccezione del fatto che le variabili sono sostituite dai registri precedentemente definiti e da altri due registri temporanei, $t0 e $t1, che corrispondono alle variabili temporanee:

```
add $t0, $s1, $s2 # il registro $t0 contiene g + h
add $t1, $s3, $s4 # il registro $t1 contiene i + j
sub $s0, $t0, $t1 # f assume il valore $t0 - $t1, cioè
                  # (g + h) - (i + j)
```

I linguaggi di programmazione utilizzano tipi di variabili semplici, che come in questo caso contengono un solo dato, ma anche strutture dati più complesse, come i vettori. Queste strutture possono contenere un numero di elementi maggiore del numero dei registri all'interno dell'elaboratore: come può un calcolatore rappresentare e consentire l'accesso a strutture dati così grandi?

Le cinque componenti del calcolatore introdotte nel Capitolo 1 e richiamate all'inizio di questo capitolo suggeriscono che benché il processore possa contenere solo un limitato numero di dati nei registri, la memoria ne può contenere milioni. Di conseguenza le strutture dati complesse, come i vettori, sono allocate in memoria.

Come è stato detto precedentemente, nel linguaggio assembler MIPS le istruzioni aritmetiche richiedono che gli operandi siano memorizzati nei registri; il linguaggio deve quindi comprendere istruzioni che trasferiscano i dati fra memoria e registri: tali istruzioni sono dette istruzioni di *trasferimento dati*. Per accedere ad una parola in memoria l'istruzione deve fornire l'*indirizzo* corrispondente; la memoria è semplicemente un grande vettore unidimensionale, con gli indirizzi che rappresentano l'indice del vettore a partire da 0. Ad esempio nella figura 3.2 l'indirizzo del terzo dato è 2 ed il valore di Memoria[2] è 10.

L'istruzione di trasferimento che sposta un dato dalla memoria ad un registro si chiama tradizionalmente *load* (carica). Il formato dell'istruzione load consiste nel suo nome seguito dal registro in cui deve essere trasferito il dato, da una costante e da un registro usato per accedere alla memoria; l'indirizzo della memoria è formato dalla somma della costante e del contenuto del secondo registro. Il nome convenzionale dell'istruzione load in linguaggio MIPS è lw, che significa *load word* (carica una parola).

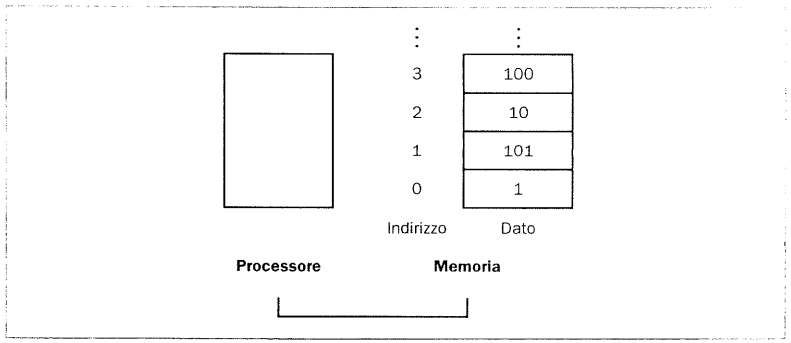


Figura 3.2 Indirizzi di memoria e contenuto delle locazioni corrispondenti. Questa è una semplificazione dell'effettiva modalità di indirizzamento del MIPS; la figura 3.3 mostrerà la modalità di indirizzamento del MIPS per parole consecutive.

Compilare un'assegnazione quando uno degli operandi è in memoria

Si assuma che A sia un vettore di 100 parole e che il compilatore associ le variabili g ed h ai registri $s1 e $s2, come nel caso precedente; si assuma inoltre che l'indirizzo di partenza del vettore, o *indirizzo di base*, sia memorizzato in $s3. Tradurre la seguente assegnazione in linguaggio C:

```
g = h + A[8];
```

Benché l'assegnazione in C svolga una singola operazione, uno degli operandi si trova in memoria e quindi per prima cosa si deve trasferire A[8] in un registro. L'indirizzo di questo elemento del vettore è dato dalla somma dell'indirizzo di base del vettore A, che si trova nel registro $s3 e del numero che permette di scegliere l'elemento 8. Il dato corrispondente deve essere memorizzato in un registro temporaneo per essere utilizzato nell'istruzione successiva; la prima istruzione compilata è quindi

```
lw   $t0, 8($s3)     # il registro temporaneo $t0 assume
                     # il valore A[8]
```

L'istruzione successiva può utilizzare il valore memorizzato in $t0 (che è uguale a A[8]), dal momento che è in un registro; l'istruzione deve sommare h ($s2) ad A[8] ($t0) e porre il risultato nel registro corrispondente a g ($s1):

```
add  $s1, $s2, $t0   # g = h + A[8]
```

La costante in un'istruzione di trasferimento dati si chiama *offset* (spiazzamento) ed il registro usato per ottenere l'indirizzo è detto *registro di base*.

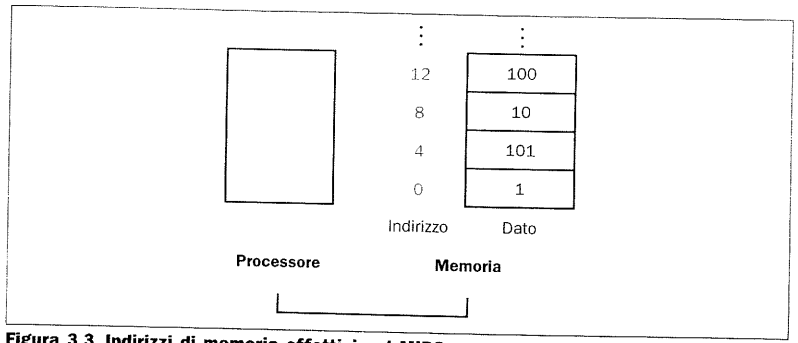


Figura 3.3 Indirizzi di memoria effettivi nel MIPS e contenuto delle locazioni corrispondenti. Gli indirizzi evidenziati sono quelli che cambiano rispetto alla figura 3.2. Dato che il MIPS indirizza il singolo byte, gli indirizzi delle parole sono multipli di 4 (ci sono 4 byte in una parola).

Interfaccia Hardware Software

Oltre ad associare le variabili ai registri, il compilatore alloca strutture dati come i vettori all'interno di locazioni di memoria, ponendo l'indirizzo di partenza corretto nell'istruzione di trasferimento dati.

Dal momento che i *byte*, composti da 8 bit, sono utilizzati in molti programmi, la maggior parte delle architetture indirizza il singolo byte, quindi l'indirizzo di una parola corrisponde all'indirizzo di uno dei 4 byte che la compongono. L'indirizzo di parole consecutive differisce quindi di 4 unità; la figura 3.3, ad esempio, mostra gli effettivi indirizzi MIPS per la figura 3.2: l'indirizzo in byte della terza parola è 8.

Nel MIPS le parole iniziano sempre ad indirizzi multipli di 4; questo requisito si chiama *vincolo di allineamento* ed è comune a molte altre architetture (il Capitolo 5 spiega perché questa restrizione rende il trasferimento dei dati più veloce).

I calcolatori con indirizzamento relativo al singolo byte si dividono fra quelli che utilizzano il byte più a sinistra (o "big end" byte) per specificare l'indirizzo della parola e quelli che utilizzano il byte più a destra (o "little end" byte); il MIPS fa parte della prima categoria. Il Paragrafo A.9 dell'Appendice A mostra le due opzioni per numerare i byte di una parola.

L'indirizzamento a byte influenza anche gli indici dei vettori: per ottenere l'indirizzo corretto in byte nel codice precedente, l'offset da sommare al registro di base `$s3` deve essere 4 × 8, cioè 32, in modo tale che l'istruzione load selezioni `A[8]` e non `A[8/4]`.

L'istruzione complementare a quella di load è chiamata tradizionalmente *store* (memorizza) e trasferisce un dato da un registro alla memoria. Il formato di questa istruzione è analogo a quello della load: il nome dell'istruzione seguito dal registro che contiene il dato da trasferire in memoria, dall'offset corrispondente all'elemento del vettore e dal registro di base. Come nel caso precedente, l'indirizzo MIPS è specificato da una costante e dal contenuto di un registro. Il nome convenzionale dell'istruzione store in linguaggio MIPS è `sw`, che significa *store word* (memorizza una parola).

Esempio

Compilare usando load e store

Si assuma che la variabile `h` sia associata al registro `$s2` e che il registro di base del vettore `A` sia `$s3`. Qual è il codice in assembler MIPS relativo alla seguente assegnazione in linguaggio C?

```
A[12] = h + A[8];
```

Soluzione

Benché l'assegnazione in C svolga una sola operazione, adesso gli operandi che si trovano in memoria sono due e quindi occorrono ancora più istruzioni MIPS del caso precedente. Le prime due istruzioni sono le stesse dell'esempio di prima, a parte l'utilizzo dell'offset corretto nell'indirizzamento dell'istruzione di load che seleziona `A[8]` e la memorizzazione della somma in `$t0`:

```
lw   $t0, 32($s3)  # il registro temporaneo $t0 assume
                   # il valore A[8]
add  $t0, $s2, $t0 # il registro temporaneo $t0 assume
                   # il valore h + A[8]
```

L'ultima istruzione memorizza la somma in `A[12]`, usando 48 come offset e `$s3` come registro di base.

```
sw   $t0, 48($s3)  # memorizza h + A[8] in A[12]
```

Spesso l'indice di un elemento di un vettore non viene specificato con una costante ma con una variabile, in modo tale che l'elemento possa variare durante l'esecuzione di un programma.

Esempio

Compilare usando un indice variabile all'interno di un vettore

Il seguente codice è un esempio di accesso ad un vettore tramite un indice variabile:

```
g = h + A[i];
```

Si assuma che `A` sia un vettore di 100 elementi il cui indirizzo di base si trova nel registro `$s3` e che il compilatore associ alle variabili `g`, `h` e `i` i registri `$s1`, `$s2` e `$s4`. Qual è il codice in assembler MIPS corrispondente all'istruzione C?

Esempio

Prima di memorizzare `A[i]` in un registro temporaneo è necessario conoscere il suo indirizzo, che non si ottiene sommando direttamente `i` all'indirizzo di base del vettore, ma sommando `i` × 4 (si ricordi che l'indirizzamento è sempre relativo al byte). L'istruzione di moltiplicazione sarà introdotta nel prossimo capitolo e per il momento l'effetto di moltiplicare `i` per 4 sarà ottenuto sommando `i` a se stesso (`i + i = 2i`) e poi sommando la somma a se stessa (`2i + 2i = 4i`):

```
add  $t1, $s4, $s4   # $t1 = 2 * i
add  $t1, $t1, $t1   # $t1 = 4 * i
```

Per ottenere l'indirizzo di `A[i]` si somma `$t1` all'indirizzo di base di `A` contenuto in `$s3`:

```
add  $t1, $t1, $s3    # $t1 = indirizzo di A[i]
                      # (4 * i + $s3)
```

Usando questo indirizzo si può caricare `A[i]` in un registro temporaneo:

```
lw   $t0, 0($t1)      # $t0 = A[i]
```

L'ultima istruzione somma `A[i]` ad `h` e pone la somma in `g`:

```
add  $s1, $s2, $t0    # g = h + A[i]
```

Interfaccia Hardware Software

Molti programmi contengono più variabili dei registri del calcolatore. Di conseguenza il compilatore cerca di mantenere nei registri le variabili usate più frequentemente e pone le restanti in memoria, utilizzando istruzioni di trasferimento dati per spostare le variabili fra registri e memoria. Il processo di allocare in memoria le variabili usate meno frequentemente o usate in fasi successive del programma, si chiama riversamento (*spilling*) dei registri.

Il già citato principio hardware che lega le dimensioni alla velocità suggerisce che l'accesso alla memoria sia più lento dell'accesso ai registri, dal momento che i registri hanno dimensione minore. Gli accessi ai dati sono pertanto più veloci se i dati si trovano nei registri invece che in memoria.

Inoltre i dati risultano più facilmente trattabili se si trovano nei registri: le istruzioni aritmetiche del MIPS possono in un colpo solo leggere due registri, eseguire l'operazione e scrivere il risultato. Le istruzioni MIPS per il trasferimento dei dati possono solo leggere o scrivere un operando, senza eseguire nessuna elaborazione su di esso.

I registri del MIPS quindi hanno *sia* un tempo di accesso minore *che* un miglior throughput rispetto alla memoria (una combinazione rara), rendendo i dati in esso contenuti semplici da usare e accessibili velocemente. Per ottenere le prestazioni migliori i compilatori MIPS devono utilizzare i registri in modo efficiente.

La figura 3.4 riassume le parti relative alla rappresentazione simbolica dell'assembler MIPS introdotte in questo paragrafo. Load word e store word sono le istruzioni che consentono di trasferire i dati fra la memoria ed i registri nell'architettura MIPS; altri calcolatori usano più istruzioni, oltre a load e store, per il trasferimento dei dati; un'architettura alternativa è quella dell'Intel 80x86, descritta nel Paragrafo 3.12.

Approfondimento: Il sistema di indirizzamento tramite offset e registro di base è ottimo per accedere alle strutture dati, dal momento che il registro può puntare all'inizio della struttura e l'offset può selezionare l'elemento desiderato; un esempio di questo tipo apparirà nel Paragrafo 3.10.

Nelle istruzioni per il trasferimento dei dati, i registri vennero originariamente introdotti per memorizzare l'indice dell'elemento del vettore, mentre l'offset era usato come indirizzo di partenza del vettore stesso; di conseguenza il registro di base è anche chiamato *registro*

Operandi MIPS

Nome	Esempio	Commenti
32 registri	$t0, $s7, ...	Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche.
2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte, quindi le parole consecutive differiscono di 4 unità. La memoria contiene le strutture dati come i vettori ed i registri riversati (spilled).

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	add $s1, $s2, $s3	$s1 = $s2 + $s3	Tre operandi; dati nei registri
	sottrazione	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Tre operandi; dati nei registri
Trasferimento dati	load word	lw $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Dato dalla memoria al registro
	store word	sw $s1, 100($s2)	Memoria[$s2 + 100] = $s1	Dato dal registro alla memoria

Figura 3.4 L'architettura MIPS mostrata fino al Paragrafo 3.3. Le parti evidenziate mostrano le istruzioni MIPS introdotte nel Paragrafo 3.3.

indice. Oggi le memorie sono molto più grandi ed il modello software per l'allocazione dei dati molto più sofisticato: l'indirizzo di base di un vettore è comunemente memorizzato in un registro, non potendo essere contenuto nell'offset, come si vedrà nel seguito.

3.4 La rappresentazione delle istruzioni all'interno del calcolatore

Ora è il momento di spiegare la differenza fra il modo in cui l'uomo indica al calcolatore di compiere un'operazione ed il modo in cui il calcolatore vede l'istruzione; prima però è utile un ripasso su come il calcolatore rappresenta i numeri.

Gli uomini imparano a contare in base 10, ma i numeri possono essere rappresentati in qualunque base; ad esempio 123 in base 10 è uguale a 1111011 in base 2.

I numeri sono rappresentati nell'hardware del calcolatore tramite una serie di segnali elettrici di livello alto o basso, quindi la rappresentazione più naturale è quella in base 2; come i numeri in base 10 sono chiamati numeri *decimali*, i numeri in base 2 sono chiamati *binari*. Una singola cifra di un numero binario è quindi l'"unità atomica" di tutti i calcoli e qualunque dato può essere espresso in bit (*binary digit*); questi blocchetti fondamentali possono avere solo due valori, che da punti di vista differenti possono essere descritti come alto o basso, acceso o spento, vero o falso, 1 o 0.

Anche le istruzioni sono memorizzate nel calcolatore come serie di segnali elettrici e possono quindi essere rappresentate da numeri; in effetti ciascun frammento di istruzione può essere visto come un numero e affiancando questi numeri si ottiene l'istruzione vera e propria.

Dato che i registri compaiono nella quasi totalità delle istruzioni, deve esistere una convenzione che consenta di assegnare un numero ad ogni registro: nel linguaggio assembler MIPS i registri da `$s0` a `$s7` sono associati ai numeri da 16 a 23 ed i registri da `$t0` a `$t7` ai numeri da 8 a 15; quindi `$s0` indica il registro 16, `$s1` il registro 17, `$s2` il registro 18, ..., `$t0` il registro 8, `$t1` il registro 9 e così via. Le convenzioni per i restanti registri saranno descritte nei paragrafi successivi.

Tradurre un'istruzione in assembler MIPS nell'istruzione macchina corrispondente

Esempio

Il passo successivo nella definizione del linguaggio MIPS viene fatto tramite un esempio: verrà mostrato il vero linguaggio MIPS dell'istruzione rappresentata simbolicamente come

```
add  $t0, $s1, $s2
```

prima come sequenza di numeri decimali e poi come sequenza di numeri binari.

Soluzione

La rappresentazione decimale è

0	17	18	8	0	32

Ciascuno dei segmenti di un'istruzione è chiamato *campo* (field). Il primo e l'ultimo campo (che in questo caso contengono rispettivamente 0 e 32) indicano al processore MIPS di eseguire una somma. Il secondo campo indica il numero del registro che contiene il primo operando sorgente (17 = `$s1`) ed il terzo si riferisce all'altro operando sorgente della somma (18 = `$s2`); il quarto campo contiene il numero del registro in cui viene memorizzata la somma (8 = `$t0`). Il quinto campo non è usato nel caso dell'addizione ed è quindi posto a 0. Questa istruzione dunque somma i registri `$s1` e `$s2` e pone il risultato in `$t0`.

La stessa istruzione può anche essere rappresentata come sequenza di campi contenenti numeri binari invece che decimali:

000000	10001	10010	01000	00000	100000
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Per distinguerla dal linguaggio assembler, la versione numerica delle istruzioni viene chiamata *linguaggio macchina* ed una sequenza di queste istruzioni *codice macchina*. Contando il numero di bit si vede che l'istruzione è lunga esattamente 32 bit, la stessa dimensione di una parola; mantenendo fede al principio che lega semplicità e regolarità, tutte le istruzioni MIPS sono lunghe 32 bit.

I campi delle istruzioni MIPS

A ciascuno dei campi delle istruzioni MIPS viene associato un nome, per poter fare più agevolmente gli opportuni riferimenti:

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Il significato del nome dato a ciascun campo è il seguente:

- *op*: Operazione base dell'istruzione, tradizionalmente chiamata *codice operativo* o *opcode* (da operating code).

- *rs*: Primo operando sorgente (registro).
- *rt*: Secondo operando sorgente (registro).
- *rd*: Registro destinazione, che contiene il risultato dell'operazione.
- *shamt*: Valore dello scalamento (shift amount). Questo termine verrà spiegato nel Capitolo 4, quando sarà esaminata l'istruzione di scalamento; il campo per il momento non sarà utilizzato e conterrà sempre 0.
- *funct*: Funzione. Seleziona una variante specifica dell'operazione base definita nel campo *op* ed è talvolta chiamato *codice funzione* (function code).

Può nascere un problema quando un'istruzione necessita di campi di dimensioni maggiori rispetto a quelle sopra specificate. Ad esempio l'istruzione load word deve specificare due registri ed una costante; se la costante dovesse essere specificata in uno dei campi da 5 bit, sarebbe limitata al valore di 2^5, cioè 32. Questa costante viene solitamente utilizzata per selezionare elementi specifici all'interno di vettori di grandi dimensioni o altri tipi di strutture dati e spesso deve assumere valori maggiori di 32: un campo da 5 bit è troppo piccolo per essere utile.

Esiste dunque un conflitto fra il desiderio di mantenere la stessa lunghezza per tutte le istruzioni e quello di avere un unico formato; ciò conduce al terzo dei principi per il progetto dell'hardware:

Principio di progetto numero 3: un buon progetto richiede buoni compromessi.

Il compromesso scelto dai progettisti del MIPS è quello di mantenere uguale la lunghezza di tutte le istruzioni, predisponendo formati diversi per tipi di istruzioni diversi. Ad esempio il formato descritto precedentemente è chiamato *tipo-R* (R sta per registro) o *formato-R*; un secondo tipo di formato è detto *tipo-I* o *formato-I* ed è utilizzato dalle istruzioni di trasferimento dati. I campi del formato-I sono

op	rs	rt	address
6 bit	5 bit	5 bit	16 bit

La presenza di un indirizzo (address) su 16 bit indica che le istruzioni load word possono caricare qualsiasi parola in un intervallo di $\pm 2^{15}$ (32768) byte, ossia di $\pm 2^{13}$ (8192) parole, rispetto all'indirizzo contenuto nel registro di base rs.

Si consideri l'istruzione load word esaminata in un esempio del paragrafo precedente:

```
lw  $t0, 32($s3) # il registro temporaneo $t0 assume
                 # il valore A[8]
```

In questo caso viene posto il valore 19 (che rappresenta `$s3`) nel campo rs, il valore 8 (che rappresenta `$t0`) nel campo rt e 32 nel campo address. Si noti che il significato del campo rt è diverso in questo tipo di istruzione: nelle istruzioni load word rt specifica il registro *destinazione*, che riceve il risultato del caricamento.

Benché l'utilizzo di più formati complichi l'hardware, mantenere simili i formati può ridurne la complessità; ad esempio i primi tre campi del tipo-R e del tipo-I hanno le stesse dimensioni e lo stesso nome, mentre il quarto campo del tipo-I ha la stessa lunghezza della somma degli ultimi tre del tipo-R.

I formati si distinguono a seconda del valore contenuto nel primo campo: a ciascun formato è assegnato un insieme di valori del campo op, in modo tale che l'hardware sappia esattamente se trattare la seconda metà dell'istruzione come tre

[illegible]	[illegible]	op	rs	rt	rd	shamt	funct	address
add (somma)	R	0	reg	reg	reg	0	32	n.a.
sub (sottrazione)	R	0	reg	reg	reg	0	34	n.a.
lw (load word)	I	35	reg	reg	n.a.	n.a.	n.a.	indirizzo
sw (store word)	I	43	reg	reg	n.a.	n.a.	n.a.	indirizzo

Figura 3.5 Codifica delle istruzioni MIPS. Nella tabella riportata sopra "reg" indica il numero di un registro, compreso fra 0 e 31, "indirizzo" un indirizzo su 16 bit, "n.a." (non applicabile) significa che il campo non è presente nel formato. Si noti che `add` e `sub` hanno lo stesso valore nel campo op e che l'hardware utilizza il campo funct per specificare l'operazione: somma (32) o sottrazione (34).

campi distinti (nel caso del tipo-R) o come un unico campo (nel caso del tipo-I). La figura 3.5 mostra il contenuto di ciascuno dei campi per le istruzioni introdotte fino al Paragrafo 3.3.

Esempio

Tradurre il linguaggio assembler MIPS in linguaggio macchina

Ora è possibile fare un esempio di tutto il processo, partendo da ciò che scrive il programmatore fino a vedere quello che esegue l'elaboratore. Assumendo che `$t1` contenga l'indirizzo di base del vettore `A` e che `$s2` corrisponda ad `h`, l'assegnazione in C

```
A[300] = h + A[300];
```

viene compilata in

```
lw   $t0, 1200($t1) # il registro temporaneo $t0 assume
                    # il valore A[300]
add  $t0, $s2, $t0  # il registro temporaneo $t0 assume
                    # il valore h + A[300]
sw   $t0, 1200($t1) # memorizza h + A[300] di nuovo
                    # in A[300]
```

Qual è il codice macchina MIPS corrispondente a queste tre istruzioni?

Soluzione

Per comodità conviene inizialmente rappresentare le istruzioni macchina nel formato decimale; dalla figura 3.5 si ottiene

op	rs	rt	rd	address / shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

L'istruzione `lw` è identificata dal numero 35 (vedere figura 3.5) nel primo campo (op). Il registro di base 9 (`$t1`) è specificato nel secondo campo (rs) ed il registro destinazione 8 (`$t0`) nel secondo (rt); l'offset per selezionare `A[300]` (1200 = 300 × 4) si trova nel campo finale (address).

La successiva istruzione di somma è identificata dal numero 0 nel primo campo (op) e dal numero 32 nell'ultimo (funct). I tre registri relativi agli operandi (18, 8 e 8) si trovano nel secondo, terzo e quarto campo e corrispondono rispettivamente a `$s2`, `$t0` e `$t0`.

L'istruzione `sw` è identificata dal valore 43 nel primo campo, mentre il resto dell'istruzione è identico alla `lw` precedente.

Il codice equivalente in formato binario è il seguente (1200 in base 10 equivale a 0000 0100 1011 0000 in base 2):

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Si noti la somiglianza fra le rappresentazioni binarie della prima e dell'ultima istruzione: l'unica differenza consiste nel terzo bit a partire da sinistra.

La figura 3.6 riassume le parti di linguaggio assembler MIPS introdotte in questo paragrafo. Come si vedrà nei Capitoli 5 e 6, la somiglianza delle rappresentazioni binarie di istruzioni concettualmente correlate semplifica il progetto dell'hardware; questo è un altro esempio di struttura regolare dell'architettura MIPS.

Approfondimento: La rappresentazione in base 2 dei numeri decimali è un modo semplice di tradurre in parole i numeri interi positivi. Nel Capitolo 4 verrà spiegato come sono rappresentati i numeri negativi, ma per il momento si assuma come vero che una parola di 32 bit può rappresentare numeri interi compresi fra -2^{31} e $+2^{31}-1$, cioè fra −2147483648 e +2147483647. Questo tipo di rappresentazione è chiamata *complemento a due*.

Quadro d'insieme

Oggi i calcolatori sono costruiti secondo due principi chiave:

1. Le istruzioni sono rappresentate come numeri.
2. I programmi sono in memoria e possono essere letti e scritti, proprio come numeri.

Questi principi portano al concetto di programma memorizzato (*stored program*), un'invenzione fondamentale per l'informatica. La figura 3.7 mostra la portata del concetto; nel caso specifico, la memoria può contenere il codice di un programma per l'elaborazione di testi, il corrispondente codice macchina compilato, il testo su cui il programma compilato sta lavorando e persino il compilatore che ha generato il codice macchina.

Operandi MIPS

Nome	Esempio	Commenti
32 registri	`$s0, $s1, ..., $s7` `$t0, $t1, ..., $t7`	Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. I registri `$s0-$s7` corrispondono a [illegible] ed i registri `$t0-$t7` a 8-15.
2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte, quindi le parole consecutive differiscono di 4 unità. La memoria contiene le strutture dati come i vettori ed i registri riversati (spilled).

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Tre operandi; dati nei registri
	sottrazione	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Tre operandi; dati nei registri
Trasferimento dati	load word	`lw $s1, 100($s2)`	`$s1` = Memoria[`$s2` + 100]	Dato dalla memoria al registro
	store word	`sw $s1, 100($s2)`	Memoria[`$s2` + 100] = `$s1`	Dato dal registro alla memoria

Linguaggio macchina MIPS

Nome	Formato	Esempio						Commenti
add	R	0	18	19	17	0	32	`add $s1, $s2, $s3`
sub	R	0	18	19	17	0	34	`sub $s1, $s2, $s3`
lw	I	35	18	17	100			`lw $s1, 100($s2)`
sw	I	43	18	17	100			`sw $s1, 100($s2)`
Dimensione del campo		6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Tutte le istruzioni MIPS sono lunghe 32 bit
Formato-R	R	op	rs	rt	rd	shamt	funct	Formato istruzioni aritmetiche
Formato-I	I	op	rs	rt	address			Formato istruzioni di trasferimento dati

Figura 3.6 L'architettura MIPS mostrata fino al Paragrafo 3.4. Le parti evidenziate mostrano le strutture del linguaggio macchina MIPS introdotte nel Paragrafo 3.4. I due formati delle istruzioni MIPS visti finora sono R ed I. I primi 16 bit sono gli stessi: entrambi i formati contengono il campo *op*, che indica l'operazione base, il campo *rs*, che specifica un operando sorgente ed il campo *rt*, che specifica l'altro operando sorgente (a parte il caso di load word, in cui specifica l'operando destinazione). Il formato-R suddivide i restanti 16 bit nel campo *rd*, che specifica il registro destinazione, nel campo *shamt*, non usato nel Capitolo 3 e quindi posto sempre a zero, e nel campo *funct*, che specifica ulteriormente l'operazione. Il formato-I utilizza gli ultimi 16 bit come un unico campo *address*.

3.5 Le istruzioni per prendere delle decisioni

L'utilità di un calcolatore automatico risiede nella possibilità di usare ripetutamente una sequenza di istruzioni, con un numero di iterazioni che dipende dal risultato dei calcoli. Quando l'iterazione è completata, va eseguita una diversa sequenza [di istruzioni] e così in molti casi ci sono due diversi flussi [di istruzioni] preceduti da un'istruzione che specifica quale strada deve essere scelta. Tale scelta può essere fatta dipendere dal segno di un numero (i calcolatori trattano lo zero come un segno più). Di conseguenza introduciamo un'istruzione ([l'istruzione di] esecuzione condizionale) che, a seconda del segno di un dato numero, causerà l'esecuzione di uno dei due flussi di istruzioni.

Burks, Goldstine e von Neumann, 1947

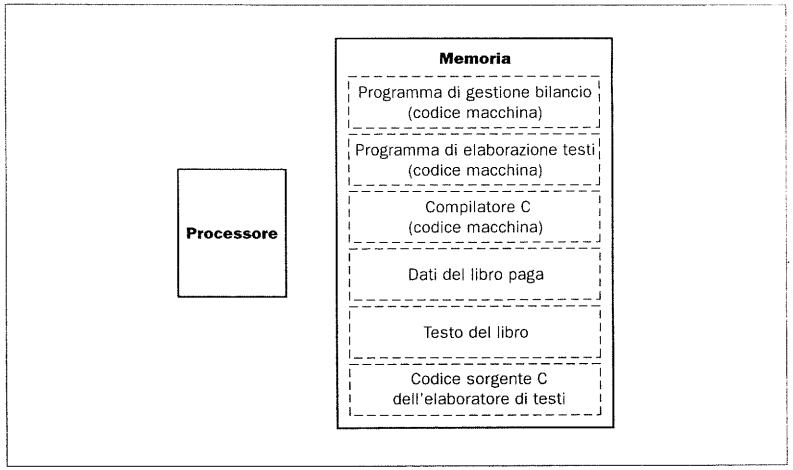


Figura 3.7 Il concetto di programma memorizzato. La memorizzazione dei programmi consente ad un calcolatore che gestisce bilanci di trasformarsi in un batter d'occhio in un calcolatore che aiuta l'autore a scrivere un libro. Il cambiamento avviene semplicemente caricando nella memoria programma e dati e specificando al calcolatore l'indirizzo di partenza per l'esecuzione. Trattare i programmi nello stesso modo dei dati semplifica sia l'hardware relativo alla memoria che il software dei sistemi operativi. Nel caso specifico, la stessa tecnologia di memoria usata per i dati può essere applicata anche ai programmi e programmi come i compilatori, ad esempio, possono tradurre il codice scritto in notazione comprensibile all'uomo in codice che il calcolatore è in grado di capire.

Ciò che contraddistingue un calcolatore da una semplice calcolatrice è la capacità di prendere delle decisioni: in base ai dati di ingresso ed ai valori creati durante le elaborazioni, possono essere eseguite istruzioni diverse. Nei linguaggi di programmazione un'operazione di decisione è comunemente rappresentata con il costrutto *if*, a volte seguito dal costrutto *go to* con un'etichetta. Il linguaggio assembler MIPS contiene due istruzioni per prendere decisioni, simili ad un costrutto *if* seguito dal *go to*; la prima istruzione è

```
beq   registro1, registro2, L1
```

e significa vai all'istruzione etichettata con `L1` se il valore contenuto nel registro `registro1` è uguale al valore di `registro2`; il codice mnemonico `beq` significa *branch if equal* (salta se uguale). La seconda istruzione è

```
bne   registro1, registro2, L1
```

e significa vai all'istruzione etichettata con `L1` se il valore contenuto nel registro `registro1` *non* è uguale al valore di `registro2`; il codice mnemonico `bne` significa *branch if not equal* (salta se non uguale). Queste due istruzioni prendono tradizionalmente il nome di *salti condizionati*.

Esempio

Compilare un costrutto *if* in un salto condizionato

Nel seguente frammento di codice C le variabili sono f, g, h, i e j:

```
        if (i == j) go to L1;
        f = g + h;
L1:     f = f - i;
```

Supponendo che le cinque variabili da f a j corrispondano ai cinque registri da $s0 a $s4, qual è il codice MIPS ottenuto dalla compilazione?

Soluzione

La prima istruzione C verifica l'uguaglianza fra due operandi e controlla l'operazione di sottrazione; dato che entrambi gli operandi sono registri, l'istruzione può essere direttamente convertita in un salto condizionato di tipo branch if equal (l'etichetta L1 sarà definita più avanti):

```
beq  $s3, $s4, L1   # va a L1 se i è uguale a j
```

L'assegnazione successiva del programma C svolge una singola operazione; se tutti gli operandi si trovano nei registri può essere convertita in una sola istruzione:

```
add  $s0, $s1, $s2  # f = g + h (si salta se i è uguale a j)
```

L'ultima linea in C può ancora essere convertita in un'unica istruzione; il problema consiste nello specificare l'indirizzo in modo che l'istruzione di salto condizionato faccia sì che la somma precedente non venga eseguita.

Le istruzioni si trovano nella memoria del calcolatore insieme ai dati, quindi devono avere indirizzi di memoria come tutte le altre celle: l'ultima istruzione è semplicemente preceduta dall'etichetta L1 a cui faceva riferimento la precedente istruzione beq.

```
L1:   sub   $s0, $s0, $s3   # f = f - 1 (eseguita sempre)
```

L'etichetta L1 quindi corrisponde all'indirizzo dell'istruzione di sottrazione.

Si noti che l'assembler evita al compilatore o al programmatore il noioso compito di calcolare gli indirizzi dei salti, proprio come avviene per gli indirizzi di caricamento e di memorizzazione dei dati (vedere Paragrafo 3.9).

Interfaccia Hardware Software

I compilatori creano frequentemente salti ed etichette che non appaiono nel linguaggio di programmazione; evitare il compito di scrivere esplicitamente le etichette ed i salti è uno dei vantaggi di utilizzare i linguaggi di programmazione ad alto livello ed è una delle ragioni per cui è più veloce programmare a questo livello.

Esempio

Compilare un costrutto *if-then-else* in un salto condizionato

Usando le stesse variabili e gli stessi registri dell'esempio precedente, si compili questo frammento di codice C:

```
if (i == j) f = g + h; else f = g - h;
```

Soluzione

La figura 3.8 mostra il diagramma di flusso che il codice MIPS deve rispettare; la prima espressione in C confronta i due operandi per vedere se sono uguali, quindi d'istinto si è tentati di usare l'istruzione beq come nel caso precedente. In generale però è più efficiente controllare la condizione inversa, per saltare la porzione di codice relativa al *then* del costrutto *if* (l'etichetta Else è definita poco oltre):

```
bne  $s3, $s4, Else # vai a Else se i è diverso da j
```

L'assegnazione successiva in C svolge una singola operazione e se tutti gli operandi si trovano all'interno di registri può essere tradotta in una sola istruzione:

```
add  $s0, $s1, $s2  # f = g + h (saltata se i è diverso da j)
```

Ora è necessario spostarsi alla fine dell'istruzione *if*. Quest'esempio introduce un altro tipo di salto, spesso chiamato *salto incondizionato*: l'istruzione corrispondente indica all'elaboratore di eseguire sempre il salto. Per distinguere i salti condizionati da quelli incondizionati, il nome MIPS dati a questi ultimi è *jump*, abbreviato in j (l'etichetta Esci è definita poco oltre).

```
j    Esci          # vai a Esci
```

L'assegnazione che si trova nella porzione relativa all'*else* del costrutto *if* può nuovamente essere tradotta in un'unica istruzione, seguita dall'etichetta Esci. Nell'esempio è mostrata anche l'etichetta Esci che segue l'istruzione, segnando la fine del codice *if-then-else* compilato:

```
Else:  sub   $s0, $s1, $s2 # f = g - h (si salta se i = j)
Esci:
```

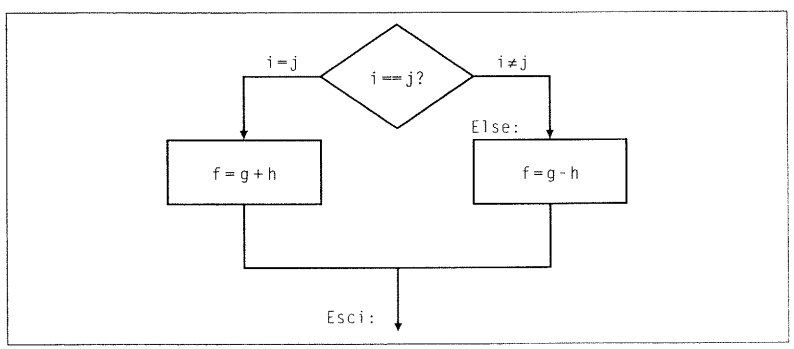


Figura 3.8 Schema dei flussi di istruzioni nel costrutto *if* illustrato in precedenza. Il blocco a sinistra corrisponde al *then* del costrutto *if*, quello a destra all'*else*.

I cicli

Prendere una decisione è importante sia per scegliere fra due alternative (come avviene nei costrutti *if*) sia per ripetere un calcolo più volte (come avviene nei cicli); in entrambi i casi vengono utilizzate le stesse istruzioni assembler.

Esempio

Compilare un ciclo con un indice variabile all'interno di un vettore

Il seguente codice C realizza un ciclo:

```
Ciclo: g = g + A[i];
       i = i + j;
       if (i != h) go to Ciclo;
```

Si assuma che A sia un vettore di 100 elementi e che il compilatore associ le variabili g, h, i e j rispettivamente ai registri $s1, $s2, $s3 e $s4; si assuma inoltre che l'indirizzo di partenza del vettore A sia memorizzato in $s5. Qual è il codice assembler MIPS corrispondente a questo ciclo?

Soluzione

Il primo passo consiste nel caricare A[i] in un registro temporaneo, seguendo l'esempio visto nel Paragrafo 3.3; è però necessario aggiungere l'etichetta Ciclo alla prima istruzione, in modo che sia possibile tornare indietro al termine del ciclo:

```
Ciclo: add $t1, $s3, $s3   # registro temporaneo $t1 = 2 * i
       add $t1, $t1, $t1   # registro temporaneo $t1 = 4 * i
       add $t1, $t1, $s5   # $t1 = indirizzo di A[i]
       lw  $t0, 0($t1)     # registro temporaneo $t0 = A[i]
```

Le due istruzioni successive sommano A[i] a g e j a i:

```
       add $s1, $s1, $t0   # g = g + A[i]
       add $s3, $s3, $s4   # i = i + j
```

L'ultima istruzione esegue il salto indietro a Ciclo se i è diverso da j:

```
       bne $s3, $s2, Ciclo # vai a Ciclo se i è diverso da j
```

Dato che il codice all'interno del ciclo modifica i, ogni volta è necessario moltiplicare tale valore per 4; il Paragrafo 3.11 mostrerà come evitare le "moltiplicazioni" all'interno di cicli come questo.

Interfaccia Hardware Software

Le sequenze di istruzioni che terminano con un'istruzione di salto sono talmente fondamentali per la compilazione che viene dato loro un nome: un *blocco base* è una sequenza di istruzioni che non contiene né istruzioni di salto (con la possibile eccezione della fine) né etichette di destinazione dei salti (con la possibile eccezione dell'inizio). Una delle prime fasi della compilazione consiste nel suddividere il programma nei blocchi base di cui è composto.

Esempio

Compilare un ciclo *while*

Di solito i programmatori non usano l'istruzione *go to* per scrivere i cicli ed è quindi compito del compilatore tradurre i cicli tradizionali in codice MIPS. Il seguente frammento di programma C è un classico ciclo:

```
while (salva[i] == k)
        i = i + j;
```

Si assuma che i, j e k corrispondano ai registri $s3, $s4 e $s5 e che l'indirizzo di base del vettore salva sia in $s6. Qual è il codice assembler MIPS corrispondente a questo ciclo?

Soluzione

Il primo passo consiste nel caricare salva[i] in un registro temporaneo, in modo analogo all'esempio precedente:

```
Ciclo: add $t1, $s3, $s3 # registro temporaneo $t1 = 2 * i
       add $t1, $t1, $t1 # registro temporaneo $t1 = 4 * i
       add $t1, $t1, $s6 # $t1 = indirizzo di salva[i]
       lw  $t0, 0($t1)   # registro temporaneo $t0 =
salva[i]
```

L'istruzione successiva esegue il controllo del ciclo, uscendo se salva[i] ≠ k:

```
       bne $t0, $s5, Esci    # vai a Esci se salva[i] = k
```

L'istruzione successiva somma i a j:

```
       add $s3, $s3, $s4 # i = i + j
```

Alla fine del ciclo c'è un salto indietro al test del *while* che si trova all'inizio del ciclo; basta aggiungere l'etichetta Esci alla fine ed il codice è completo:

```
       j   Ciclo          # vai a Ciclo
Esci:
```

Si veda l'Esercizio 3.9 per l'ottimizzazione di questa sequenza.

I test per verificare l'uguaglianza o la diseguaglianza fra due variabili sono probabilmente i più diffusi, ma talvolta è utile riconoscere se una data variabile è minore di un'altra, ad esempio nel caso di un ciclo *for* con cui si voglia controllare se una variabile indice è minore di zero. Questi confronti nel linguaggio assembler MIPS sono eseguiti da un'istruzione che confronta due registri e pone un terzo registro pari a 1 se il primo è minore del secondo e pari a 0 se si verifica la condizione opposta; l'istruzione MIPS si chiama *set on less than* (poni a uno se minore) o slt. Ad esempio

```
slt   $t0, $s3, $s4
```

vuol dire che il registro $t0 viene posto a 1 se il valore nel registro $s3 è minore del valore nel registro $s4; in caso contrario il registro $t0 è posto a 0.

Interfaccia Hardware Software

I compilatori MIPS utilizzano le istruzioni `slt`, `beq`, `bne` e il valore dello zero contenuto sempre nel registro di sola lettura `$zero` per creare tutte le condizioni relazionali: uguale, diverso, minore, minore o uguale, maggiore, maggiore o uguale (come ci si può aspettare, il registro `$zero` corrisponde al registro 0).

Esempio

Compilare un test del tipo "minore di"

Qual è il codice per verificare se la variabile `a` (corrispondente al registro `$s0`) è minore della variabile `b` (registro `$s1`) e saltare all'etichetta `Minore` se la condizione è vera?

Soluzione

Il primo passo consiste nell'usare l'istruzione set on less ed un registro temporaneo:

```
slt  $t0, $s0, $s1        # $t0 diventa 1 se $s0 < $s1 (a < b)
```

Il registro `$t0` diventa uguale a 1 se `a` è minore di `b`; di conseguenza un salto condizionato su `$t0` diverso da zero ha l'effetto di eseguire il salto se `a` è minore di `b`. Il registro `$zero` contiene sempre il valore 0, quindi il test finale è realizzato con un'istruzione `bne` che confronta i registri `$t0` e `$zero`:

```
bne  $t0, $zero, Minore  # vai a Minore se $t0 ≠ 0
                         # (cioè se a < b)
```

La coppia di istruzioni `slt` e `bne` implementa il salto che dipende dalla condizione "minore di".

Seguendo l'indicazione di von Neumann sulla semplicità dei "dispositivi", l'architettura MIPS non include nessuna istruzione specifica di salto condizionato nel caso "minore di". L'hardware diventerebbe troppo complesso: aumenterebbe la durata del ciclo di clock oppure questa istruzione richiederebbe un numero maggiore di cicli di clock. Sono più utili due istruzioni veloci.

Il costrutto case/switch

La maggioranza dei linguaggi di programmazione possiede un costrutto *case* o *switch* che consente al programmatore di scegliere fra più alternative in base ad un unico valore. Il costrutto *switch* può essere realizzato mediante una sequenza di condizioni, trasformando lo *switch* in una catena di costrutti *if-then-else*. Talvolta però le diverse alternative di un'istruzione *switch* possono essere specificate in modo efficiente memorizzandole in una tabella di indirizzi in cui si trovano le diverse sequenze di istruzioni; questa tabella si chiama *tabella degli indirizzi di salto* (jump address table) ed il programma deve solo conoscere un indice per saltare alla sequenza di codice appropriata. La tabella è quindi semplicemente un vettore di parole che contiene gli indirizzi corrispondenti alle etichette presenti nel codice.

Per permettere ciò, i calcolatori come il MIPS includono un'istruzione di salto tramite registro (*jump register* o `jr`), che effettua un salto incondizionato all'indirizzo specificato in un registro. Il programma carica l'indice opportuno della tabella di salto in un registro, quindi salta all'indirizzo scelto con un'istruzione di salto tramite registro.

Esempio

Compilare un costrutto *switch* usando una tabella degli indirizzi di salto

La versione C di un costrutto di tipo case è chiamata *switch*. Il seguente codice C sceglie fra quattro alternative a seconda che il valore di `k` sia 0, 1, 2 o 3.

```
switch (k) {
      case 0: f = i + j; break;    /*k = 0 */;
      case 1: f = g + h; break;    /*k = 1 */;
      case 2: f = g - h; break;    /*k = 2 */;
      case 3: f = i - j; break;    /*k = 3 */;
}
```

Si assuma che le sei variabili da `f` a `k` corrispondano ai sei registri da `$s0` a `$s5` e che il registro `$t2` contenga il valore 4. Qual è il corrispondente codice MIPS?

Soluzione

Si usa la variabile `k` del costrutto *switch* per indicizzare la tabella degli indirizzi di salto e poi si effettua il salto all'indirizzo caricato dalla tabella. Per prima cosa occorre analizzare `k` per verificare che soddisfi ad una delle quattro condizioni (0 <= k <= 3); se questo non avviene, il programma esce dal costrutto *switch*.

```
slt   $t3, $s5, $zero     # verifica se k < 0
bne   $t3, $zero, Esci    # se k < 0 salta a Esci
slt   $t3, $s5, $t2       # verifica se k > 4
bne   $t3, $zero, Esci    # se k >= 4 salta a Esci
```

Dal momento che la variabile `k` rappresenta l'indice in una tabella di parole, bisogna moltiplicare `k` per 4 per convertirlo in byte:

```
add   $t1, $s5, $s5       # $t1 = 2 * k
add   $t1, $t1, $t1       # $t1 = 4 * k
```

Si assuma che quattro parole di memoria consecutive, a partire dall'indirizzo specificato in `$t4`, contengano gli indirizzi corrispondenti alle etichette `L0`, `L1`, `L2` ed `L3`; è possibile specificare il corretto indirizzo di salto nel modo seguente:

```
add   $t1, $t1, $t4       # $t1 = indirizzo
                          # di TabellaSalti[k]
lw    $t0, 0($t1)         # $t0 = TabellaSalti[k]
```

Un'istruzione di salto tramite registro salta all'indirizzo specificato da `$t0`.

```
jr    $t0                 # salto dipendente dal contenuto
                          # di $t0
```

In questo esempio i primi tre casi dello *switch* sono molto simili: un'etichetta, una sola istruzione che svolge quanto richiesto dal *case* ed il salto all'uscita del costrutto *switch*:

```
L0:   add $s0, $s3, $s4   # k = 0, quindi f = i + j
      j   Esci            # fine del case: salta a Esci
L1:   add $s0, $s1, $s2   # k = 1, quindi f = g + h
      j   Esci            # fine del case: salta a Esci
L2:   sub $s0, $s1, $s2   # k = 2, quindi f = g - h
      j   Esci            # fine del case: salta a Esci
```

Un esempio più complesso potrebbe avere un maggior numero di istruzioni in ciascun caso.

L'ultimo caso non necessita del salto di uscita, essendo l'ultima delle istruzioni dello switch; l'uscita dal costrutto switch è evidenziata dall'etichetta `Esci`, che segue l'ultima istruzione:

```
L3:   sub $s0, $s3, $s4   # k = 3, quindi f = i - j
Esci:                     # fine del costrutto switch
```

La figura 3.9 riassume le parti di linguaggio assembler MIPS introdotte in questo paragrafo. In questa fase alle rappresentazioni simboliche già trattate sono state aggiunte le istruzioni di salto e la possibilità di fissare a 0 in modo permanente il valore di un registro.

Approfondimento: Il lettore non deve al momento preoccuparsi dei *salti ritardati* (delayed branches), che saranno trattati nel Capitolo 6: l'assembler MIPS li rende trasparenti al programmatore. Per i programmatori C non familiari con l'infinitamente abusato costrutto `go to`, esso trasferisce il controllo dal punto in cui compare l'istruzione al punto in cui compare l'etichetta corrispondente.

3.6 La gestione delle procedure

Una procedura, o subroutine, è uno strumento che i programmatori usano per strutturare i programmi, sia per renderli più facili da capire che per permettere il riutilizzo del codice. Le procedure consentono al programmatore di concentrarsi su una parte del problema: i parametri rappresentano una barriera fra la procedura ed il resto del programma e permettono di scambiare i dati.

Si può pensare alle procedure come a delle spie che partono con un piano segreto, acquisiscono risorse, svolgono il loro lavoro, nascondono le tracce e tornano al punto di partenza con dei risultati; quando la missione è completata, tutto il resto prosegue normalmente. In più le spie operano soltanto in base a ciò che "devono sapere" e non possono fare assunzioni su chi le ha assoldate.

In modo simile, durante l'esecuzione di una procedura il programma segue questi sei passi:

1. Mettere i parametri in un luogo accessibile alla procedura.
2. Trasferire il controllo alla procedura.
3. Acquisire le risorse necessarie alla memorizzazione dei dati.
4. Eseguire il compito richiesto.
5. Mettere il risultato in un luogo accessibile al programma chiamante.
6. Restituire il controllo al punto di origine.

Operandi MIPS

Nome	Esempio	Commenti

32 registri	`$s0, $s1, ..., $s7`
	`$t0, $t1, ..., $t7,`
	`$zero`
		Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. I registri `$s0-$s7` corrispondono a `16-23` ed i registri `$t0-$t7` a `8-15`. Il registro `$zero` ha sempre valore `0`.

2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte, quindi le parole consecutive differiscono di 4 unità. La memoria contiene le strutture dati come i vettori ed i registri riversati (spilled).

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Tre operandi; dati nei registri
	sottrazione	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Tre operandi; dati nei registri
Trasferimento dati	load word	`lw $s1, 100($s2)`	`$s1` = Memoria[`$s2` + 100]	Dato dalla memoria al registro
	store word	`sw $s1, 100($s2)`	Memoria[`$s2` + 100] = `$s1`	Dato dal registro alla memoria
Salto condizionato	branch on equal	`beq $s1, $s2, L`	if (`$s1 == $s2`) go to `L`	Controllo dell'uguaglianza ed eventuale salto
	branch on not equal	`bne $s1, $s2, L`	if (`$s1 != $s2`) go to `L`	Controllo della disuguaglianza ed eventuale salto
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0;	Confronto del tipo "minore di": usato con `beq`, `bne`
Salto incondizionato	jump	`j 2500`	go to `10000`	Salto ad un indirizzo specificato
	jump register	`jr $t1`	go to `$t1`	Usato nei costrutti *switch*

Linguaggio macchina MIPS

Nome	Formato	Esempio						Commenti
`add`	R	0	18	19	17	0	32	`add $s1, $s2, $s3`
`sub`	R	0	18	19	17	0	34	`sub $s1, $s2, $s3`
`lw`	I	35	18	17	100			`lw $s1, 100($s2)`
`sw`	I	43	18	17	100			`sw $s1, 100($s2)`
`beq`	I	4	17	18	25			`beq $s1, $s2, 100`
`bne`	I	5	17	18	25			`bne $s1, $s2, 100`
`slt`	R	0	18	19	17	0	42	`slt $s1, $s2, $s3`
`j`	J	2	2500					`j 10000` (vedere Paragrafo 3.8)
`jr`	R	0	9	0	0	0	8	`jr $t1`
Dimensione del campo		6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Tutte le istruzioni MIPS sono lunghe 32 bit
Formato-R	R	op	rs	rt	rd	shamt	funct	Formato istruzioni aritmetiche
Formato-I	I	op	rs	rt	address			Formato istruzioni di trasferimento dati e istruzioni di salto condizionato

Figura 3.9 L'architettura MIPS mostrata fino al Paragrafo 3.5. Le parti evidenziate mostrano le strutture MIPS introdotte nel Paragrafo 3.5. Il formato-J, usato per le istruzioni jump, sarà spiegato nel Paragrafo 3.8, dove verranno anche spiegati i valori corretti per l'indirizzo delle istruzioni di salto.

Come riportato in precedenza, i registri sono il luogo che permette l'accesso più rapido ai dati, quindi è bene usarli il più possibile; il software MIPS di conseguenza alloca i seguenti registri (dei suoi 32) per le chiamate a procedura:

- `$a0 - $a3`: quattro registri argomento per il passaggio dei parametri;
- `$v0 - $v1`: due registri valore per la restituzione dei valori;
- `$ra`: un registro di ritorno per tornare al punto di origine.

Oltre ad allocare questi registri, il linguaggio assembler MIPS include un'istruzione apposita per le procedure, che salta ad un indirizzo e contemporaneamente salva l'indirizzo dell'istruzione successiva nel registro `$ra`. L'istruzione *jump-and-link* (`jal`, salta e crea un collegamento) si scrive semplicemente come

```
jal    IndirizzoProcedura
```

La parte *link* del nome dell'istruzione si riferisce al fatto che viene creato un indirizzo, o collegamento (link), che permette alla procedura di tornare all'indirizzo corretto; questo "link", memorizzato nel registro `$ra`, è detto *indirizzo di ritorno*. L'indirizzo di ritorno è necessario perché la stessa procedura può essere richiamata in più parti del programma.

Nell'idea della memorizzazione dei programmi è implicita la necessità di mantenere in un registro l'indirizzo dell'istruzione in esecuzione; per ragioni storiche questo registro è quasi sempre chiamato *program counter* (contatore di programma), abbreviato in *PC* nell'architettura MIPS, anche se il nome più corretto sarebbe *registro dell'indirizzo dell'istruzione* (instruction address register). L'istruzione `jal` salva il valore PC + 4 nel registro `$ra` per creare il collegamento all'indirizzo di ritorno dalla procedura.

L'istruzione per effettuare il salto di ritorno è già nota:

```
jr     $ra
```

L'istruzione di salto tramite registro, usata in precedenza per il costrutto *switch*, salta all'indirizzo memorizzato nel registro `$ra`, facendo esattamente quanto desiderato. Il *programma chiamante* mette i valori dei parametri da passare alla procedura nei registri `$a0` - `$a3` e utilizza l'istruzione `jal X` per saltare alla procedura `X` (chiamata anche *programma chiamato*); il programma chiamato esegue le operazioni richieste, memorizza i risultati nei registri `$v0` – `$v1` e restituisce il controllo al chiamante con l'istruzione `jr $ra`.

Usare più registri

Si supponga che un compilatore abbia bisogno, all'interno di una procedura, di un numero maggiore di registri rispetto ai quattro per i parametri ed ai due per la restituzione dei valori. Dal momento che quando la missione è completa le tracce vengono cancellate, il valore dei registri alla fine della procedura deve essere lo stesso che avevano prima della chiamata; in questa situazione è necessario riversare i registri in memoria, un esempio di quanto è stato descritto nella seconda "Interfaccia Hardware-Software" del Paragrafo 3.3.

La struttura ideale per riversare i registri è lo *stack* (pila), una coda del tipo last-in first-out (cioè l'ultimo ad entrare è il primo ad uscire). Lo stack ha bisogno del puntatore all'indirizzo del dato introdotto più recentemente, per indicare dove la prossima procedura può memorizzare i registri da riversare e dove può recuperare i vecchi valori dei registri. Il puntatore allo stack è aggiornato ogni volta che viene inserito o estratto il valore di un registro. L'uso dello stack è così comune che esistono dei termini gergali per indicare il trasferimento di dati da e verso lo stack: per *push* si intende la memorizzazione di un dato nello stack e per *pop* la sua estrazione.

Il software MIPS alloca un altro registro appositamente per lo stack: lo *stack pointer* (`$sp`), usato per salvare i registri che servono al programma chiamato. Per ragioni storiche, lo stack "cresce" a partire da indirizzi di memoria alti verso indirizzi di memoria bassi. Questa convenzione indica che quando vengono inseriti dei dati nello stack il valore dello stack pointer diminuisce; al contrario, quando sono estratti dati dallo stack aumenta il valore dello stack pointer, riducendo la dimensione dello stack.

Compilare una procedura che non richiama altre procedure

Esempio

Si trasformi in procedura il secondo esempio del Paragrafo 3.2:

```
int esempio_foglia (int g, int h, int i, int j)
{
        int f;

        f = (g + h) - (i + j);
        return f;
}
```

Per tutto il resto del paragrafo si assumerà la possibilità di sommare o sottrarre costanti come 4, 8 o 12; il Paragrafo 3.8 svelerà il modo in cui le costanti sono trattate nel linguaggio assembler MIPS. Qual è il codice MIPS ottenuto dalla compilazione?

Soluzione

I parametri `g`, `h`, `i` e `j` corrispondono ai registri argomento `$a0`, `$a1`, `$a2` e `$a3`, mentre `f` corrisponde a `$s0`. Il programma compilato inizia con l'etichetta della procedura:

```
esempio_foglia:
```

Il passo successivo consiste nel salvare i registri usati dalla procedura. L'assegnazione C nel corpo della procedura è identica a quella dell'esempio citato, che usa due registri temporanei; è quindi necessario salvare tre registri: `$s0`, `$t0` e `$t1`. Si crea quindi spazio per tre parole all'interno dello stack, dove vengono memorizzati i vecchi valori:

```
sub  $sp, $sp, 12  # aggiornamento dello stack per fare
                   # spazio a tre elementi
sw   $t1, 8($sp)   # salvataggio del registro $t1
sw   $t0, 4($sp)   # salvataggio del registro $t0
sw   $s0, 0($sp)   # salvataggio del registro $s0
```

La figura 3.10 mostra lo stack prima, durante e dopo la chiamata alla procedura. Le tre istruzioni successive corrispondono al corpo della procedura e sono analoghe all'esempio precedente:

```
add  $t0, $a0, $a1 # il registro $t0 contiene g + h
add  $t1, $a2, $a3 # il registro $t1 contiene i + j
sub  $s0, $t0, $t1 # f = $t0 - $t1, cioè (g + h) - (i + j)
```

Per restituire il valore di `f`, occorre copiarlo in un registro di ritorno:

```
add  $v0, $s0, $zero  # restituzione di f ($v0 = $s0 + 0)
```

Prima del ritorno al programma chiamante, i vecchi valori vengono ripristinati all'interno dei registri, estraendoli dallo stack:

```
lw   $s0, 0($sp)   # ripristino del registro $s0
lw   $t0, 4($sp)   # ripristino del registro $t0
lw   $t1, 8($sp)   # ripristino del registro $t1
add  $sp, $sp, 12  # aggiornamento dello stack per eliminare
                   # tre elementi
```

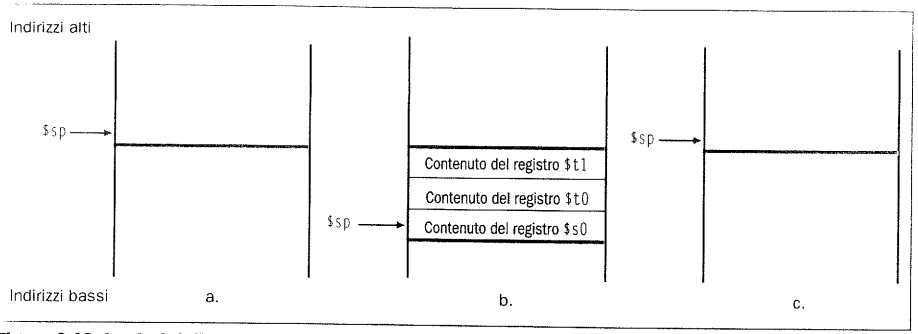


Figura 3.10 I valori dello stack pointer e dello stack prima (a), durante (b) e dopo (c) la chiamata a procedura. Lo stack pointer punta sempre alla cima ("top") dello stack (in questo disegno punta all'ultima parola presente nello stack).

La procedura termina con un'istruzione di salto tramite registro che utilizza l'indirizzo di ritorno:

```
jr   $ra # ritorno al programma chiamante
```

Nell'esempio sopra riportato sono stati usati dei registri temporanei ed è stata fatta l'assunzione che il loro valore originale dovesse essere salvato e ripristinato. Per evitare di salvare e ripristinare registri il cui valore non viene mai utilizzato, cosa che può accadere con i registri temporanei, il software MIPS offre due classi di registri:

- `$t0` - `$t9`: 10 registri temporanei che *non* sono preservati in caso di chiamata a procedura;
- `$s0` - `$s7`: 8 registri che devono essere preservati in caso di chiamata a procedura (se utilizzati devono essere salvati e rispristinati dal programma chiamato).

Questa semplice convenzione riduce la necessità di salvare registri in memoria. Nell'esempio precedente, dato che il programma chiamante non si aspetta che i registri `$t0` e `$t1` siano preservati durante la chiamata a procedura, si possono eliminare dal codice due istruzioni di store e due istruzione di load. È però necessario in ogni caso salvare e ripristinare `$s0`, dal momento che la procedura è costretta ad assumere che il chiamante abbia bisogno del suo valore originale.

Procedure annidate

Le procedure che non chiamano altre procedure sono chiamate *foglie*; la vita del programmatore sarebbe più semplice se tutte le procedure fossero di questo tipo, ma ciò non è vero. Una spia può utilizzare altre spie per compiere parte di una missione e queste spie a loro volta possono usarne altre; allo stesso modo le procedure possono richiamare altre procedure. In più le procedure ricorsive invocano persino "cloni" di se stesse. Occorre fare attenzione all'uso dei registri nelle procedure; ne occorre ancora di più se le procedure chiamano altre procedure.

Si supponga ad esempio che il programma principale chiami la procedura A con un parametro uguale a 3, mettendo il valore 3 nel registro `$a0` ed usando l'istruzione `jal A`. Si supponga poi che la procedura A chiami la procedura B con una `jal B` passandole il valore 7, anch'esso posto in `$a0`; dato che A non ha ancora terminato il proprio compito si verifica un conflitto nell'uso del registro `$a0`. In modo analogo c'è un conflitto sull'indirizzo di ritorno dalla procedura, memorizzato in `$ra`, perché dopo la chiamata a B esso contiene l'indirizzo di ritorno di B; se non si adottano misure per prevenire il problema, la procedura A non sarà più in grado di restituire il controllo al proprio chiamante.

Una soluzione consiste nel salvare nello stack tutti i registri che devono essere preservati. Il programma chiamante memorizza nello stack i registri argomento (`$a0` - `$a3`) o i registri temporanei (`$t0` - `$t9`) di cui ha ancora bisogno dopo la chiamata; il chiamato salva nello stack il registro di ritorno `$ra` e gli altri registri (`$s0` - `$s7`) che utilizza. Lo stack pointer `$sp` è aggiornato per tener conto del numero di registri memorizzati nello stack; alla fine i registri vengono ripristinati e lo stack pointer riaggiornato.

Compilare una procedura ricorsiva, mostrando l'annidamento delle procedure

Esempio

Si consideri una procedura ricorsiva che calcola i numeri fattoriali:

```
int fact (int n)
{
     if (n < 1) return (1);
          else return (n * fatt (n-1));
}
```

Si assuma la possibilità di sommare o sottrarre costanti come 1 o 8, come verrà mostrato nel Paragrafo 3.8. Qual è il codice MIPS ottenuto dalla compilazione?

Soluzione

Il parametro `n` corrisponde al registro argomento `$a0`. Il programma compilato inizia con l'etichetta della procedura, quindi salva nello stack due registri, l'indirizzo di ritorno e `$a0`:

```
fatt:
     sub  $sp, $sp, 8     # aggiornamento dello stack per
                          # fare spazio a due elementi
     sw   $ra, 4($sp)     # salvataggio del registro di
                          # ritorno
     sw   $a0, 0($sp)     # salvataggio del parametro n
```

La prima volta che la procedura `fatt` viene chiamata, `sw` salva un indirizzo interno al programma chiamante. Le due istruzioni successive verificano se `n` è minore di 1, saltando a `L1` se `n <= 1`.

```
     slt  $t0, $a0, 1     # test per n < 1
     beq  $t0, $zero, L1  # se n >= 1 salta a L1
```

Se `n` è minore di 1, `fatt` restituisce 1 mettendolo in un registro valore: somma 1 a 0 e lo memorizza in `$v0`. Quindi ripristina dallo stack i due valori salvati e salta all'indirizzo di ritorno:

```
        add  $v0, $zero, 1   # restituisce 1
        add  $sp, $sp, 8     # aggiornamento dello stack per
                             # eliminare due elementi
        jr   $ra             # ritorno all'istruzione
                             # successiva a jal
```

Prima dell'aggiornamento dello stack pointer si sarebbero dovuti ripristinare $a0 e $ra, ma dato che non cambiano quando n è minore di 1 è possibile saltare queste istruzioni.

Se n non è minore di 1, il parametro n viene decrementato e viene nuovamente chiamata la procedura fatt passandole tale valore:

```
L1:     sub  $a0, $a0, 1     # n > 1: l'argomento diventa (n - 1)
        jal  fatt            # chiamata a fatt con (n - 1)
```

L'istruzione successiva è quella a cui ritorna la procedura fatt; il vecchio indirizzo di ritorno ed il vecchio valore del parametro sono ripristinati, oltre ad eseguire l'aggiornamento dello stack pointer:

```
        lw   $a0, 0($sp)     # ritorno da jal: ripristino di n
        lw   $ra, 4($sp)     # ripristino del registro di ritorno
        sub  $sp, $sp, 8     # aggiornamento dello stack per
                             # eliminare due elementi
```

Quindi, nel registro valore $v0 viene memorizzato il prodotto del valore corrente per il vecchio parametro $a0; qui si assume l'esistenza di un'operazione di moltiplicazione, che sarà analizzata in dettaglio nel Capitolo 4:

```
        mul  $v0, $a0, $v0   # restituzione di n * fatt (n - 1)
```

Infine la procedura fatt salta nuovamente all'indirizzo di ritorno:

```
        jr   $ra             # ritorno al programma chiamante
```

La figura 3.11 riassume ciò che viene preservato durante una chiamata a procedura. Si noti che sono usati diversi accorgimenti per preservare lo stack. La parte di stack che sta sopra a $sp è preservata assicurandosi semplicemente che il programma chiamato non scriva oltre $sp; $sp è a sua volta preservato perché il programma chiamato somma e sottrae sempre lo stesso valore e gli altri registri sono preservati (se necessario) salvandoli nello stack e ripristinandoli in un secondo tempo. Queste azioni garantiscono anche che il chiamante riottenga dallo stack lo stesso valore che vi aveva memorizzato, dato che il chiamato si impegna a non modificare né $sp né la parte dello stack usata dal chiamante, cioè quella che si trova al di sopra di $sp.

Preservati	Non preservati
Registri argomento: $a0 - $a3	Registri di ritorno: $v0 - $v1
Registri: $s0 - $s7	Registri temporanei: $t0 - $t9
Stack pointer: $sp	
Registro di ritorno: $ra	
Stack al di sopra dello stack pointer	Stack al di sotto dello stack pointer

Figura 3.11 Che cosa viene e che cosa non viene preservato durante una chiamata a procedura. Anche il frame pointer register ed il global pointer register, discussi nel seguito, se necessario sono preservati durante le chiamate.

Allocare spazio per nuovi dati

Un'ultima complicazione consiste nel fatto che lo stack può essere usato anche per memorizzare le variabili locali delle procedure che non stanno nei registri, ad esempio vettori o strutture locali. Il segmento dello stack che contiene i registri salvati da una procedura e le variabili locali prende il nome di *finestra di attivazione* (procedure frame o activation record). La figura 3.12 mostra lo stato dello stack prima, durante e dopo una procedura.

Alcuni software MIPS utilizzano un *frame pointer* ($fp, puntatore a frame) per indirizzare la prima parola del frame di una procedura. Lo stack pointer può cambiare durante l'esecuzione di una procedura, così i riferimenti alle variabili locali in memoria potrebbero avere offset diversi a seconda della posizione nella procedura, rendendo le procedure difficili da leggere. In alternativa il frame pointer offre una base di partenza stabile per i riferimenti alla memoria locale. Si noti che il record di attivazione è presente nello stack sia che il frame pointer sia usato esplicitamente, sia che non sia usato; nell'esempio precedente $fp non è stato usato, non essendo cambiato il valore di $sp all'interno della procedura: lo stack in questo caso viene aggiornato solo all'entrata e all'uscita della procedura.

La figura 3.13 riassume le convenzioni per i registri dell'assembler MIPS e la figura 3.14 le parti di istruzioni MIPS finora descritte.

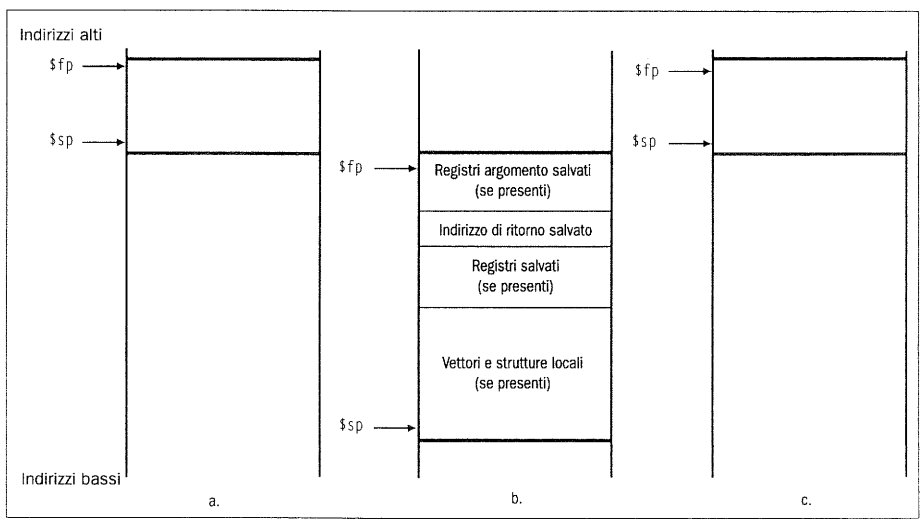


Figura 3.12 L'allocazione dello stack prima (a), durante (b) e dopo (c) la chiamata a procedura. Il frame pointer ($fp) punta alla prima parola del frame, spesso ad un registro salvato, e lo stack pointer ($sp) punta alla cima dello stack. Lo stack è aggiornato per fare spazio a tutti i registri salvati ed alle variabili locali residenti in memoria. Dal momento che lo stack pointer può cambiare durante l'esecuzione del programma, è più facile per i programmatori fare riferimento alle variabili tramite il frame pointer, che invece rimane stabile (anche se tutto ciò potrebbe essere fatto con lo stack pointer e un po' di aritmetica sugli indirizzi). Se in una procedura non ci sono variabili locali sullo stack, il calcolatore risparmia tempo *non* assegnando il frame pointer. Quando viene usato, il frame pointer è inizializzato con l'indirizzo di $sp durante la chiamata e $sp è ripristinato usando $fp.

Nome	Numero del registro	Uso	Preservato in una chiamata a procedura?
$zero	0	valore costante 0	n.a.
$v0-$v1	2-3	valori dei risultati e valutazione di espressioni	no
$a0-$a3	4-7	parametri	sì
$t0-$t7	8-15	variabili temporanee	no
$s0-$s7	16-23	variabili salvate	sì
$t8-$t9	24-25	altre variabili temporanee	no
$gp	28	global pointer	sì
$sp	29	stack pointer	sì
$fp	30	frame pointer	sì
$ra	31	indirizzo di ritorno	sì

Figura 3.13 Convenzioni per i registri MIPS. Il registro 1, chiamato `$at`, è riservato all'assemblatore (vedere Paragrafo 3.9) ed i registri 26-27, chiamati `$k0-$k1`, sono riservati al sistema operativo.

Approfondimento: Che cosa succederebbe se si volessero passare più di quattro parametri ad una procedura? Per convenzione i parametri extra vengono messi nello stack al di sopra del frame pointer: la procedura si aspetta i primi quattro parametri nei registri `$a0-$a3` ed i restanti in memoria, indirizzabili con il frame pointer.

Come citato nella didascalia di figura 3.12, il frame pointer è una buona scelta perché in questo modo tutti i riferimenti alle variabili nello stack all'interno di una procedura hanno lo stesso offset. Il frame pointer, però, non è strettamente necessario: il compilatore GNU C per il MIPS usa il frame pointer, ma il MIPS / Silicon Graphics no, utilizzando il registro 30 come un registro aggiuntivo (`$s8`).

Approfondimento: L'istruzione `jal` salva effettivamente l'indirizzo dell'istruzione successiva nel registro `$ra`, consentendo alla procedura di ritornare semplicemente con un `jr $ra`.

Interfaccia Hardware Software

Quando una variabile C viene memorizzata, la sua interpretazione dipende sia dal *tipo* che dalla classe di memorizzazione (*storage class*). I tipi, ad esempio gli interi e i caratteri, saranno analizzati in dettaglio nel Capitolo 4. Il C ha due classi di memorizzazione: *automatica* e *statica*. Le variabili automatiche sono locali ad una procedura e sono scartate all'uscita di quest'ultima; le variabili statiche continuano ad esistere all'interno ed all'esterno delle procedure. Le variabili C dichiarate al di fuori di tutte le procedure sono considerate statiche, come anche tutte le variabili precedute dalla parola chiave `static`; tutte le altre sono automatiche. Per semplificare l'accesso ai dati statici, il software MIPS si riserva un altro registro, chiamato *global pointer* (puntatore globale) o `$gp`.

Nel Paragrafo 3.9 si vedrà come sono allocati in memoria i dati statici.

Operandi MIPS

Nome	Esempio	Commenti
32 registri	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. Il registro `$zero` ha sempre valore 0. $gp (28) è il global pointer, $sp (29) lo stack pointer, $fp (30) il frame pointer e $ra (31) l'indirizzo di ritorno.
2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte, quindi le parole consecutive differiscono di 4 unità. La memoria contiene le strutture dati, come i vettori, i registri riversati (spilled) e i registri salvati durante le chiamate a procedura.

$gp $RA

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	add $s1, $s2, $s3	$s1 = $s2 + $s3	Tre operandi; dati nei registri
	sottrazione	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Tre operandi; dati nei registri
Trasferimento dati	load word	lw $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Dato dalla memoria al registro
	store word	sw $s1, 100($s2)	Memoria[$s2 + 100] = $s1	Dato dal registro alla memoria
Salto condizionato	branch on equal	beq $s1, $s2, L	if ($s1 == $s2) go to L	Controllo dell'uguaglianza ed eventuale salto
	branch on not equal	bne $s1, $s2, L	if ($s1 != $s2) go to L	Controllo della disuguaglianza ed eventuale salto
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0;	Confronto del tipo "minore di"; usato con beq, bne
Salto incondizionato	jump	j 2500	go to 10000	Salto ad un indirizzo specificato
	jump register	jr $ra	go to $ra	Usato nei costrutti *switch* e nell'uscita dalle procedure
	jump and link	jal 2500	$ra = PC + 4; go to 10000	Per le chiamate a procedura

Linguaggio macchina MIPS

Nome	Formato	Esempio						Commenti
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
lw	I	35	18	17	100			lw $s1, 100($s2)
sw	I	43	18	17	100			sw $s1, 100($s2)
beq	I	4	17	18	25			beq $s1, $s2, 100
bne	I	5	17	18	25			bne $s1, $s2, 100
slt	R	0	18	19	17	0	42	slt $s1, $s2, $s3
j	J	2	2500					j 10000 (v. Paragrafo 3.8)
jr	R	0	9	0	0	0	8	jr $t1
jal	J	3	2500					jal 10000 (v. Paragrafo 3.8)
Dimensione del campo		6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Tutte le istruzioni MIPS sono lunghe 32 bit
Formato-R	R	op	rs	rt	rd	shamt	funct	Formato istruzioni aritmetiche
Formato-I	I	op	rs	rt		address		Formato istruzioni di trasferimento dati e istruzioni di salto condizionato

Figura 3.14 L'architettura MIPS mostrata fino al Paragrafo 3.6. Le parti evidenziate mostrano le strutture MIPS introdotte nel Paragrafo 3.6. Il formato-J, usato per le istruzioni di jump e jump-and-link, sarà spiegato nel Paragrafo 3.8, dove verrà anche spiegato perché mettere 2500 nel campo indirizzo delle istruzioni beq e bne equivale a scrivere 10000 in assembler.

3.7 Oltre i numeri

I calcolatori furono inventati per macinare numeri, ma non appena diventarono commercialmente disponibili furono utilizzati anche per l'elaborazione di testi. Oggi la maggior parte dei calcolatori usa 8 bit (1 byte) per rappresentare i caratteri e la codifica ASCII (da American Standard Code for Information Interchange) è di gran lunga la più utilizzata. La figura 3.15 riassume la codifica ASCII.

Poiché con un'opportuna sequenza di istruzioni di load e store si può estrarre un byte da una parola, le istruzioni load word e store word potrebbero essere usate per trasferire i byte. L'elaborazione dei testi è però talmente diffusa che il MIPS offre delle istruzioni apposite per trasferire i byte: load byte (`lb`) prende un byte dalla memoria, mettendolo negli 8 bit meno significativi di un registro (quelli a destra), mentre store byte (`sb`) prende il byte corrispondente agli 8 bit meno significativi di un registro e lo mette in memoria. È quindi possibile copiare un byte con la sequenza di istruzioni

```
lb    $t0, 0($sp)  # legge un byte dalla sorgente
sb    $t0, 0($gp)  # scrive un byte nella destinazione
```

Normalmente i caratteri sono raggruppati in stringhe, composte da un numero variabile di caratteri. Ci sono tre modi per rappresentare una stringa: (1) il primo elemento della stringa ne definisce la lunghezza, (2) una variabile di appoggio contiene la lunghezza della stringa (come in una struttura) oppure (3) l'ultimo elemento della stringa è marcato da un carattere particolare di fine stringa. Il C utilizza il terzo modo, terminando le stringhe con un byte che ha valore 0 (il codice ASCII

Codice ASCII	Carattere	Codice ASCII	Carattere	Codice ASCII	Carattere	Codice ASCII	Carattere	Codice ASCII	Carattere	Codice ASCII	Carattere
32	spazio	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Figura 3.15 La codifica ASCII dei caratteri. Si noti che le lettere maiuscole e minuscole differiscono esattamente di 32; ciò consente di utilizzare una scorciatoia per convertire le minuscole in maiuscole e viceversa. I valori non mostrati sono quelli per la formattazione dei caratteri: il 9, ad esempio, rappresenta il carattere di tabulazione, mentre il 13 è il carattere che permette di andare a capo. Altri codici utili sono l'8, per la cancellazione di un carattere, e lo 0, che rappresenta il carattere Null, usato dal linguaggio di programmazione C per identificare la fine di una stringa.

Null); in C, quindi, la stringa "Cal" è rappresentata da 4 byte, i cui valori decimali sono rispettivamente 67, 97, 108, 0.

Esempio

Compilare una procedura per copiare una stringa, mostrando l'utilizzo delle stringhe da parte del C

La procedura `strcpy` copia la stringa `y` nella stringa `x`, usando il byte Null come carattere di fine stringa:

```
void strcpy (char x[], char y[])
{
    int i;
    while ((x[i] = y[i]) != 0) /* copia e test del byte */
        i = i + 1;
}
```

Qual è il codice MIPS corrispondente?

Soluzione

Il segmento di codice MIPS viene mostrato nel seguito. Si assume nuovamente la possibilità di sommare o sottrarre costanti come 1 o 4, come si vedrà nel Paragrafo 3.8. Si assume inoltre che gli indirizzi di base dei vettori `x` e `y` si trovino rispettivamente in `$a0` e `$a1`, mentre `i` è in `$s0`. `strcpy` aggiorna lo stack pointer e poi salva il registro `$s0` nello stack:

```
strcpy:
    sub  $sp, $sp, 4     # aggiorna lo stack per l'inserzione
                         # di un elemento
    sw   $s0, 4($sp)     # salva $s0
```

Per inizializzare `i` a 0, l'istruzione successiva pone `$s0` a 0 sommando 0 a 0 e mettendo la somma in `$s0`:

```
    add  $s0, $zero, $zero   # i = 0 + 0
```

Questo è l'inizio del ciclo. L'indirizzo di `y[i]` è formato sommando `i` a `y[]`:

```
L1: add  $t1, $a1, $s0   # indirizzo di y[i] in $t1
```

Si noti che non è necessario moltiplicare `i` per quattro, dato che `y` è un vettore di *byte* e non di parole come nei casi precedenti.
Per caricare il carattere presente in `y[i]` si usa l'istruzione load byte, che lo mette in `$t2`:

```
    lb   $t2, 0($t1)     # $t2 = y[i]
```

Con un calcolo analogo si mette l'indirizzo di `x[i]` in `$t3` ed il carattere che si trova in `$t2` nella locazione di memoria così individuata.

```
    add  $t3, $a0, $s0   # indirizzo di x[i] in $t3
    sb   $t2, 0($t3)     # x[i] = y[i]
```

Ora si può incrementare `i` e tornare all'inizio del ciclo se il carattere è diverso da 0, cioè se non è il carattere di fine stringa.

```
    add  $s0, $s0, 1     # i = i + 1
    bne  $t2, $zero, L1  # se y[i] != 0 vai a L1
```

Se non si torna indietro significa che la stringa è terminata: vengono ripristinati i valori di $s0 e dello stack pointer, quindi si esce dalla procedura.

```
lw   $s0, 4($sp)     # y[i] == 0: fine della stringa;
                     # rispristina $s0
add  $sp, $sp, 4     # aggiorna lo stack eliminando
                     # un elemento
jr   $ra             # ritorno
```

Solitamente in C le copie di stringhe vengono fatte tramite i puntatori, per evitare le operazioni sull'indice i. Si veda il Paragrafo 3.11 per il confronto fra vettori e puntatori.

Dal momento che la procedura strcpy descritta sopra è una procedura di tipo foglia (cioè non chiama altre procedure), il compilatore può allocare i in un registro temporaneo, evitando di salvare e ripristinare $s0. Invece di pensare ai registri $t soltanto come a registri temporanei, è meglio considerarli come registri che le procedure possono usare a loro piacimento; quando il compilatore trova una procedura foglia, prima di usare i registri di tipo $s esaurisce tutti i registri temporanei.

Approfondimento: Esiste una codifica universale per i caratteri di tutte le lingue parlate dall'uomo, chiamata *Unicode*, che utilizza 16 bit per ciascun carattere. Il linguaggio Java, ad esempio, usa Unicode. Il set di istruzioni completo del MIPS ha delle istruzioni esplicite per il load e lo store di quantità su 16 bit, chiamate *halfword* (cioè mezze parole). Questo libro non considera le halfword per mantenere il set di istruzioni il più comprensibile possibile, ma il Paragrafo A.10 riporta l'intero set di istruzioni.

Inoltre il software MIPS cerca di mantenere lo stack allineato agli indirizzi delle parole, permettendo al programma di utilizzare sempre le istruzioni lw e sw (che necessitano di tale allineamento) per accedere allo stack. Questa convenzione significa che una variabile di tipo char nello stack viene allocata in 4 byte, anche se ne sarebbe sufficiente 1. Variabili di tipo stringa e vettori di caratteri invece *compattano 4 byte* in ogni parola.

3.8 Altri modi di indirizzamento del MIPS

I progettisti dell'architettura MIPS hanno previsto altri due modi per accedere ai dati: il primo è pensato per rendere veloce l'accesso a costanti di piccole dimensioni, il secondo per rendere i salti più efficienti.

Gli operandi costanti o immediati

I programmi usano molto spesso delle costanti per eseguire le operazioni richieste: ad esempio per incrementare un indice allo scopo di accedere all'elemento successivo di un vettore, per contare il numero di iterazioni in un ciclo o per aggiornare lo stack nel caso di procedure annidate. Per citare due esempi, nel compilatore C gcc il 52% delle operazioni aritmetiche contiene una costante ed il programma di simulazione spice ha il 69% di operazioni di questo tipo.

Usando solo le istruzioni che appaiono in figura 3.14, sarebbe necessario caricare la costante in memoria prima di poterla utilizzare (le costanti dovrebbero essere inserite in memoria durante il caricamento del programma). Ad esempio per sommare la costante 4 al registro $sp si potrebbe scrivere il codice


```
lw   $t0, IndCostante4($zero)  # $t0 = costante 4
add  $sp, $sp, $t0             # $sp = $sp + $t0 ($t0 == 4)
```

assumendo che IndCostante4 sia l'indirizzo di memoria in cui si trova la costante 4.

Un modo alternativo che evita l'accesso in memoria consiste nell'offrire delle particolari versioni delle istruzioni aritmetiche in cui uno degli operandi è una costante, con la novità rappresentata dal fatto che la costante è contenuta dentro l'*istruzione* stessa. Secondo il principio di usare strutture regolari, il formato di queste istruzioni è lo stesso di quelle di trasferimento dati e dei salti condizionati. Infatti la I del formato-I sta proprio per *immediato*, il nome dato tradizionalmente a questo tipo di operandi. Il campo MIPS contenente la costante è lungo 16 bit.

Esempio

Tradurre le costanti assembler in codice macchina

L'istruzione di somma in cui uno degli operandi è una costante si chiama *add immediate*, ossia addi. Per sommare 4 al registro $sp si scrive semplicemente

```
addi   $sp, $sp, 4    # $sp = $sp + 4
```

Il valore del campo op dell'istruzione addi è 8. Si provi a scrivere la parte restante del codice macchina MIPS relativo a questa istruzione.

Soluzione

All'istruzione corrisponde il seguente codice macchina, scritto in formato decimale (la figura 3.13 mostra che il numero corrispondente al registro $sp è 29):

op	rs	rt	immediato
8	29	29	4

In binario si ha quindi

001000	11101	11101	0000 0000 0000 0100

Gli operandi immediati, o costanti, sono molto utilizzati anche nelle operazioni di confronto. Dato che il registro $zero contiene sempre 0, si può confrontare facilmente una variabile con questo valore, ma per eseguire confronti con altri valori esiste una versione di tipo immediato dell'istruzione set on less. Ad esempio, per controllare se il registro $s2 è minore di 10, si può scrivere semplicemente

```
slti  $t0, $s2, 10 # $t0 = 1 se $s2 < 10
```

La versione in linguaggio macchina di lui $t0, 255 # $t0 è il registro 8

001111	00000	01000	0000 0000 1111 1111

Contenuto del registro $t0 dopo l'esecuzione di lui $t0, 255:

0000 0000 1111 1111	0000 0000 0000 0000

Figura 3.16 L'effetto dell'istruzione lui. L'istruzione lui trasferisce il campo da 16 bit relativo alla costante nei 16 bit più significativi del registro, riempiendo i restanti 16 con degli zeri. Come si vedrà nel Capitolo 4, questa istruzione ha lo stesso effetto di moltiplicare la costante per 2^{16} prima di caricarla nel registro.

Il modo di indirizzamento immediato consente di illustrare l'ultimo dei principi fondamentali per il progetto dell'hardware, menzionato una prima volta nel Capitolo 2:

Principio di progetto numero 4: rendere veloce l'evento più frequente.

Gli operandi di tipo costante sono molto frequenti e il fatto che le costanti siano parte delle istruzioni aritmetiche rende la loro esecuzione più veloce rispetto al caso in cui le costanti si trovino in memoria.

Benché le costanti solitamente siano di piccole dimensioni, rappresentabili su 16 bit, talvolta possono essere più grandi. Il set di istruzioni MIPS comprende l'istruzione *load upper immediate* (`lui`), che permette di caricare i 16 bit più significativi di una costante nella parte alta di un registro, consentendo ad un'istruzione successiva di specificare la parte bassa della costante. La figura 3.16 mostra il funzionamento dell'istruzione `lui`.

Caricare una costante da 32 bit

Esempio

Qual è il codice assembler MIPS per caricare questa costante da 32 bit nel registro `$s0`?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Soluzione

Per prima cosa con un'istruzione `lui` si caricano i 16 bit più significativi, il cui valore in decimale è 61:

```
lui $s0, 61 # 61 in decimale = 0000 0000 0011 1101 in
binario
```

A questo punto il registro `$s0` contiene

```
0000 0000 0011 1101 0000 0000 0000 0000
```

Il passo successivo consiste nel sommare i 16 bit meno significativi, il cui valore in decimale è 2304:

```
addi $s0, $s0, 2304 # 2304 in decimale = 0000 1001 0000 0000
                    # in binario
```

Il valore finale del registro `$s0` è quindi quello desiderato:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Interfaccia Hardware Software

Compilatori ed assemblatori devono dividere in blocchi le costanti molto grandi e successivamente riassemblarle in un registro. Come ci si può aspettare, la dimensione limitata del campo riservato alle costanti può rappresentare un problema per gli indirizzi di memoria nelle operazioni di trasferimento dati, oltre che per le costanti all'interno delle istruzioni di tipo immediato. Se questo lavoro è a carico dell'assemblatore, come nel caso del software MIPS, allora quest'ultimo deve avere un registro temporaneo

disponibile per costruire le costanti di grandi dimensioni: questa è una delle ragioni per cui il registro `$at` è riservato all'assemblatore.

In conclusione, la rappresentazione simbolica del linguaggio macchina MIPS non è più limitata dall'hardware ma da ciò che il progettista dell'assemblatore ha scelto di supportare (si veda il Paragrafo 3.9). Gli autori hanno scelto di rimanere legati all'hardware per spiegare l'architettura del calcolatore, rendendo esplicito l'eventuale utilizzo delle funzioni avanzate dell'assemblatore che non fanno parte dell'architettura dell'elaboratore.

Approfondimento: È necessario prestare molta attenzione quando si creano costanti da 32 bit. L'istruzione `addi` copia il bit più significativo dei 16 contenuti nel campo immediato dell'istruzione nei 16 bit più significativi di una parola (estensione del segno). Un'istruzione riportata nel paragrafo seguente, `ori`, che sta per *logical or immediate*, carica degli zeri nei 16 bit più significativi ed è quindi usata dall'assemblatore insieme alla `lui` per creare le costanti da 32 bit.

L'indirizzamento nei salti

Il modo di indirizzamento più semplice è quello dell'istruzione MIPS jump, che utilizza il formato-J, avente 6 bit per specificare il codice operativo ed i restanti bit per il campo indirizzo. Quindi l'istruzione

```
j     10000 # vai alla locazione 10000
```

è assemblata come segue:

2	10000
6 bit	26 bit

dove il valore codice operativo della jump è 2 e l'indirizzo del salto è 10000.

Contrariamente all'istruzione di jump, le istruzioni di salto condizionato devono specificare due operandi oltre all'indirizzo di salto. Quindi

```
bne   $s0, $s1, Esci     # vai a Esci se $s0 è diverso da $s1
```

è assemblata in questa istruzione, che ha solo 16 bit a disposizione per l'indirizzo di salto:

5	16	17	Esci
6 bit	5 bit	5 bit	16 bit

Se gli indirizzi del programma devono essere rappresentabili su 16 bit, significa che nessun programma può avere dimensioni superiori a 2^{16}, il che non è realistico per i programmi attuali. Una valida alternativa consiste nello specificare un registro il cui valore deve essere sommato all'indirizzo del salto; l'istruzione di salto condizionato dovrebbe quindi effettuare il seguente calcolo:

Program counter = Registro + Indirizzo di salto

La somma consentirebbe al programma di raggiungere una dimensione pari a 2^{32} e di essere ancora in grado di usare le istruzioni di salto condizionato, risolvendo il problema. La domanda è: quale registro usare?

La risposta deriva dal contesto in cui i salti condizionati vengono usati: essi si trovano solitamente nei cicli e nei costrutti *if*, quindi hanno la tendenza ad eseguire salti ad istruzioni vicine. Per fare un esempio, quasi la metà delle istruzioni di salto condizionato nei programmi gcc e spice hanno distanza inferiore alle 16 istruzioni. Dal momento che il program counter (PC) contiene l'indirizzo dell'istruzione corrente, si può saltare fino ad una distanza di $\pm 2^{15}$ istruzioni rispetto all'istruzione in esecuzione sommando all'indirizzo di salto il contenuto del registro PC. Quasi tutti i cicli ed i costrutti *if* hanno dimensione inferiore a 2^{16} parole, quindi il program counter è la scelta ideale.

Questo modo di indirizzamento per i salti condizionati si chiama *indirizzamento PC-relative* (cioè relativo al program counter). Come si vedrà nel Capitolo 5, dal punto di vista dell'hardware è conveniente incrementare il program counter per puntare all'istruzione successiva già nella fase preliminare dell'esecuzione, quindi l'indirizzo MIPS in realtà è relativo a quello dell'istruzione successiva a quella in esecuzione (PC+4) e non all'indirizzo corrente (PC).

Come molti dei calcolatori recenti, il MIPS utilizza l'indirizzamento PC-relative per tutti i tipi di salto condizionato, dato che la destinazione di queste istruzioni ha altissima probabilità di essere vicina all'istruzione di salto. Al contrario le istruzioni di jump-and-link chiamano procedure che non hanno motivo di essere vicine, quindi solitamente usano altri modi di indirizzamento. L'architettura MIPS usa il formato-J, che consente un maggiore spazio di indirizzamento, per le istruzioni di jump e jump-and-link.

Esempio

Calcolare l'offset di salti condizionati in linguaggio macchina

Il ciclo *while* riportato nel quarto esempio del Paragrafo 3.5 è stato compilato generando il seguente codice assembler MIPS:

```
Ciclo: add  $t1, $s3, $s3    # $t1 = 2 * i
       add  $t1, $t1, $t1    # $t1 = 4 * i
       add  $t1, $t1, $s5    # $t1 = indirizzo di salva[i]
       lw   $t0, 0($t1)      # $t0 = salva[i]
       bne  $t0, $s5, Esci   # vai a Esci se salva[i] è
                             # diverso da k
       add  $s3, $s3, $s4    # i = i + j
       j    Ciclo            # vai a Ciclo
Esci:
```

Se si assume che il ciclo sia memorizzato a partire dalla locazione 80000, qual è il codice macchina MIPS corrispondente?

Soluzione Le istruzioni assemblate e gli indirizzi corrispondenti sono riportati nel seguito:

80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	21	9	0	32
80012	35	9	8	0		
80016	5	8	21	8		
80020	0	19	20	19	0	32
80024	2	80000				
80028	...					
80012	35	9	8	0		

Si ricordi che le istruzioni MIPS indirizzano i byte, quindi gli indirizzi di parole consecutive differiscono di quattro unità, il numero di byte in una parola. L'istruzione `bne` della quinta riga somma 8 byte all'indirizzo dell'istruzione *successiva* (80020), specificando la destinazione del salto come distanza da tale istruzione (8) invece che come distanza dall'istruzione in esecuzione (12) o come indirizzo assoluto (80028). L'istruzione jump dell'ultima riga usa invece l'indirizzo completo corrispondente all'etichetta `Ciclo`.

Dato che tutte le istruzioni sono lunghe 4 byte, il MIPS estende la dimensione del salto facendo in modo che l'indirizzo PC-relative si riferisca al numero di parole che lo separano dalla destinazione, invece che al numero di byte. Quindi il campo di 16 bit, interpretato come indirizzo relativo in parole invece che in byte, può rappresentare salti quattro volte più ampi. Quindi nell'esempio sopra riportato il campo address dell'istruzione `bne` alla locazione 80016 avrebbe dovuto essere 2 anziché 8. L'indirizzamento relativo alle parole è la ragione per cui la codifica in linguaggio macchina delle istruzioni `beq` e `bne` riportata nelle figure 3.9 e 3.15 contiene 2500 invece di 10000 nel campo address (a differenza dell'assembler, in cui viene scritto 10000).

Interfaccia Hardware Software

La quasi totalità delle istruzioni di salto condizionato ha come destinazione una locazione di memoria vicina, ma occasionalmente può accadere di dover eseguire un salto ad una distanza tale da non poter essere rappresentata nei 16 bit del campo address. L'assembler risolve il problema nello stesso modo in cui risolve il problema di indirizzi o costanti molto grandi: inserisce un salto incondizionato (jump) al posto di quello condizionato (branch), invertendo la condizione originale in modo che controlli l'esecuzione o la non esecuzione del jump.

Esempio

Saltare ad indirizzi lontani

Dato il seguente salto condizionato, che controlla l'uguaglianza dei registri `$s0` e `$s1`

```
        beq     $s0, $s1, L1
```

lo si sostituisca con una coppia di istruzioni che permetta di saltare ad una distanza maggiore.

Soluzione

Il salto condizionato può essere sostituito da queste istruzioni:

```
        bne     $s0, $s1, L2
        j       L1
L2:
```

Approfondimento: Anche il campo di 26 bit all'interno delle istruzioni di jump contiene un indirizzo in parole: ciò significa che ha una capacità di indirizzamento pari ad un campo da 28 bit che indirizzi i byte. Dato che il program counter è lungo 32 bit, i restanti 4 devono provenire da qualche altra parte; l'istruzione di jump del MIPS sostituisce soltanto i 28 bit meno significativi del program counter, lasciando gli altri 4 inalterati. Il loader e il linker (vedere Paragrafo 3.9) devono evitare per quanto possibile di caricare il programma a cavallo dei "confini" che suddividono la memoria in blocchi da 256 MB (64 milioni di istruzioni), altrimenti il jump deve essere sostituito da un jump register preceduto dalle altre istruzioni necessarie a caricare in un registro l'indirizzo completo a 32 bit.

Riassunto dei modi di indirizzamento del MIPS

In questo paragrafo sono stati descritti due nuovi formati di indirizzamento; i diversi formati di indirizzamento di un set di istruzioni sono in generale chiamati modi di indirizzamento. I modi di indirizzamento del MIPS sono i seguenti:

1. *Register addressing* (indirizzamento tramite registro), in cui l'operando è un registro.
2. *Base* o *displacement addressing* (indirizzamento tramite base o tramite spiazzamento), in cui l'operando è in una locazione di memoria individuata dalla somma del contenuto di un registro e di una costante specificata nell'istruzione.
3. *Immediate addressing* (indirizzamento immediato), in cui l'operando è una costante specificata nell'istruzione.
4. *PC-relative addressing* (indirizzamento relativo al program counter), in cui l'indirizzo è la somma del contenuto del program counter e della costante contenuta nell'istruzione.
5. *Pseudodirect addressing* (indirizzamento pseudo-diretto), in cui l'indirizzo è ottenuto concatenando i 26 bit dell'istruzione con i 4 bit più significativi del program counter.

Si noti che un'operazione può usare più modi di indirizzamento; la somma ad esempio può avere un operando immediato (`addi`) o tutti gli operandi di tipo registro (`add`). La figura 3.17 mostra il modo in cui gli operandi sono identificati in ciascun modo di indirizzamento. Il Paragrafo 3.12 estenderà questa lista tenendo conto dei modi di indirizzamento utilizzati da altre architetture di calcolatori.

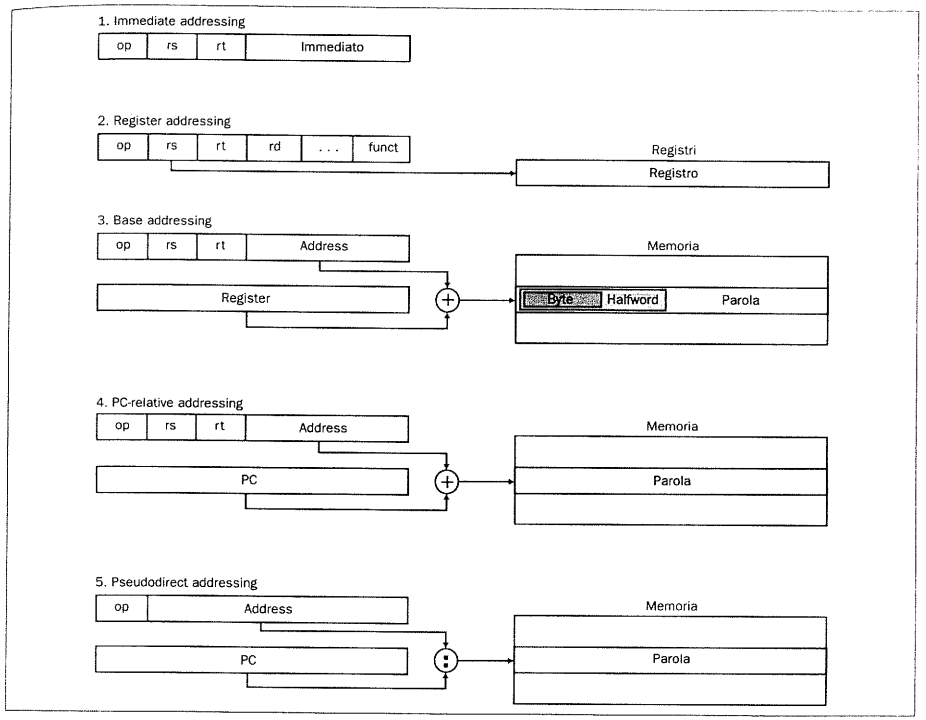


Figura 3.17 Illustrazione dei cinque modi di indirizzamento del MIPS. Gli operandi sono evidenziati. L'operando del modo 3 è in memoria, mentre quello del modo 2 è un registro. Si noti che differenti versioni delle istruzioni di load e store accedono ai byte, alle halfword ed alle parole. Nel modo 1 l'operando è costituito da 16 bit all'interno dell'istruzione stessa. I modi 4 e 5 sono utilizzati per indirizzare istruzioni in memoria; il modo 4 aggiunge un indirizzo su 16 bit al valore del program counter, mentre il modo 5 concatena un indirizzo su 26 bit ai 4 bit più significativi del PC.

Interfaccia Hardware Software

Nonostante l'architettura MIPS mostrata finora abbia gli indirizzi espressi su 32 bit, quasi tutti i microprocessori (compreso il MIPS) possiedono delle estensioni per l'indirizzamento a 64 bit (si veda l'Estensione Web I all'indirizzo *www.mkp.com/cod2e.htm*). Queste estensioni furono sviluppate in risposta all'esigenza di scrivere programmi di grandi dimensioni. Il processo di estensione del set di istruzioni permette di potenziare l'architettura garantendo allo stesso tempo la compatibilità del software verso le generazioni future dell'architettura.

op (31:26)								
31-29 \ 28-26	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	formato-R	Bltz/gez	jump	jump &link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm.
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	lh	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	sh	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op (31:26)=010000 (TLB), rs(25:21)								
25-24 \ 23-21	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op (31:26) = 000000 (formato-R), funct(5:0)								
5-3 \ 2-0	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	sll		srl	sra	sllv		srlv	srav
1(001)	jump reg.	jarl			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	nor
5(101)			set l.t.	sltu				
6(110)								
7(111)								

Figura 3.18 Codifica delle istruzioni MIPS. Questa notazione fornisce il valore del campo combinando i valori di riga e colonna. Ad esempio guardando nella parte alta della figura, l'istruzione `load word` si trova nella riga 4 (100_{due} per i bit 31-29 dell'istruzione) e nella colonna 3 (011_{due} per i bit 28-26 dell'istruzione), quindi il valore del campo op corrispondente (bit 31-26) è 100011_{due}. Le parole sottolineate indicano che il campo è definito altrove: ad esempio il `formato-R` in riga 0 e colonna 0 (op = 000000_{due}) viene definito nella parte bassa della figura. La posizione di `subtract` in riga 4 e colonna 2 in basso indica che il campo funct (bit 5-0) dell'istruzione vale 100010_{due} e che il campo op (bit 31-26) è 000000_{due}. Il valore `FlPt` in riga 2 e colonna 1 è definito nella figura 4.48 del Capitolo 4. `Bltz/gez` è il codice operativo di quattro istruzioni che si trovano nell'Appendice A: `bltz`, `bgez`, `bltzal` e `bgezal`. Le istruzioni colorate e indicate con il nome completo sono quelle descritte nel Capitolo 3, mentre quelle colorate e indicate con il codice mnemonico saranno descritte nel Capitolo 4. L'Appendice A copre tutte le istruzioni.

Decodificare il linguaggio macchina

In alcuni casi è utile ricostruire il linguaggio assembler a partire dal codice macchina, ad esempio per analizzare il core dump, ossia un file contenente una copia della memoria generato da un errore in un programma. La figura 3.18 mostra la codifica MIPS dei campi all'interno delle istruzioni macchina; tale figura può essere utilizzata per tradurre manualmente da codice macchina a codice assembler e viceversa.

Decodificare il linguaggio macchina

Esempio

Qual è il linguaggio assembler corrispondente alla seguente istruzione macchina?

```
(Bit:   31  28  26                          5    2  0)
        0000 0000 1010 1111 1000 0000 0010 0000
```

Soluzione

Il primo passo consiste nel guardare il campo op, per determinare il tipo di operazione. Facendo riferimento alla figura 3.18, quando i bit 31-29 valgono 000 e i bit 28-26 sono 000, l'istruzione è di formato-R. Riordinando l'istruzione binaria secondo i campi del formato-R, elencati nella figura 3.19, si ottiene

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

La parte in basso della figura 3.18 determina le operazioni di un'istruzione di formato-R; in questo caso i bit 5-3 valgono 100 ed i bit 2-0 sono 000, rappresentando un'istruzione `add`.

Si può decodificare il resto dell'istruzione guardando il contenuto dei campi. I valori decimali sono 5 per rs, 15 per rt, 16 per rd, mentre il campo shamt non è utilizzato. Dalla figura 3.13 si ricava che questi numeri rappresentano i registri `$a1`, `$t7` e `$s0`. L'istruzione assembler è quindi

```
add  $s0, $a1, $t7
```

La figura 3.20 mostra il linguaggio assembler MIPS descritto fino al Capitolo 3; le parti di codice non ancora presentate hanno principalmente a che fare con l'aritmetica e saranno esaminate nel prossimo capitolo.

3.9 Avviare un programma

Questo paragrafo descrive i quattro passi necessari a trasformare un programma C residente in un file su disco, in un programma in esecuzione su un calcolatore. La figura 3.21 mostra la gerarchia della trasformazione; alcuni calcolatori raggruppano questi passi per ridurre il tempo di traduzione, ma concettualmente tutti i programmi passano sempre attraverso quattro fasi, trattate nel seguito del paragrafo.

Nome	Esempio						Commenti
Dimensione del campo	6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Tutte le istruzioni MIPS sono lunghe 32 bit
formato-R	op	rs	rt	rd	shamt	funct	Formato delle istruzioni aritmetiche
formato-I	op	rs	rt	address / immediate			Formato delle istruzioni di trasferimento, di salto condizionato, immediate
formato-J	op	target address					Formato delle istruzioni di salto incondizionato

Figura 3.19 Il formato delle istruzioni MIPS introdotte nel Capitolo 3. Le parti evidenziate mostrano i formati introdotti in questo paragrafo.

Operandi MIPS

Nome	Esempio	Commenti
32 registri	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. Il registro $zero ha sempre valore 0. Il registro $at è riservato all'assemblatore per la gestione di costanti di grandi dimensioni.
2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte, quindi le parole consecutive differiscono di 4 unità. La memoria contiene le strutture dati, come i vettori, i registri riversati (spilled) e i registri salvati durante le chiamate a procedura.

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	add $s1, $s2, $s3	$s1 = $s2 + $s3	Tre operandi; dati nei registri
	sottrazione	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Tre operandi; dati nei registri
	somma imm.	addi $s1, $s2, 100	$s1 = $s2 + 100	Usata per sommare costanti
Trasferimento dati	load word	lw $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Parola dalla memoria al registro
	store word	sw $s1, 100($s2)	Memoria[$s2 + 100] = $s1	Parola dal registro alla memoria
	load byte	lb $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Byte dalla memoria al registro
	store byte	sb $s1, 100($s2)	Memoria[$s2 + 100]= $s1	Byte dal registro alla memoria
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Carica una costante nei 16 bit più significativi
Salto condizionato	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Controllo dell'uguaglianza ed eventuale salto; salto PC-relative
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Controllo della disuguaglianza ed eventuale salto; salto PC-relative
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Confronto del tipo "minore di"; usato con beq, bne
	set less than immediate	slti $s1, $s2, 100	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Confronto con una costante
Salto incondizionato	jump	j 2500	go to 10000	Salto ad un indirizzo specificato
	jump register	jr $ra	go to $ra	Usato nei costrutti *switch* e nell'uscita dalle procedure
	jump and link	jal 2500	$ra = PC + 4; go to 10000	Per le chiamate a procedura

Figura 3.20 Le istruzioni MIPS mostrate nel Capitolo 3. Le parti evidenziate si riferiscono ai Paragrafi 3.7 e 3.8.

Il compilatore

Il compilatore trasforma il programma C in un *programma in linguaggio assembler*, cioè in una forma simbolica di ciò che il calcolatore è in grado di capire. I programmi scritti in un linguaggio ad alto livello hanno un numero di righe di codice minore rispetto al linguaggio assembler, quindi la produttività dei programmatori è più elevata.

Nel 1975 molti sistemi operativi e molti assemblatori erano scritti in linguaggio assembler, dal momento che la capacità della memoria era ridotta ed i compilatori inefficienti. Oggi la capacità delle memorie è cresciuta di 16.000 volte, riducendo il problema delle dimensioni dei programmi, ed i compilatori ottimizzati sono in grado di produrre codice assembler buono quasi quanto quello dei programmatori assembler esperti e qualche volta migliore, nel caso di programmi molto grandi.

L'assemblatore

Come citato nel paragrafo precedente, dato che il linguaggio assembler rappresenta l'interfaccia verso il software di livello più alto, l'assemblatore può anche trattare

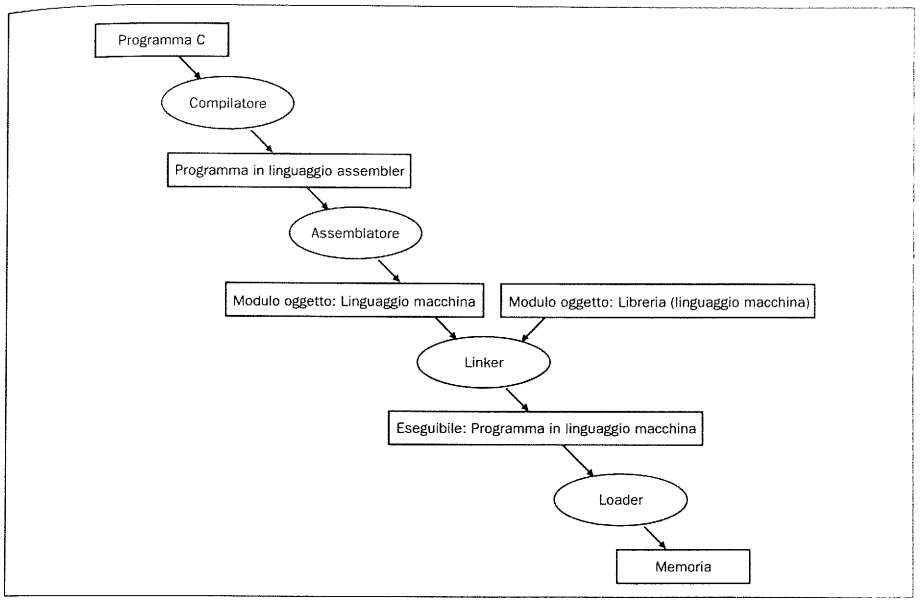


Figura 3.21 Gerarchia di trasformazione. Un programma scritto in linguaggio ad alto livello viene per prima cosa compilato in un linguaggio assembler e poi assemblato per ottenere un modulo oggetto in linguaggio macchina. Il linker combina uno o più moduli con le procedure contenute nelle librerie, per esplicitare tutti i riferimenti incrociati; il loader infine colloca il codice macchina nelle opportune locazioni di memoria, in modo che possa essere eseguito dal processore. Per accelerare il processo di trasformazione alcuni passi possono essere saltati o eseguiti insieme ad altri. Alcuni compilatori producono direttamente il codice macchina e alcuni sistemi usano dei linker-loader che eseguono insieme gli ultimi due passi. Per identificare il tipo di file, il sistema operativo Unix segue una convenzione per l'estensione dei file: i file sorgenti in C vengono chiamati `x.c`, i file assembler `x.s`, i file oggetto `x.o` ed il file eseguibile è solitamente `a.out`. Il sistema operativo MS-DOS usa le estensioni `.C`, `.ASM`, `.OBJ` e `.EXE` rispettivamente.

versioni modificate delle istruzioni in linguaggio macchina come se fossero istruzioni vere e proprie. Queste istruzioni, chiamate *pseudoistruzioni*, non devono essere implementate in hardware, ma la loro presenza nel linguaggio assembler semplifica le fasi di traduzione e di programmazione.

Ad esempio l'hardware MIPS si assicura che il registro `$zero` contenga sempre il valore 0; quindi il registro `$zero` fornisce il valore 0 ogni volta che viene usato ed il programmatore non può cambiare il valore di `$zero`. Il registro `$zero` è usato per creare l'istruzione in linguaggio assembler `move`, che copia il contenuto di un registro in un altro. L'assemblatore MIPS accetta questa istruzione anche se non fa parte dell'architettura MIPS:

```
move $t0, $t1          # il registro $t0 assume il valore
                       # del registro $t1
```

e la converte nel linguaggio macchina equivalente a

```
add  $t0, $zero, $t1   # il registro $t0 assume il valore 0 più
                       # quello del registro $t1
```

L'assemblatore MIPS converte l'istruzione `blt` (branch on less than) nelle due istruzioni `slt` e `bne` (vedere l'esempio nel Paragrafo 3.5); altri esempi comprendo-

no le istruzioni `bgt`, `bge` e `ble`. Inoltre converte i salti condizionati a locazioni di memoria distanti in un branch e una jump. Come citato precedentemente, l'assemblatore MIPS può persino permettere il caricamento in un registro di costanti da 32 bit, nonostante il limite di 16 bit delle istruzioni immediate.

Riassumendo, le pseudoistruzioni consentono all'assembler MIPS di avere un set di istruzioni più ricco di quello implementato via hardware; l'unico costo è rappresentato dal registro `$at`, riservato all'assemblatore. Quando si scrivono programmi in assembler l'utilizzo delle pseudoistruzioni rende il compito più semplice; per capire l'architettura MIPS e per ottenere le migliori prestazioni, comunque, è bene conoscere le reali istruzioni MIPS, riportate nelle figure 3.18 e 3.20.

Gli assemblatori inoltre accettano numeri espressi in più basi. Oltre alla base due e a quella decimale, solitamente accettano una base più concisa di quella binaria, che però può essere facilmente convertita nella corrispondente sequenza di bit: l'assemblatore MIPS usa la base 16, chiamata *esadecimale*; il pedice "esa" indica un numero esadecimale. Le cifre esadecimali comprendono i numeri da 0 a 9 e le lettere da *a* a *f* per le restanti 6 cifre; ad esempio la sequenza di bit riportata nell'ultimo esempio del Paragrafo 3.8 è qui mostrata sia in formato binario che in quello esadecimale:

$$0000\ 0000\ 1010\ 1111\ 1000\ 0000\ 0010\ 0000_{due} = 00af\ 8020_{esa}$$

Tutte le caratteristiche dell'assemblatore appena illustrate sono molto utili, ma il suo compito principale rimane quello di generare il linguaggio macchina. L'assemblatore converte un programma assembler in un *file oggetto*, che è una combinazione di istruzioni in *linguaggio macchina*, di dati e di informazioni necessarie a collocare le istruzioni in memoria nella posizione opportuna.

Per produrre la versione binaria di ciascuna istruzione del linguaggio assembler, l'assemblatore deve determinare gli indirizzi corrispondenti a tutte le etichette: esso tiene traccia di tutte le etichette usate nei salti e nei trasferimenti di dati in una tabella, detta *symbol table* (tabella dei simboli); come ci si può aspettare, la tabella contiene coppie del tipo simbolo – indirizzo.

I file oggetto dei sistemi Unix tipicamente contengono sei frammenti distinti:

- L'intestazione del file oggetto (*object file header*) descrive la dimensione e la posizione degli altri frammenti del file oggetto.
- Il segmento di testo (*text segment*) contiene il codice in linguaggio macchina.
- Il segmento di dati (*data segment*) contiene tutti i dati che fanno parte del programma, siano essi *dati statici*, allocati durante tutto il programma, o *dati dinamici*, che aumentano o diminuiscono a seconda delle esigenze del programma.
- Le informazioni di rilocazione (*relocation information*) identificano le istruzioni ed i dati che, quando il programma è posto in memoria, dipendono da indirizzi assoluti.
- La *symbol table* contiene le etichette che non sono ancora definite, ad esempio quelle che fanno riferimento a moduli esterni.
- Le informazioni per il debugger (*debugging information*) contengono una descrizione concisa di come sono stati compilati i moduli, in modo che il debugger possa associare le istruzioni macchina al codice sorgente C per rendere leggibili le strutture dati.

La trattazione relativa al linker mostrerà come vengano accorpate procedure già assemblate, ad esempio quelle contenute nelle librerie.

Il linker

Tutto ciò che è stato visto finora può suggerire che un piccolo cambiamento su un'unica linea di codice richiederebbe di ricompilare e riassemblare l'intero programma. Tradurre tutto dal principio è un terribile spreco di risorse computazionali, particolarmente pesante per le librerie standard, dato che i programmatori ricompilerebbero e riassemblerebbero procedure che per definizione non cambiano praticamente mai. Una valida alternativa consiste nel compilare ed assemblare ciascuna procedura indipendentemente dalle altre, in modo che un cambiamento su un'unica linea di codice renda necessario ricompilare e riassemblare un'unica procedura. Tale alternativa richiede un nuovo programma di sistema, chiamato *link editor* (programma che gestisce i collegamenti) o *linker*, che prende tutti i programmi in codice macchina che sono stati assemblati in modo indipendente e li "cuce" insieme.

Il linker esegue tre passi:

1. Mette in memoria in modo simbolico il codice ed i moduli dei dati.
2. Determina gli indirizzi dei dati e delle etichette che compaiono nelle istruzioni.
3. Mette insieme i riferimenti interni ed esterni.

Il linker usa le informazioni di rilocazione e la symbol table all'interno di ciascun modulo per risolvere tutte le etichette non definite. Questi riferimenti si trovano nelle istruzioni di salto condizionato e incondizionato e negli indirizzi dei dati, quindi il compito del programma è più o meno quello di un editor: trovare gli indirizzi vecchi e sostituirli con quelli nuovi. Questa attività di "editing" (correzione) è all'origine del nome "link editor", abbreviato in linker. Il motivo dell'esistenza del linker è che è molto più veloce "cucire" il codice piuttosto che ricompilarlo e riassemblarlo.

Dopo che tutti i riferimenti esterni sono stati risolti, il linker determina le locazioni di memoria che ciascun modulo dovrà occupare. La figura 3.22 mostra la convenzione MIPS per l'allocazione di programmi e dati in memoria. Dato che tutti i file sono assemblati in modo indipendente, l'assemblatore non può sapere la posizione relativa delle istruzioni e dei dati di un modulo rispetto agli altri; quando il linker mette un modulo in memoria tutti i riferimenti *assoluti*, cioè gli indirizzi di memoria non relativi ai registri, devono essere *rilocati* per riflettere la posizione reale.

Il linker produce un *file eseguibile* che può essere eseguito su un calcolatore. Di norma questo file ha lo stesso formato di un file oggetto, ma non contiene riferimenti non risolti, informazioni di rilocazione, tabella dei simboli e informazioni per il debugger. È possibile fare questo passo in modo parziale, come nel caso delle librerie, che contengono ancora indirizzi non risolti e quindi in realtà risultano essere dei file oggetto.

Esempio

Eseguire il link di due file oggetto

Eseguire il link dei due file oggetto riportati nel seguito, mostrando gli indirizzi aggiornati delle prime (poche) istruzioni del file eseguibile. Le istruzioni sono mostrate in linguaggio assembler solo per rendere l'esempio più leggibile: in realtà le istruzioni sono dei numeri.

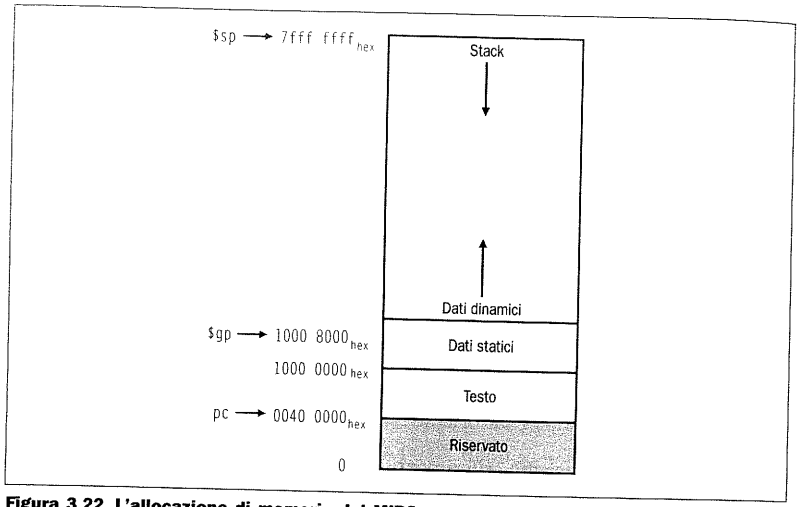


Figura 3.22 L'allocazione di memoria del MIPS per programmi e dati. Partendo dall'alto, lo stack pointer è inizializzato a `7fff ffff`$_{esa}$ e scende verso il segmento dei dati. All'altro capo, il codice del programma ("testo") parte dalla locazione `0400 0000`$_{esa}$. I dati statici iniziano da `1000 0000`$_{esa}$; seguono i dati dinamici, allocati in C dalla funzione `malloc`, che crescono verso lo stack. Il global pointer `$gp` semplifica l'accesso ai dati; è inizializzato a `1000 8000`$_{esa}$ in modo da consentire l'accesso nell'intervallo da `1000 0000`$_{esa}$ a `1000 ffff`$_{esa}$ utilizzando un offset (positivo o negativo) di 16 bit a partire dal valore di `$gp` (si vedano gli indirizzi in complemento a due nel Capitolo 4).

Si noti che all'interno dei file oggetto sono evidenziati gli indirizzi ed i simboli che devono essere riaggiornati nel processo di link: le istruzioni che fanno riferimento agli indirizzi delle procedure `A` e `B` e quelle che fanno riferimento agli indirizzi delle parole di dato `X` e `Y`.

Intestazione del file oggetto			
	Nome	Procedura A	
	Dimensione del testo	100_{esa}	
	Dimensione dei dati	20_{esa}	
Segmento di testo	Indirizzo	Istruzione	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	...	...	
Segmento di dati	0	(X)	
	...	...	
Informazioni di rilocazione	Indirizzo	Tipo di istruzione	Dipendenza
	0	`lw`	X
	4	`jal`	B
Tabella dei simboli	Etichetta	Indirizzo	
	Y	-	
	B	-	

Intestazione del file oggetto			
	Nome	Procedura B	
	Dimensione del testo	200_{esa}	
	Dimensione dei dati	30_{esa}	
Segmento di testo	Indirizzo	Istruzione	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	...	...	
Segmento di dati	0	(Y)	
...	...		
Informazioni di riallocazione	Indirizzo	Tipo di istruzione	Dipendenza
	0	`lw`	
	4	`jal`	A
Tabella dei simboli	Etichetta	Indirizzo	
	Y	-	
	A	-	

Soluzione

La procedura `A` deve trovare l'indirizzo della variabile etichettata `X` per metterlo nell'istruzione load e l'indirizzo della procedura `B` per inserirlo nell'istruzione `jal`. La procedura `B` ha bisogno dell'indirizzo della variabile etichettata `Y` per metterlo nell'istruzione store e dell'indirizzo della procedura `A` per la sua istruzione `jal`.

Intestazione del file eseguibile		
	Dimensione del testo	300esa
	Dimensione dei dati	50esa
Segmento di testo	Indirizzo	Istruzione
	0040 0000$_{esa}$	`lw $a0, 8020 (($gp)`
	0040 0004$_{esa}$	`jal 40 0100`
	...	...
	0040 0100$_{esa}$	`sw $a1, 8020 (($gp)`
	0040 0104$_{esa}$	`jal 40 0000`
	...	...
Segmento di dati	Indirizzo	
	1000 0000$_{esa}$	(X)
	...	...
	1000 0020$_{esa}$	(Y)
	...	...

Dalla figura 3.22 si ricava che il segmento di testo inizia all'indirizzo `0400 0000`$_{esa}$ e che il segmento dati inizia da `1000 0000`$_{esa}$. Il testo della procedura `A` è posto nel primo dei due indirizzi ed i suoi dati nel secondo. L'intestazione del file oggetto relativo alla procedura `A` indica che il suo testo è lungo 100_{esa} byte e che i suoi dati ne occupano 20_{esa}, quindi l'indirizzo di partenza del testo della procedura `B` è `0040 0100`$_{esa}$ ed i suoi dati partono da `1000 0020`$_{esa}$.

A questo punto il linker aggiorna il campo address delle istruzioni, usando il campo che determina il tipo dell'istruzione per riconoscerne il formato. In questo caso sono presenti due tipi:

1. Le istruzioni `jal` sono semplici, dal momento che usano l'indirizzamento pseudo-diretto. Il campo address dell'istruzione `jal` che si trova in $0040\ 0004_{esa}$ assume il valore $40\ 0100_{esa}$ (l'indirizzo della procedura `B`) e la `jal` all'indirizzo $0040\ 0104_{esa}$ ha come campo address il valore $40\ 0000_{esa}$ (l'indirizzo della procedura `A`).
2. Gli indirizzi delle istruzioni di trasferimento dati sono un po' più complessi da ricavare, dal momento che dipendono dal registro di base. In questo esempio è il global pointer ad essere usato come indirizzo di base; la figura 3.22 mostra che `$gp` è inizializzato a $1000\ 8000_{esa}$. Per ottenere l'indirizzo $1000\ 0000_{esa}$ (quello della parola `X`) bisogna mettere 8000_{esa} nel campo address dell'istruzione `lw` che si trova all'indirizzo $0040\ 0000_{esa}$. Il Capitolo 4 spiega l'aritmetica in complemento a due su 16 bit, mostrando il motivo per cui aggiungendo 8000_{esa} al campo address si ottiene proprio l'indirizzo $1000\ 0000_{esa}$. In modo analogo occorre mettere il valore 8020_{esa} nel campo address dell'istruzione `sw` che si trova all'indirizzo $0040\ 0100_{esa}$ per ottenere l'indirizzo $1000\ 0020_{esa}$ (quello della parola `Y`).

Il loader

Una volta che il file eseguibile è memorizzato su disco, il sistema operativo può leggerlo per effettuare il trasferimento in memoria e per avviarne l'esecuzione. Nei sistemi Unix vengono eseguiti i seguenti passi:

1. Lettura dell'intestazione del file eseguibile per determinare la lunghezza dei segmenti di testo e di dati.
2. Creazione di uno spazio di indirizzamento sufficiente a contenere testo e dati.
3. Copia delle istruzioni e dei dati dal file eseguibile in memoria.
4. Copia nello stack degli eventuali parametri passati al programma principale.
5. Inizializzazione dei registri dell'elaboratore e posizionamento dello stack pointer alla prima locazione libera.
6. Salto ad una procedura di inizializzazione (start-up) che copia i parametri nei registri argomento e chiama la procedura principale del programma. Quando la procedura principale restituisce il controllo, la procedura di start-up termina il programma con una chiamata di sistema `exit`.

I Paragrafi A.3 e A.4 dell'Appendice A descrivono in maggior dettaglio linker e loader.

3.10 Un esempio riassuntivo

Mostrare sempre piccoli frammenti di codice assembler alla volta rischia di non dare l'idea del modo un cui è fatto un programma assembler completo. Per ovviare a ciò, in questo paragrafo sarà presentato il codice MIPS di due procedure scritte in C, una per scambiare fra loro due elementi consecutivi di un vettore ed una per ordinarli.


```
scambia (int v[], int k)
{
 int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

Figura 3.23 Una procedura C che scambia fra loro due locazioni di memoria. Questa procedura sarà utilizzata nell'esempio di ordinamento successivo. L'estensione Web III all'indirizzo *www.mkp.com/cod2e.htm* mostra le versioni C e Pascal affiancate di questa procedura.

La procedura `scambia`

Il codice C della procedura `scambia`, che scambia semplicemente fra loro due locazioni di memoria, è mostrato in figura 3.23. Per tradurre dal C all'assembler in generale si seguono questi passi:

1. Si allocano i registri per le variabili del programma.
2. Si produce il codice relativo al corpo della procedura.
3. Si preserva il valore dei registri dalle modifiche che possono avvenire nel corso della procedura.

Nel seguito le tre parti della procedura `scambia` vengono prima descritte separatamente, poi vengono assemblate.

L'allocazione dei registri per la procedura `scambia`

Come descritto all'inizio del Paragrafo 3.6, la convenzione del MIPS per quanto riguarda il passaggio dei parametri prevede l'utilizzo dei registri `$a0`, `$a1`, `$a2` e `$a3`; dal momento che la procedura di scambio ha solo due parametri, `v` e `k`, questi si troveranno nei registri `$a0` e `$a1`. L'unica altra variabile del programma è `temp`, che viene associata al registro `$t0` dato che `scambia` è una procedura di tipo foglia (vedere il Paragrafo 3.7). L'allocazione dei registri corrisponde alla dichiarazione delle variabili nella prima parte della procedura di figura 3.23.

Il corpo della procedura `scambia`

Le restanti linee di codice C della procedura `scambia` sono

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Ricordando che gli indirizzi di memoria delle istruzioni MIPS fanno riferimento ai byte e che quindi le parole consecutive differiscono di 4 byte, occorre moltiplicare l'indice `k` per 4 prima di sommarlo all'indirizzo del vettore. *Dimenticare che gli indirizzi di due parole consecutive differiscono di 4 invece che di 1 è un errore molto comune nella programmazione assembler.* Il primo passo in questo caso consiste quindi nell'ottenere l'indirizzo di `v[k]` moltiplicando `k` per 4:

```
add   $t1, $a1, $a1   # $t1 = k * 2
add   $t1, $t1, $t1   # $t1 = k * 4
add   $t1, $a0, $t1   # $t1 = v + (k * 4): nel registro $t1 c'è
                      # l'indirizzo di v[k]
```

Ora si può caricare `v[k]` usando `$t1` e `v[k+1]` aggiungendo 4 a `$t1`:

```
lw    $t0, 0($t1)     # $t0 (temp) = v[k]
lw    $t2, 4($t1)     # $t2 = v[k+1]: l'elemento successivo di
v
```

Il passo successivo consiste nel memorizzare $t0 e $t2 scambiando gli indirizzi:

```
sw    $t2, 0($t1)     # v[k] = $t2
sw    $t0, 4($t1)     # v[k+1] = $t0 (temp)
```

Dopo aver allocato i registri e scritto il corpo della procedura, non resta che aggiungere il codice per salvare il contenuto dei registri utilizzati dalla procedura `scambia`. Dato che non vengono usati registri di tipo `$s`, questa parte non è necessaria.

La procedura `scambia` completa

La procedura completa, che comprende l'etichetta della procedura e l'istruzione di ritorno, è mostrata in figura 3.24; per rendere il programma di facile lettura, a ciascun blocco di codice è associata la finalità.

La procedura `ordina`

Per permettere al lettore di apprezzare la rigorosità della programmazione in linguaggio assembler viene descritto un altro esempio, di complessità maggiore: viene creata una procedura, che chiama la procedura `scambia`, il cui scopo è ordinare un vettore di numeri interi; la figura 3.25 mostra la versione in C del programma. Anche in questo caso la procedura sarà costruita in passi successivi, terminando con il codice assembler completo.

L'allocazione dei registri per la procedura `ordina`

I due parametri della procedura `ordina`, `v` ed `n`, sono nei registri argomento `$a0` e `$a1`, mentre i registri `$s0` e `$s1` sono associati rispettivamente alle variabili `i` e `j`.

Corpo della procedura

```
scambia: add  $t1, $a1, $a1 # $t1 = k * 2
         add  $t1, $t1, $t1 # $t1 = k * 4
         add  $t1, $a0, $t1 # $t1 = v + (k * 4)
                            # nel registro $t1 c'è l'indirizzo di v[k]
         lw   $t0, 0($t1)   # $t0 (temp) = v[k]
         lw   $t2, 4($t1)   # $t2 = v[k+1]
                            # si riferisce all'elemento successivo di v
         sw   $t2, 0($t1)   # v[k] = $t2
         sw   $t0, 4($t1)   # v[k+1] = $t0 (temp)
```

Ritorno dalla procedura

```
         jr   $ra           # ritorno alla procedura chiamante
```

Figura 3.24 Il codice assembler MIPS della procedura `scambia` di figura 3.23.


```
ordina (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i = i + 1) {
    for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j = j - 1) {scambia (v,
j);
    }
  }
}
```

Figura 3.25 Una procedura C che esegue l'ordinamento di un vettore `v`. Nel caso di scarsa familiarità con il linguaggio C, le tre parti del costrutto *for* più esterno rappresentano rispettivamente l'inizializzazione precedente alla prima iterazione (`i = 0`), il test di fine ciclo (`i < n`) e l'operazione da svolgere al termine di ogni iterazione (`i = i + 1`). L'estensione Web II all'indirizzo *www.mkp.com/cod2e.htm* mostra le versioni C e Pascal affiancate di questa procedura.

Il corpo della procedura `ordina`

Il corpo della procedura consiste in due cicli *for* annidati e nella chiamata alla procedura `scambia` con l'opportuno passaggio dei parametri. In questo esempio il codice viene sviluppato dall'esterno all'interno.

Il primo passaggio consiste nella traduzione del ciclo *for* più esterno:

```
for (i = 0; i < n; i = i + 1) {
```

In C il ciclo *for* è composto da tre parti: inizializzazione, test di fine ciclo ed incremento per ogni iterazione. Basta una sola istruzione per inizializzare `i` a 0, traducendo la prima parte del costrutto *for*:

```
        move    $s0, $zero        # i = 0
```

Si ricordi che `move` è una pseudoistruzione fornita dall'assemblatore a vantaggio del programmatore assembler (vedere Paragrafo 3.9). Un'altra istruzione è sufficiente ad incrementare `i`, traducendo l'ultima parte del costrutto *for*:

```
        addi    $s0, $s0, 1       # i = i + 1
```

Il ciclo termina se la condizione `i < n` *non* è vera o in altre parole se `i >= n`. L'istruzione set on less than pone il registro `$t1` a uno se `$s0 < $a1`, a zero in caso contrario; dal momento che si vuol controllare la condizione `$s0 >= $a1`, il salto viene eseguito se il registro `$t0` vale 0. Il test dunque viene tradotto in due istruzioni:

```
forltst: slt    $t0, $s0, $a1     # $t0 = 0 se $s0 >= $a1
                                  # (i >= n)
         beq    $t0, $zero, escil # vai a escil se $s0 >= $a1
                                  # (i >= n)
```

Come ultima istruzione del ciclo compare un salto incondizionato che ritorna all'inizio del ciclo stesso:

```
         j      forltst           # torna al test del ciclo più
                                  # esterno
escil:
```

Lo scheletro di codice del ciclo *for* più esterno è quindi

```
         move   $s0, $zero        # i = 0
forltst: slt    $t0, $s0, $a1     # $t0 = 0
                                  # se $s0 >= $a1 (i >= n)
```

```
        beq     $t0, $zero, esci1 # vai a esci1 se $s0 >= $a1
                                  # (i >= n)
                ...
                (corpo del ciclo for più esterno)
                ...
        addi    $s0, $s0, 1       # i = i + 1
        j       for1tst           # torna al test del ciclo più
                                  # esterno
esci1:
```

L'Esercizio 3.9 esplora la possibilità di una versione ottimizzata per questo tipo di ciclo.

Il ciclo *for* più interno ha la seguente espressione in linguaggio C:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j = j - 1) {
```

La parte di inizializzazione è nuovamente composta da una sola istruzione:

```
addi    $s1, $s0, -1          # j = i - 1
```

Anche il decremento di `j` alla fine del ciclo può essere tradotto in una singola istruzione:

```
addi    $s1, $s1, -1          # j = j - 1
```

Il test di fine ciclo è composto da due parti: si esce dal ciclo se almeno una delle condizioni non è verificata; la prima condizione è falsa, quindi provoca la fine del ciclo, se `j < 0`:

```
for2tst: slti $t0, $s1, 0        # $t0 = 1 se $s1 < 0 (j < 0)
         bne  $t0, $zero, esci2  # vai a esci2 se $s1 < 0 (j < 0)
```

L'ultima istruzione fa sì che venga saltata la seconda parte del test; se questo non si verifica, allora `j >= 0`.

La seconda condizione implica che il ciclo termini se `v[j] > v[j+1]` *non* è vera o in altre parole se `v[j] <= v[j+1]`. Per prima cosa bisogna ottenere l'indirizzo moltiplicando `j` per 4 (gli indirizzi sono sempre espressi in byte) e sommarlo all'indirizzo base di `v`:

```
add     $t1, $s1, $s1         # $t1 = j * 2
add     $t1, $t1, $t1         # $t1 = j * 4
add     $t2, $a0, $t1         # $t2 = v + (j * 4)
```

Ora è possibile caricare `v[j]`:

```
lw      $t3, 0($t2)           # $t3 = v[j]
```

Dal momento che è noto che il secondo elemento del vettore da considerare è l'elemento successivo, basta sommare 4 all'indirizzo contenuto in `$t2` per ottenere `v[j+1]`:

```
lw      $t4, 4($t2)           # $t4 = v[j+1]
```

Il test `v[j] <= v[j+1]` è analogo a `v[j+1] >= v[j]`, quindi le due istruzioni che realizzano il test sono

```
slt     $t0, $t4, $t3         # $t0 = 0 se $t4 >= $t3
beq     $t0, $zero, esci2     # vai a esci2 se $t4 >= $t3
```


Come ultima istruzione è presente un salto incondizionato che ritorna all'inizio del ciclo più interno:

```
j       for2tst               # torna al test del ciclo più
                              # interno
```

Combinando i vari frammenti, lo scheletro del ciclo *for* più interno è quindi il seguente:

```
  addi   $s1, $s0, -1          # j = i - 1
for2tst: slti $t0, $s1, 0      # $t0 = 1 se $s1 < 0 (j < 0)
  bne    $t0, $zero, esci2     # vai a esci2 se $s1 < 0 (j < 0)
  add    $t1, $s1, $s1         # $t1 = j * 2
  add    $t1, $t1, $t1         # $t1 = j * 4
  add    $t2, $a0, $t1         # $t2 = v + (j * 4)
  lw     $t3, 0($t2)           # $t3 = v[j]
  lw     $t4, 4($t2)           # $t4 = v[j+1]
  slt    $t0, $t4, $t3         # $t0 = 0 se $t4 >= $t3
  beq    $t0, $zero, esci2     # vai a esci2 se $t4 >= $t3
         ...
         (corpo del ciclo for più interno)
         ...
  addi   $s1, $s1, -1          # j = j - 1
  j      for2tst               # torna al test del ciclo più
                               # interno
esci2:
```

La chiamata a procedura in `ordina`

Il passo successivo consiste nel tradurre il corpo del ciclo *for* più interno:

```
scambia (v, j);
```

La chiamata alla procedura `scambia` è molto semplice:

```
jal    scambia
```

Il passaggio dei parametri in `ordina`

Il problema nasce dal fatto che la procedura `ordina` ha bisogno di conservare i propri parametri nei registri `$a0` e `$a1`, mentre la procedura `scambia` si aspetta i propri esattamente negli stessi due registri. Una possibile soluzione consiste nel copiare i parametri della procedura `ordina` all'interno di altri registri nella fase iniziale della procedura stessa, in modo da rendere disponibili a `scambia` i registri `$a0` e `$a1` (la copia è più veloce della memorizzazione e del recupero dei dati dallo stack). Quindi all'inizio della procedura si copiano `$a0` e `$a1` rispettivamente in `$s2` e `$s3`:

```
move    $s2, $a0     # copia del parametro $a0 in $s2
move    $s3, $a1     # copia del parametro $a1 in $s3
```

A questo punto si passano i parametri alla procedura `scambia` con le due istruzioni seguenti:

```
move    $a0, $s2     # il primo parametro di scambia è v
move    $a1, $s1     # il secondo parametro di scambia è j
```

Il salvataggio dei registri in `ordina`

Rimane ancora da scrivere la parte di codice relativa al salvataggio ed al ripristino dei registri. Chiaramente è necessario salvare l'indirizzo di ritorno nel registro `$ra`, dal momento che `ordina` è una procedura ed è stata essa stessa chiamata. Inoltre la procedura `ordina` utilizza i registri `$s0`, `$s1`, `$s2` e `$s3`, che devono quindi essere salvati. La parte iniziale della procedura `ordina` è quindi

```
addi   $sp, $sp, -20   # crea spazio nello stack per 5'registri
sw     $ra, 16($sp)    # salva $ra nello stack
sw     $s3, 12($sp)    # salva $s3 nello stack
sw     $s2, 8($sp)     # salva $s2 nello stack
sw     $s1, 4($sp)     # salva $s1 nello stack
sw     $s0, 0($sp)     # salva $s0 nello stack
```

La parte finale della procedura contiene l'inverso di tutte queste istruzioni, seguite da un `jr` all'indirizzo di ritorno.

La procedura `ordina` completa

Tutti i frammenti della procedura sono riuniti nella figura 3.26, avendo avuto cura di sostituire i riferimenti ai registri `$a0` e `$a1` all'interno dei cicli *for* con riferimenti ai registri `$s2` e `$s3`. Per rendere il codice più leggibile, a ciascun blocco di codice è nuovamente associata la finalità. In questo esempio le 9 linee di codice C della procedura `ordina` sono tradotte in 35 linee di codice assembler.

Approfondimento: Un tipo di ottimizzazione che in questo caso può dare buoni risultati è l'*inserimento in linea (inlining) della procedura*: invece di passare gli argomenti sotto forma di parametri e di chiamare il codice con un'istruzione `jal`, il compilatore copia il codice dal corpo della procedura `scambia` nel punto in cui compare la sua chiamata. In questo esempio l'inserimento in linea eliminerebbe quattro istruzioni. Il lato negativo di questa ottimizzazione è che il codice compilato ha dimensione maggiore, dato che si assume che la procedura in linea sia chiamata da più locazioni di memoria; un'espansione del codice di questo tipo può avere prestazioni *peggiori* se aumenta la percentuale di accesso a dati che non si trovato nella cache (vedere il Capitolo 7).

I compilatori MIPS riservano sempre dello spazio nello stack per gli argomenti, nel caso che questi debbano essere salvati, quindi in realtà decrementano `$sp` di 16 per fare spazio a tutti e 4 i registri argomento (16 byte). Una ragione di ciò è che il C ha un'opzione `vararg` che consente ad un puntatore di selezionare, ad esempio, il terzo parametro di una procedura. Quando il compilatore incontra una (poco frequente) `vararg`, copia i registri all'interno delle locazioni riservate dello stack.

3.11 Vettori e puntatori

Tutti coloro che iniziano a programmare devono affrontare la sfida dei puntatori. Confrontare un codice assembler che usa vettori ed indici con un codice assembler che utilizza i puntatori aiuta a comprendere le differenze. Questo paragrafo mostra le versioni in C ed assembler di due procedure che azzerano una sequenza di parole in memoria, una usando gli indici e l'altra usando i puntatori. La figura 3.27 mostra le due procedure in C.

Salvataggio dei registri

```
ordina: addi  $sp, $sp, -20      # crea spazio nello stack per 5 registri
        sw    $ra, 16($sp)       # salva $ra nello stack
        sw    $s3, 12($sp)       # salva $s3 nello stack
        sw    $s2, 8($sp)        # salva $s2 nello stack
        sw    $s1, 4($sp)        # salva $s1 nello stack
        sw    $s0, 0($sp)        # salva $s0 nello stack
```

Corpo della procedura

```
Spostamento
dei parametri
        move  $s2, $a0           # copia del parametro $a0 in $s2 (salva $a0)
        move  $s3, $a1           # copia del parametro $a1 in $s3 (salva $a1)

Ciclo più esterno
        move  $s0, $zero         # i = 0
for1tst:slt   $t0, $s0, $s3      # $t0 = 0 se $s0 >= $s3 (i >= n)
        beq   $t0, $zero, esci1  # vai a esci1 se $s0 >= $s3 (i >= n)

Ciclo più interno
        addi  $s1, $s0, -1       # j = i - 1
for2tst:slti  $t0, $s1, 0        # $t0 = 1 se $s1 < 0 (j < 0)
        bne   $t0, $zero, esci2  # vai a esci2 se $s1 < 0 (j < 0)
        add   $t1, $s1, $s1      # $t1 = j * 2
        add   $t1, $t1, $t1      # $t1 = j * 4
        add   $t2, $s2, $t1      # $t2 = v + (j * 4)
        lw    $t3, 0($t2)        # $t3 = v[j]
        lw    $t4, 4($t2)        # $t4 = v[j+1]
        slt   $t0, $t4, $t3      # $t0 = 0 se $t4 >= $t3
        beq   $t0, $zero, esci2  # vai a esci2 se $t4 >= $t3

Passaggio
dei parametri
e chiamata
        move  $a0, $s2           # il primo parametro di scambia è v (ex $a0)
        move  $a1, $s1           # il secondo parametro di scambia è j
        jal   scambia            # codice scambia in figura 3.24

Ciclo più interno
        addi  $s1, $s1, -1       # j = j - 1
        j     for2tst            # torna al test del ciclo più interno

Ciclo più esterno
esci2:  addi  $s0, $s0, 1        # i = i + 1
        j     for1tst            # torna al test del ciclo più esterno
```

Ripristino dei registri

```
esci1:  lw    $s0, 0($sp)        # ripristina $s0 dallo stack
        lw    $s1, 4($sp)        # ripristina $s1 dallo stack
        lw    $s2, 8($sp)        # ripristina $s2 dallo stack
        lw    $s3, 12($sp)       # ripristina $s3 dallo stack
        lw    $ra, 16($sp)       # ripristina $ra dallo stack
        addi  $sp, $sp, 20       # riaggiorna lo stack pointer
```

Ritorno dalla procedura

```
        jr    $ra                # ritorno alla procedura chiamante
```

Figura 3.26 La versione in linguaggio assembler MIPS della procedura `ordina` di figura 3.25.

Interfaccia Hardware Software

La gente era solita sentirsi suggerire l'uso dei puntatori in C per ottenere codice più efficiente di quello ottenibile usando indici e vettori: "usate i puntatori, anche se il codice diventa incomprensibile." La procedura `azzera2` di figura 3.27 è un esempio di questo tipo. I compilatori ottimizzati di oggi possono produrre un codice altrettanto efficiente usando indici e vettori. Lo scopo del presente paragrafo è mostrare come i puntatori vengano tradotti in istruzioni MIPS, non sostenere uno stile di programmazione su cui c'è molto da discutere.

```
azzera1 (int vett[], int dim)
{
  int i;
  for (i = 0; i < dim; i = i + 1)
       vett[i] = 0;
}

azzera2 (int *vett, int dim)
{
  int *p;
  for (p = &vett[0]; p < &vett[dim]; p = p + 1)
       *p = 0;
}
```

Figura 3.27 Due procedure C che azzerano tutti gli elementi di un vettore. `Azzera1` utilizza un indice, mentre `azzera2` un puntatore. La seconda procedura necessita di qualche spiegazione per chi non ha familiarità con il linguaggio C. L'indirizzo di una variabile è indicato da `&`, mentre l'oggetto puntato da un puntatore è indicato da `*`. Dalle dichiarazioni si vede che sia `vett` che `p` sono puntatori ad interi. La prima parte del ciclo *for* della procedura `azzera2` assegna l'indirizzo del primo elemento di `vett` al puntatore `p`; la seconda parte effettua un controllo per vedere se il puntatore ha superato l'ultimo elemento di `vett`. Incrementare di uno il puntatore, nell'ultima parte del ciclo *for*, significa spostarlo all'elemento successivo di dimensioni pari a quelle dichiarate: dato che `p` è un puntatore ad interi, il compilatore genera delle istruzioni MIPS che incrementano `p` di 4 unità, corrispondenti al numero di byte di un intero MIPS. L'assegnazione all'interno del ciclo pone a 0 l'oggetto puntato da `p`.

La versione della procedura che utilizza un vettore

Si parte con la procedura `azzera1`, la versione che utilizza un vettore, concentrandosi sul corpo del ciclo ed ignorando il resto del codice. Si assume inoltre che i due parametri `vett` e `dim` si trovino nei registri `$a0` e `$a1` e che `i` sia allocato nel registro `$t0`.

L'inizializzazione di `i`, cioè la prima parte del ciclo *for*, è diretta:

```
        move   $t0, $zero          # i = 0 ($t0 = 0)
```

Per porre `vett[i]` a 0 per prima cosa occorre trovare il suo indirizzo: a questo scopo si moltiplica `i` per 4 ottenendo l'indirizzo in byte:

```
ciclo1: add    $t1, $t0, $t0       # $t1 = i * 2
        add    $t1, $t1, $t1       # $t1 = i * 4
```

Dal momento che l'indirizzo di partenza del vettore è in un registro, si deve sommare l'indice per ottenere l'indirizzo di `vett[i]`:

```
        add    $t2, $a0, $t1       # $t2 = indirizzo di vett[i]
```

Questo è un esempio classico di indirizzamento tramite indice (si veda la parte del Paragrafo 3.12 riguardante il PowerPC). Ora finalmente si può memorizzare 0 nella locazione trovata:

```
        sw     $zero, 0($t2)       # vett[i] = 0
```

Questa istruzione si trova alla fine del corpo del ciclo *for*, quindi il passo successivo consiste nell'incrementare `i`:

```
        addi   $t0, $t0, 1         # i = i + 1
```


Il test del ciclo verifica se `i` è minore di `dim`:

```
        slt    $t3, $t0, $a1       # $t3 = (i < dim)
        bne    $t3, $zero, ciclo1  # vai a ciclo1 se (i < dim)
```

Fino a qui sono stati mostrati tutti i frammenti della procedura; il codice MIPS completo che azzera tutti gli elementi di un vettore (usando un indice) è quindi il seguente:

```
        move   $t0, $zero          # i = 0 ($t0 = 0)
ciclo1: add    $t1, $t0, $t0       # $t1 = i * 2
        add    $t1, $t1, $t1       # $t1 = i * 4
        add    $t2, $a0, $t1       # $t2 = indirizzo di
vett[i]
        sw     $zero, 0($t2)       # vett[i] = 0
        addi   $t0, $t0, 1         # i = i + 1
        slt    $t3, $t0, $a1       # $t3 = (i < dim)
        bne    $t3, $zero, ciclo1  # vai a ciclo1 se (i < dim)
```

Si noti che questo codice funziona solo se `dim` è maggiore di zero.

La versione della procedura che utilizza i puntatori

La seconda procedura, quella che utilizza i puntatori, alloca i due parametri `vett` e `dim` nei registri `$a0` e `$a1` e il puntatore `p` in `$t0`. Il codice inizia con l'assegnazione dell'indirizzo del primo elemento del vettore al puntatore `p`:

```
        move   $t0, $a0            # p = indirizzo di vett[0]
```

La parte successiva di codice è relativa al corpo del ciclo *for*, che semplicemente memorizza 0 nell'oggetto puntato da `p`:

```
ciclo2: sw     $zero, 0($t0)       # memoria[p] = 0
```

Questa istruzione costituisce il corpo del ciclo, mentre l'istruzione seguente deve aggiornare `p` per far sì che punti alla parola successiva:

```
        addi   $t0, $t0, 4         # p = p + 4
```

In C incrementare un puntatore di 1 unità significa farlo puntare all'oggetto successivo nella sequenza definita. Dato che `p` è un puntatore ad interi, ciascuno dei quali occupa 4 byte, il compilatore incrementa `p` di 4 unità.

La fase seguente consiste nel test di fine ciclo; per prima cosa occorre calcolare l'indirizzo dell'ultimo elemento del vettore, moltiplicando `dim` per 4 al fine di ottenere il valore in byte:

```
        add    $t1, $a1, $a1       # $t1 = dim * 2
        add    $t1, $t1, $t1       # $t1 = dim * 4
```

e sommando il prodotto all'indirizzo di partenza del vettore per ottenere l'indirizzo della prima parola che *non appartiene* al vettore:

```
        add    $t2, $a0, $t1       # $t2 = indirizzo
                                   # di vett[dim]
```

Il test controlla semplicemente che `p` sia minore dell'ultimo elemento di `vett`:

```
        slt    $t3, $t0, $t2       # $t3 = (p < &vett[dim])
        bne    $t3, $zero, ciclo2  # vai a ciclo2 se
                                   # (p < &vett[dim])
```

Avendo completato i vari frammenti, la versione di codice che usa i puntatori per azzerare un vettore è la seguente:

```
        move   $t0, $a0              # p = indirizzo di vett[0]
ciclo2: sw     $zero, 0($t0)         # memoria[p] = 0
        addi   $t0, $t0, 4           # p = p + 4
        add    $t1, $a1, $a1         # $t1 = dim * 2
        add    $t1, $t1, $t1         # $t1 = dim * 4
        add    $t2, $a0, $t1         # $t2 = indirizzo di vett[dim]
        slt    $t3, $t0, $t2         # $t3 = (p < &vett[dim])
        bne    $t3, $zero, ciclo2    # vai a ciclo2 se
                                     # (p < &vett[dim])
```

Come nel caso della prima versione, il codice assume che `dim` sia maggiore di 0.

Si noti che questo programma calcola l'indirizzo corrispondente alla fine del vettore durante ogni iterazione del ciclo, anche se non cambia; spostando il calcolo al di fuori del ciclo si ottiene una versione del codice cui corrisponde un tempo di esecuzione inferiore:

```
        move   $t0, $a0              # p = indirizzo di vett[0]
        add    $t1, $a1, $a1         # $t1 = dim * 2
        add    $t1, $t1, $t1         # $t1 = dim * 4
        add    $t2, $a0, $t1         # $t2 = indirizzo di vett[dim]
ciclo2: sw     $zero, 0($t0)         # memoria[p] = 0
        addi   $t0, $t0, 4           # p = p + 4
        slt    $t3, $t0, $t2         # $t3 = (p < &vett[dim])
        bne    $t3, $zero, ciclo2    # vai a ciclo2 se
                                     # (p < &vett[dim])
```

Il confronto fra le due versioni

Confrontare le due versioni affiancando l'una all'altra consente di illustrare le differenze fra indici e puntatori (i cambiamenti introdotti dall'uso dei puntatori sono evidenziati):

```
        move $t0, $zero          # i = 0 ($t0 = 0)
ciclo1: add  $t1, $t0, $t0       # $t1 = i * 2
        add  $t1, $t1, $t1       # $t1 = i * 4
        add  $t2, $a0, $t1       # $t2 = &vett[i]
        sw   $zero, 0($t2)       # vett[i] = 0
        addi $t0, $t0, 1         # i = i + 1
        slt  $t3, $t0, $a1       # $t3 = (i < dim)
        bne  $t3, $zero, ciclo1  # vai a ciclo1 se ()
```

```
        move  $t0, $a0           # p = &vett[0]
        add   $t1, $a1, $a1      # $t1 = dim * 2
        add   $t1, $t1, $t1      # $t1 = dim * 4
        add   $t2, $a0, $t1      # $t2 = &vett[dim]
ciclo2: sw    $zero, 0($t0)      # memoria[p] = 0
        addi  $t0, $t0, 4        # p = p + 4
        slt   $t3, $t0, $t2      # $t3 = (p < &vett[dim])
        bne   $t3, $zero, ciclo2 # vai a ciclo2 se ()
```

La versione di sinistra deve avere la "moltiplicazione" e la somma necessariamente all'interno del ciclo, dato che `i` viene incrementato e quindi l'indirizzo deve essere ricalcolato a partire dal nuovo valore; la versione di destra incrementa direttamente il puntatore `p`. La versione con i puntatori riduce quindi da 7 a 4 il numero di istruzioni eseguite per ogni ciclo. Molti compilatori di oggi comunque ottimizzerebbero il codice C di `azzeral` per produrre codice assembler simile a quello riportato a destra.

Approfondimento: Il compilatore C aggiungerebbe un test per assicurarsi che dim sia maggiore di 0; un modo per fare ciò potrebbe essere aggiungere, appena prima dell'inizio del ciclo, un'istruzione jump che salti all'istruzione `slt`.

3.12 Un caso reale: le istruzioni del Power PC e dell'80x86

La bellezza è negli occhi di chi guarda.

Detto popolare

Chi progetta il set di istruzioni talvolta fornisce istruzioni più potenti di quelle che si trovano nel MIPS. Lo scopo è ridurre il numero di istruzioni eseguite da un programma, ma il rischio è che tale riduzione vada a scapito della semplicità, aumentando il tempo di esecuzione dei programmi a causa della maggiore lentezza con cui vengono eseguite le istruzioni. Tale rallentamento può derivare da un maggior periodo di clock o dalla necessità di un maggior numero di cicli per eseguire le istruzioni (si riveda a questo proposito il Paragrafo 2.8).

La strada che porta a operazioni complesse è dunque irta di pericoli: per evitare questo tipo di problemi i progettisti tendono a seguire quella delle istruzioni semplici. Il Paragrafo 3.13 mostra un classico errore legato alla complessità delle istruzioni.

L'IBM / Motorola PowerPC

Il PowerPC, costruito da IBM e Motorola e montato sul Macintosh della Apple, ha forti analogie con il MIPS: entrambi hanno i registri di 32 bit, le istruzioni lunghe 32 bit ed il trasferimento dati possibile soltanto tramite operazioni di tipo load e store. La principale differenza consiste in due modi di indirizzamento addizionali e in poche operazioni in più.

Indirizzamento tramite indice

Negli esempi precedenti sono stati esaminati casi in cui un registro mantiene l'indirizzo di base di un vettore ed un altro registro l'indice del vettore. Il PowerPC fornisce un modo di indirizzamento, spesso chiamato *indexed addressing* (indirizzamento tramite indice), che permette di sommare i valori dei due registri: il codice MIPS

```
add   $t0, $a0, $s3   # $a0 contiene la base di un vettore,
                      # $s3 l'indice
lw    $t1, 0($t0)     # $t1 = memoria[$a0+$s3]
```

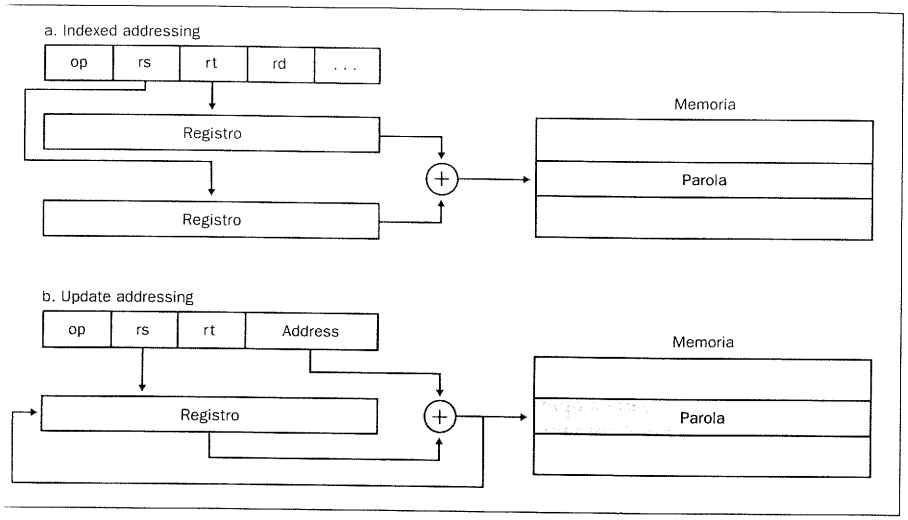
nel PowerPC potrebbe essere sostituito da una sola istruzione:

```
lw    $t1, $a0+$s3    # $t1 = memoria[$a0+$s3]
```

Utilizzando le stesse notazioni della figura 3.17, la figura 3.28 mostra l'indirizzamento tramite indice; questo modo di indirizzamento è disponibile per le operazioni di load e di store.

Indirizzamento di tipo update

Si immagini il caso di una sequenza di codice da applicare ad un vettore di parole in memoria, come nel caso della versione `azzeral` della procedura di azzeramento di un vettore (paragrafo precedente). Il caricamento di una parola seguito dal-



igura 3.28 I modi di indirizzamento indexed e update. Sono evidenziati gli operandi.

l'incremento del registro di base per puntare alla parola successiva sono operazioni eseguite frequentemente. L'idea che sta alla base di questo modo di indirizzamento (*update addressing*) è quella di avere una nuova versione delle istruzioni per il trasferimento dei dati che automaticamente incrementi il registro di base in modo da puntare alla parola successiva ogni volta che viene fatto il trasferimento di un dato. Dato che l'architettura MIPS usa l'indirizzamento a byte e una parola è composta da 4 byte, questo nuovo modo di indirizzamento è equivalente alla coppia di istruzioni

```
lw    $t0, 4($s3)    # $t0 = memoria[$s3+4]
addi  $s3, $s3, 4    # $s3 = $s3 + 4
```

Il PowerPC include un'istruzione del tipo

```
lwu   $t0, 4($s3)    # $t0 = memoria[$s3+4]; $s3 = $s3 + 4
```

che quindi aggiorna il valore del registro all'interno dell'istruzione di load. La figura 3.28 mostra anche l'indirizzamento di tipo update. Il PowerPC consente l'update addressing sia per il base addressing che per l'indexed addressing e sia per le operazioni load che per quelle di store.

Le istruzioni in più del PowerPC

Le istruzioni del PowerPC seguono lo stesso stile architetturale di quelle del MIPS, affidandosi all'esecuzione veloce di istruzioni semplici per ottenere buone prestazioni; ci sono però alcune eccezioni.

La prima è l'esistenza di operazioni di load e di store multiplo, che possono trasferire fino a 32 parole in una singola istruzione e che utilizzate in modo congiunto consentono di riallocare velocemente i dati da una porzione di memoria ad un'altra. Queste operazioni inoltre riducono la dimensione del codice durante il salvataggio ed il ripristino dei registri.

Un secondo esempio riguarda i cicli. Il PowerPC ha un particolare registro, distinto dagli altri 32, che ha le funzioni di contatore e che è utilizzato per migliorare le prestazioni nell'esecuzione dei cicli *for*.

Si supponga di voler eseguire il seguente codice C:

```
for (i = n; i != 0; i = i - 1)
  {...}
```

Per decrementare un registro, confrontarlo con il valore 0 ed eseguire un salto ogni volta che è diverso, si dovrebbero usare le seguenti istruzioni MIPS:

```
Ciclo: ...
       addi  $t0, $t0, -1        # $t0 = $t0 - 1
       bne   $t0, $zero, Ciclo   # vai a Ciclo se $t0 != 0
```

mentre con il PowerPC si potrebbe usare un'unica istruzione:

```
       bc    Ciclo, ctr!=0       # $ctr = $ctr -1; vai a Ciclo
                                 # se $ctr != 0
```

Interfaccia Hardware Software

Oltre a violare il principio della semplicità, queste operazioni sofisticate potrebbero non corrispondere *esattamente* a ciò che il compilatore deve generare. Se ad esempio il compilatore dovesse decrementare il registro di quattro unità invece di una o se dovesse effettuare un salto nel caso di una relazione del tipo "minore o uguale a" invece del caso di disuguaglianza, allora le istruzioni descritte sopra non andrebbero bene. Quando sono posti di fronte a questo tipo di obiezioni, i progettisti del set di istruzioni dovrebbero generalizzare le operazioni, aggiungendo un operando addizionale per specificare la quantità dell'incremento o un'opzione per definire la condizione per l'esecuzione del salto. A questo punto però si corre il rischio che il caso più comune, come l'incremento di una unità, sia più lento di una sequenza di operazioni semplici.

L'Intel 80x86

Il MIPS fu il risultato del lavoro di un ristretto gruppo di persone nel 1985; i blocchi di questa architettura si adattano bene l'uno all'altro e l'intera architettura può essere descritta in modo conciso. Questo non è il caso dell'80x86, che è il risultato del lavoro indipendente di molti gruppi ed è in evoluzione da circa 20 anni, con l'aggiunta progressiva di nuove funzioni al set di istruzioni originale. Le pietre miliari dell'evoluzione dell'80x86 sono:

- **1978:** L'architettura 8086 viene annunciata come un'estensione assembler-compatibile dell'allora popolarissimo 8080, un microprocessore ad 8 bit. L'8086 è un'architettura a 16 bit, con tutti i registri interni da 16 bit; a differenza del MIPS, i registri sono dedicati ad usi specifici, quindi l'8086 non è considerato un'architettura di tipo *general-purpose register*.
- **1980:** Viene annunciato il coprocessore matematico 8087, che estende l'architettura 8086 con 60 nuove istruzioni in virgola mobile. Invece di usare i registri basa le proprie operazioni su uno stack (vedere i Paragrafi 3.4 e 4.9).

- **1982:** L'80286 estende l'architettura dell'8086 aumentando lo spazio di indirizzamento a 24 bit attraverso la creazione di un complicato sistema di mappatura degli indirizzi ed una modalità di utilizzo di tipo protetto (vedere il Capitolo 7) ed aggiungendo alcune istruzioni per rendere più regolare il set di istruzioni e per gestire il modo protetto.
- **1985:** L'80386 estende l'architettura dell'80286 a 32 bit. Oltre a fornire un'architettura a 32 bit, con registri da 32 bit e spazio di indirizzamento pari a 32 bit, l'80386 aggiunge nuovi modi di indirizzamento e nuove operazioni. Le istruzioni addizionali rendono l'80386 vicino ad essere un calcolatore general-purpose register. Inoltre esso fornisce un supporto alla paginazione oltre che alla segmentazione degli indirizzi (vedere il Capitolo 7). Come l'80286, anche l'80386 ha una modalità che permette di eseguire i programmi scritti per l'8086 senza apportare modifiche.
- **1989-95:** I processori successivi, l'80486 del 1989, il Pentium del 1992 e il Pentium Pro del 1995, mirano ad ottenere prestazioni più elevate, aggiungendo solo quattro nuove istruzioni al set visibile all'utente: tre per la gestione multiprocessore (vedere Capitolo 9) ed una per il trasferimento di dati condizionato.
- **1997:** Dopo l'introduzione in commercio del Pentium e del Pentium Pro, l'Intel annuncia l'espansione di queste architetture con il set di istruzioni MMX. Questo nuovo set di 57 istruzioni usa uno stack di dati in virgola mobile per accelerare le applicazioni legate alla multimedialità ed alle comunicazioni. Le istruzioni MMX tipicamente operano su più dati alla volta, secondo la tradizione delle architetture SIMD (single instruction, multiple data: istruzione singola, dati multipli).

Come si vede da questa storia, la compatibilità ha influito come una "gabbia dorata" sull'architettura dell'80x86: ad ogni passo il software preesistente era troppo importante perché si rischiasse di perderlo con delle modifiche architetturali significative.

Qualunque siano i difetti dell'80x86, si tenga comunque a mente che esistono più processori appartenenti a questa famiglia che a qualunque altra nel mondo, forse 300 milioni nel 1997. Nonostante ciò, il peso delle generazioni precedenti ha prodotto un'architettura difficile da spiegare ed impossibile da apprezzare.

Un consiglio al lettore su quanto segue è di non leggere il paragrafo con l'attenzione di chi vuole imparare a scrivere programmi in assembler 80x86: lo scopo qui è mostrare la forza e la debolezza dell'architettura più diffusa nel mondo.

Invece di mostrare l'intero set di istruzioni a 16 e a 32 bit, in questo paragrafo ci si concentrerà sul sottoinsieme delle istruzioni a 32 bit nato con l'80386, dal momento che questa parte dell'architettura assumerà sempre maggiore importanza col tempo. Si inizia la trattazione con i registri e i modi di indirizzamento, per passare poi alle operazioni sui numeri interi e concludere con l'analisi della codifica delle istruzioni.

I registri e i modi di indirizzamento dell'80x86

L'evoluzione del set di istruzioni si nota chiaramente osservando i registri dell'80386 (vedere figura 3.29): l'80386 essenzialmente estende tutti i registri a 16 bit (a parte i registri di segmento) a 32 bit, mettendo una *E* davanti al nome di ciascuno per indicare la versione a 32 bit. I registri nel loro complesso saranno qui

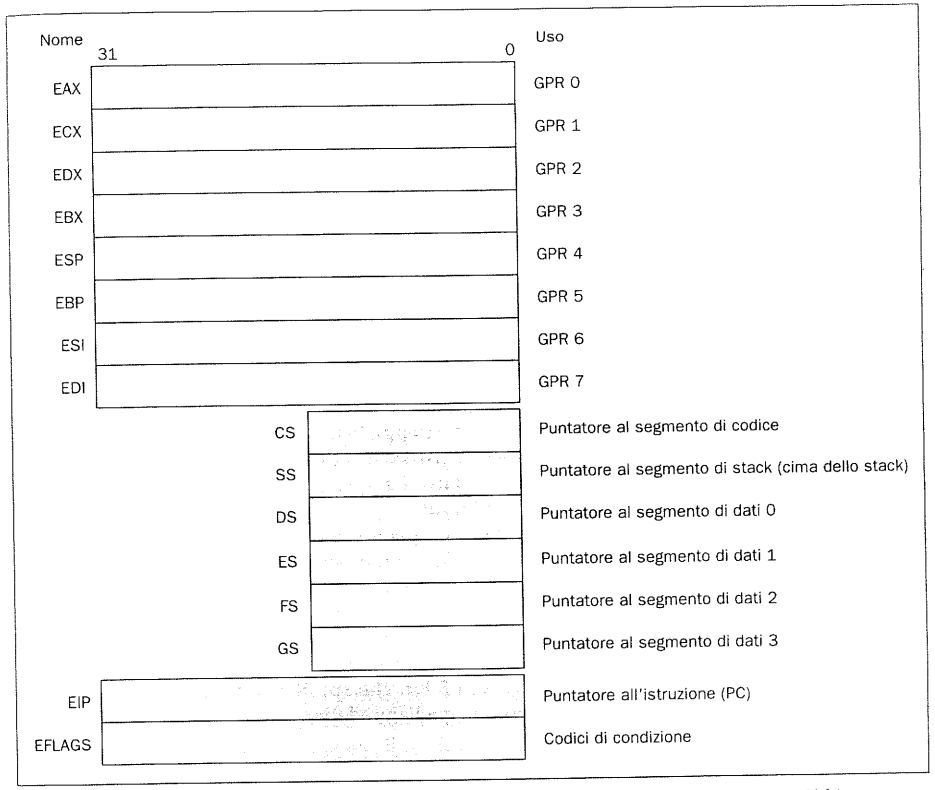


Figura 3.29 Il set dei registri dell'80386. A partire dall'80386, gli 8 registri in alto sono estesi a 32 bit e possono essere usati come registri general-purpose.

chiamati GPR (general-purpose registers, cioè registri general-purpose). L'80386 ha solo 8 GPR, quindi il MIPS ne può usare quattro volte tanti.

Le istruzioni aritmetiche, logiche e di trasferimento dati hanno due operandi e le combinazioni consentite sono illustrate in figura 3.30. Ci sono due importanti differenze rispetto al MIPS. La prima è che nell'80x86 le istruzioni aritmetiche e logiche hanno sempre un operando che funge sia da sorgente che da destinazione, mentre il MIPS consente di avere registri distinti per sorgenti e destinazione; questa restrizione rende più stringente il problema del numero limitato di registri, dato che uno dei registri sorgente deve essere necessariamente modificato. La seconda importante differenza consiste nel fatto che uno degli operandi può essere in memoria, a differenza del MIPS e del PowerPC.

I sette modi di indirizzamento relativi ai dati in memoria, descritti in dettaglio poco oltre, consentono di inserire nelle istruzioni indirizzi di due dimensioni diverse: l'entità chiamata *displacement* (spiazzamento) può essere di 8 o di 32 bit.

Benché possa essere applicato qualunque modo di indirizzamento ad un operando in memoria, ci sono delle restrizioni sui *registri* utilizzabili nei vari modi. La

Tipo dell'operando sorgente / destinazione	Tipo del secondo operando sorgente
Registro	Registro
Registro	Immediato
Registro	Memoria
Memoria	Registro
Memoria	Immediato

Figura 3.30 Le istruzioni aritmetiche, logiche e di trasferimento dati. L'80x86 consente le combinazioni di operandi mostrate in figura: l'unica restrizione riguarda l'assenza di una modalità memoria-memoria. Gli operandi immediati possono avere lunghezza 8, 16 o 32 bit; il termine registro indica uno dei 14 registri principali di figura 3.29 (esclusi EIP ed EFLAGS).

figura 3.31 mostra i modi di indirizzamento dell'80x86 ed i GPR che non possono essere usati con tale modo, oltre alle istruzioni MIPS per ottenere lo stesso risultato.

Le operazioni sui numeri interi dell'80x86

L'8086 consente di utilizzare dati a 8 bit (*byte*) o a 16 bit (parola); l'80386 aggiunge gli indirizzi ed i dati a 32 bit (*double word* o doppia parola). La restrizione sui tipi di dati si applica sia alle operazioni sui registri che agli accessi in memoria. Quasi tutte le operazioni possono lavorare su dati a 8 bit o su dati di dimensioni maggiori, di 16 o 32 bit a seconda del modo utilizzato.

Chiaramente alcuni programmi devono operare su dati di tutte e tre le dimensioni, quindi l'80386 fornisce un modo efficiente di specificare la loro dimensione senza estendere il codice in modo significativo. I progettisti pensarono che la maggior parte dei programmi avrebbe usato o dati a 16 bit o dati a 32 bit e decisero di scegliere a priori una di queste due dimensioni. La dimensione è specificata da uno dei bit del registro di segmento del codice e per sovrascrivere l'opzione occorre far precedere l'istruzione da un prefisso di 8 bit, che specifica all'elaboratore che il dato utilizzato ha dimensione diversa da quella predefinita.

Modo	Descrizione	Restrizione sui registri	Codice MIPS equivalente
Register indirect	L'indirizzo è in un registro	no ESP o EBP	`lw $s0, 0($s1)`
Based mode con displacement a 8 o 32 bit	L'indirizzo è il contenuto del registro di base più il displacement	no ESP o EBP	`lw $s0, 100($s1)` `# displacement <= 16 bit`

Base plus scaled index	L'indirizzo è
	Base + (2^{Scale} × Index),
	dove Scale può valere 0, 1, 2 o 3
		Base: qualsiasi GPR
		Index: no ESP
			`mul $t0, $s1, 4`
			`add $t0, $t0, $s1`
			`lw $s0, 0($t0)`

Base plus scaled index con displacement a 8 o 32 bit	L'indirizzo è
	Base + (2^{Scale} × Index) + displacement,
	dove Scale può valere 0, 1, 2 o 3
		Base: qualsiasi GPR
		Index: no ESP
			`mul $t0, $s1, 4`
			`add $t0, $t0, $s1`
			`lw $s0, 100($t0)`
			`# displacement <= 16 bit`

Figura 3.31 I modi di indirizzamento a 32 bit dell'80x86, con le restrizioni sui registri ed il codice MIPS equivalente. Il modo di indirizzamento Based plus scaled index, che non si trova né nel MIPS né nel PowerPC, è incluso per evitare la moltiplicazione per quattro (fattore di scala pari a 2) necessaria a trasformare l'indice contenuto in un registro in un indirizzo in byte (vedere figure 3.24 e 3.26). Un fattore di scala pari a 1 è utilizzato per i dati a 16 bit ed un fattore pari a 3 per quelli a 64 bit. Un fattore di scala pari a 0 significa che l'indirizzo non viene modificato. Se nel secondo o nel quarto modo di indirizzamento il displacement è maggiore di 16 bit, allora il codice MIPS equivalente richiede due istruzioni in più: una `lui` per caricare i 16 bit più significativi del displacement e una `add` per sommare la parte alta dell'indirizzo con il registro di base `$s1`. L'Intel dà due nomi diversi al modo di indirizzamento Based, cioè Based e Indexed, ma essi sostanzialmente sono identici e appaiono insieme in questa figura.

La soluzione di adottare un prefisso è presa in prestito dall'8086, che prevedeva diversi prefissi per modificare il comportamento delle istruzioni. I tre prefissi originari modificavano i registri di segmento predefiniti, impedivano l'accesso al bus per realizzare un semaforo (vedere il Capitolo 9) o ripetevano l'istruzione successiva fino a quando il registro ECX scendeva al valore 0. L'ultimo tipo di prefisso, affiancato ad una istruzione di move, permetteva di trasferire un numero variabile di byte. L'80386 aggiunge un altro prefisso per modificare la dimensione degli indirizzi.

Le operazioni sui numeri interi dell'80x86 possono essere suddivise in quattro principali categorie:

1. Le istruzioni per lo spostamento dei dati, che includono move, push e pop.
2. Le istruzioni aritmetiche e logiche, inclusa quella di confronto e le operazioni aritmetiche su numeri interi e decimali.
3. Le istruzioni di controllo del flusso, che includono i salti condizionati e incondizionati, le chiamate a procedura ed il ritorno.
4. Le istruzioni per la gestione delle stringhe, che includono lo spostamento ed il confronto fra stringhe.

Le prime due categorie non sono degne di nota, a parte il fatto che le istruzioni aritmetiche e logiche consentono che l'operando destinazione sia un registro o una locazione di memoria. La figura 3.32 mostra alcune istruzioni tipiche dell'80x86 con il loro significato.

Nel PowerPC e nell'80x86 i salti condizionati si basano sui codici di controllo o *flag*, modificati in seguito ad una operazione; sono prevalentemente usati per confrontare con 0 un certo risultato. I salti condizionati quindi effettuano un controllo sui flag. A favore delle condizioni di controllo c'è il fatto che sono parte delle operazioni e che il loro test è più veloce del confronto fra due registri, come nel caso di `beq` e `bne` del MIPS. A loro sfavore invece c'è il fatto che, dovendo usare dell'hardware addizionale, il confronto con lo 0 allunga i tempi delle operazioni, e che spesso il programmatore ha bisogno di testare valori diversi dal risultato dell'ultima operazione. Inoltre gli indirizzi di salto PC-relative devono essere specificati come numero di byte, dato che a differenza del MIPS nell'80x86 non è vero che tutte le istruzioni sono lunghe 32 bit.

Le istruzioni per la gestione delle stringhe risalgono all'8080 e nella maggior parte dei programmi non vengono utilizzate; infatti in generale sono più lente delle

Istruzione	Funzione
JE nome	if equal (condition code) {EIP=nome}; EIP-128 <= nome < EIP+128
JMP nome	EIP=nome;
CALL nome	SP=SP-4; M[SP]=EIP+5; EIP=nome
MOVW EBX, [EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX, #6765	EAX=EAX+6765
TEST EDX, #42	Setta i flag con il risultato dell'and fra EDX e 42_{esa}
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

Figura 3.32 Alcune istruzioni tipiche dell'80x86, con la relativa funzione. Una lista delle operazioni più frequenti compare in figura 3.33. L'istruzione `CALL` salva sullo stack il puntatore (EIP) all'istruzione successiva (EIP è il program counter dell'Intel).

Istruzione	Significato
Controllo	**Salti condizionati e incondizionati**
JNZ, JZ	Salto condizionato a EIP+offset a 8bit; nomi alternativi sono JNE (per JNZ) e JE (per JZ)
JMP	Salto incondizionato; offset a 8 o a 16 bit
CALL	Chiamata a procedura, con offset a 16 bit; l'indirizzo di ritorno è memorizzato nello stack
RET	Prende dallo stack l'indirizzo di ritorno ed esegue il salto
LOOP	Ciclo: decrementa ECX e salta a EIP+offset a 8bit se ECX è diverso da 0
Trasferimento dati	**Spostamento di dati fra registri o fra registri e memoria**
MOV	Sposta un dato da un registro ad un altro o fra registro e memoria
PUSH, POP	Mette nello stack l'operando sorgente; recupera dallo stack un operando e lo mette in un registro
LES	Carica dalla memoria ES ed uno dei GPR
Aritmetiche e logiche	**Operazioni aritmetiche e logiche su dati nei registri o in memoria**
ADD, SUB	Somma il sorgente alla destinazione, sottrae il sorgente alla destinazione; formato registro-memoria
CMP	Confronta il sorgente con la destinazione; formato registro-memoria
SHL, SHR, RCR	Scalamento a sinistra, scalamento a destra, rotazione verso destra con inserimento del flag di carry
CBW	Converte il byte che si trova negli 8 bit meno significativi di EAX in una parola che occupa i 16 bit meno significativi di EAX
TEST	Mette i valori opportuni nei flag facendo l'AND logico fra sorgente e destinazione
INC, DEC	Incrementa la destinazione, decrementa la destinazione; formato registro-memoria
OR, XOR	OR logico, OR esclusivo; formato registro-memoria
Stringhe	**Trasferimenti fra operandi stringhe; lunghezza data da un prefisso di ripetizione**
MOVS	Copia una stringa sorgente in una stringa destinazione incrementando ESI ed EDI; può essere ripetuta
LODS	Carica nel registro EAX un byte, una parola o una double word appartenente ad una stringa

Figura 3.33 Alcune operazioni tipiche dell'80x86. Molte operazioni usano il formato registro-memoria, dove uno dei due operandi (sorgente o destinazione) è in memoria e l'altro o è un registro o è un operando immediato.

procedure software equivalenti (vedere il primo errore riportato nel Paragrafo 3.13).

La figura 3.33 elenca alcune delle istruzioni sui numeri interi dell'80x86; molte di esse sono disponibili sia nel formato byte che in quello word.

La codifica delle istruzioni 80x86

La codifica delle istruzioni 80x86 è decisamente complessa, con molti formati diversi per le istruzioni, che possono variare da 1 byte (quando non ci sono operandi) a 17 byte. Questa può essere considerata la peggiore caratteristica dell'architettura 80x86.

La figura 3.34 mostra il formato di molte delle istruzioni illustrate in figura 3.32. Il byte del codice operativo solitamente contiene un bit che indica se l'operando è da 8 o 32 bit. Il codice operativo di alcune istruzioni include il modo di indirizzamento ed il registro; questo si verifica in molte istruzioni della forma "registro = registro – operazione – operando immediato". Altre istruzioni utilizzano un "postbyte" o extra byte per il codice operativo, etichettato con "mod, reg, r/m", che contiene le informazioni relative al modo di indirizzamento; questo byte aggiuntivo è presente in molte delle istruzioni che accedono alla memoria. Il modo di indirizzamento base plus scaled index usa un secondo byte aggiuntivo, etichettato con "sc, index, base".

La figura 3.35 mostra la codifica dei due post-byte per la specifica degli indirizzi, sia del modo a 16 bit che in quello a 32 bit. Sfortunatamente, per capire appieno

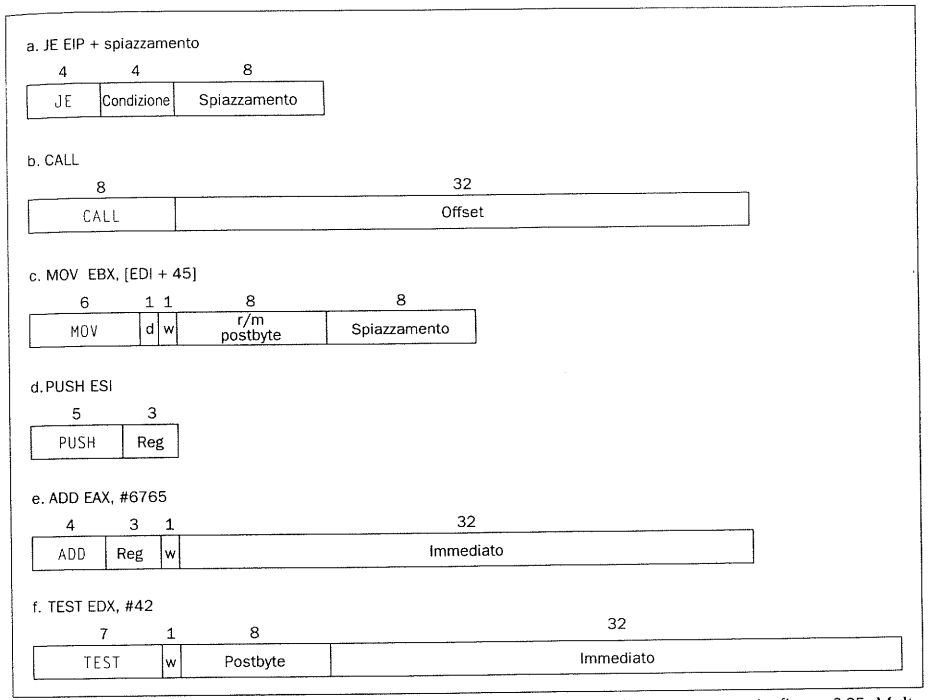


Figura 3.34 Formati tipici delle istruzioni 80x86. La codifica del post-byte è mostrata in figura 3.35. Molte istruzioni contengono il campo da 1 bit w, che specifica se l'operazione agisce sul byte o sulla double word. Il campo d all'interno della MOV si usa nelle operazioni di trasferimento dati da e verso la memoria per specificare la direzione del trasferimento. L'istruzione ADD richiede 32 bit per il campo immediato, dato che nel modo a 32 bit gli operandi immediati possono essere da 8 o da 32 bit. Il campo immediato dell'istruzione TEST è lungo 32 bit, perché nel modo a 32 bit non si possono usare operandi immediati da 8 bit. Nell'insieme le istruzioni variano di dimensione da 1 a 17 byte. Le dimensioni elevate possono essere dovute ai prefissi di un byte, agli operandi immediati su 4 byte, agli indirizzi con un displacement di 4 byte, ai codici operativi su 2 byte ed all'utilizzo del modo di indirizzamento base plus scaled index, che aggiunge un ulteriore byte.

quali registri e quali modi di indirizzamento siano disponibili, occorre considerare la codifica di tutti i modi di indirizzamento e a volte persino la codifica delle singole istruzioni.

Conclusioni sull'80x86

L'Intel ha costruito un processore a 16 bit due anni prima di quelli (più eleganti) dei suoi concorrenti (ad esempio il Motorola 68000) e questo ha fatto sì che l'8086 fosse scelto come processore del PC IBM. In generale gli ingegneri dell'Intel riconoscono che è più difficile costruire l'8086 piuttosto che un calcolatore tipo MIPS, ma il mercato molto più vasto consente all'Intel di avere molte risorse da investire per superare il problema della complessità crescente. Ciò che all'80x86 manca in stile, è compensato dalla diffusione, rendendolo apprezzabile se visto nella giusta prospettiva.

Ciò che lo salva è che i componenti architetturali dell'80x86 usati più frequentemente non sono troppo difficili da implementare, come l'Intel ha dimostrato

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
16b	32b	16b	32b	16b	32b	16b	32b				
0	AL	AX	EAX	0	ind=BX+SI	=EAX	*stesso ind di mod=0*	*stesso ind di mod=0*	*stesso ind di mod=0*	*stesso ind di mod=0*	*come il campo reg*
1	CL	CX	ECX	1	ind=BX+DI	=ECX	"	"	"	"	"
2	DL	DX	EDX	2	ind=BP+SI	=EDX	"	"	"	"	"
3	BL	BX	EBX	3	ind=BP+DI	=EBX	+disp8	+disp8	+disp16	+disp32	"
4	AH	SP	ESP	4	ind=SI	=*(sib)*	SI+disp8	*(sib)*+disp8	SI+ disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	ind=DI	= disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	ind=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	ind=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

Figura 3.35 La codifica 80x86 del primo post-byte "mod, reg, r/m" di specifica degli indirizzi. Le prime quattro colonne mostrano la codifica su 3 bit del campo reg, che dipende dal bit w del codice operativo e dal fatto che il calcolatore si trovi nel modo a 16 bit (8086) o in quello a 32 bit (80386). Le restanti colonne illustrano i campi mod e r/m. Il significato del campo di 3 bit r/m dipende dal valore contenuto nel campo da 2 bit mod e dalle dimensioni dell'indirizzo. In pratica i registri usati nel calcolo degli indirizzi sono elencati nella sesta e settima colonna sotto mod=0, sotto mod=1 aggiungendo un displacement di 8 bit o sotto mod=2 aggiungendo un displacement da 16 o 32 bit secondo il modo di indirizzamento. Le eccezioni sono: r/m=6 quando mod=1 o mod=2 nel modo a 16 bit seleziona BP più il displacement, r/m=5 quando mod=1 o mod=2 nel modo a 32 bit seleziona EBP più il displacement e r/m=4 nel modo a 32 bit quando mod è diverso da 3, dove (sib) significa che si utilizza il modo di indirizzamento scaled index mostrato in figura 3.31. Quando mod=3, il campo r/m indica un registro, seguendo lo stesso tipo di codifica del campo reg combinato con il bit w.

migliorando rapidamente le prestazioni dei programmi che lavorano sui numeri interi, sin dal 1978. Per ottenere queste prestazioni i compilatori devono evitare le parti dell'architettura difficili da implementare in modo veloce.

Errori e trabocchetti

Errore: più le istruzioni sono potenti, migliori sono le prestazioni.

Parte della potenzialità dell'80x86 sta nell'uso dei prefissi, che possono modificare l'esecuzione dell'istruzione successiva. Ad esempio esiste un prefisso che causa la ripetizione dell'istruzione seguente fino a che un contatore scende a 0, quindi per eseguire trasferimenti di dati da 32 bit da una porzione di memoria ad un'altra può sembrare che la soluzione migliore sia un'istruzione di move, preceduta dal prefisso di ripetizione. Su un processore Pentium a 133 MHz (con chip set Triton, DRAM EDO da 60 ns, cache da 256 KB) questo programma trasferirebbe i dati alla velocità di circa 40 MB/s.

Un metodo alternativo, che usa le istruzioni standard presenti su tutti i calcolatori, consiste nel caricare i dati nei registri (load) e poi memorizzare il contenuto dei registri in memoria (store). Questa seconda versione del programma, con il codice ripetuto in modo da evitare l'overhead dei cicli, sullo stesso processore trasferisce i dati ad una velocità di 60 MB/s, cioè 1,5 volte più in fretta. Una terza versione, che usa i registri in virgola mobile (più grandi) invece dei registri interi, trasferisce invece i dati ad una velocità di 80 MB/s, 2 volte più in fretta rispetto alla versione che usa l'istruzione complessa.

Errore: programmare in linguaggio assembler per ottenere le migliori prestazioni.

3.13

Tempo fa i compilatori producevano sequenze di istruzioni poco efficienti, ma la loro crescente sofisticatezza rende sempre minore la distanza fra il codice generato e quello scritto a mano. Infatti per essere in grado di competere con gli attuali compilatori il programmatore assembler dovrebbe avere perfettamente chiari i concetti dei Capitoli 6 e 7 sulla pipeline e sulla gerarchia di memoria.

La battaglia fra compilatori e programmatori assembler è in una fase in cui l'uomo perde terreno ogni giorno. Ad esempio il C offre al programmatore la possibilità di suggerire al compilatore quali variabili mantenere nei registri e quali in memoria. Quando i compilatori non erano molto efficienti nell'allocazione dei registri, questi suggerimenti avevano un'importanza vitale: molti libri di testo sul C dedicavano parte degli esempi al modo di dare suggerimenti efficaci al compilatore. Oggi in generale i compilatori ignorano questi suggerimenti, dato che sono in grado di allocare i registri meglio di quanto farebbe il programmatore.

Come controesempio specifico, si eseguano i programmi assembler MIPS riportati nelle figure 3.24 e 3.26 per confrontarne le prestazioni con i programmi C di figura 3.23 e 3.25. La figura 3.36 mostra i risultati: come si può vedere, il programma compilato è 1,5 volte più veloce del codice assemblato. Il compilatore è stato in grado di generare del codice assembler che si adatta perfettamente al contesto, mentre il codice scritto a mano era volutamente un po' più generico per renderlo più facile da capire e da modificare. Nello specifico, i miglioramenti apportati dal compilatore C sono una convenzione migliore per le chiamate a procedura ed una modifica nel calcolo degli indirizzi volta a spostare la moltiplicazione al di fuori del ciclo più interno.

Anche *se* scrivere l'assembler a mano volesse dire ottenere un codice più efficiente, i rischi legati a questa attività sono la quantità di tempo impiegata nella codifica e nel debug, la perdita di portabilità e la difficoltà di manutenzione del codice. Uno dei pochi assiomi ampiamente accettati dell'ingegneria del software è che il tempo di codifica è proporzionale alle linee di codice ed ovviamente un programma assembler contiene più linee di un programma C. Una volta codificato, il rischio è che il programma diventi molto diffuso: la vita di questi programmi è più lunga del previsto, il che significa che qualcuno dovrà necessariamente aggiornare il codice ad anni di distanza e farlo funzionare con nuove versioni del sistema operativo o su nuovi modelli di calcolatori. Scrivere in linguaggio ad alto livello, invece che in assembler, non solo permette ai compilatori futuri di ottimizzare il codice per i calcolatori del futuro, ma rende anche il software facile da mantenere ed il programma portabile su architetture diverse.

Trabocchetto: dimenticare che l'indirizzo di parole consecutive negli elaboratori con indirizzamento a byte non differisce di una unità.

Molti programmatori assembler hanno faticato a lungo su errori fatti assumendo che l'indirizzo della parola successiva fosse ottenibile incrementando l'indirizzo

Linguaggio	Tempo
Assembler	37,9 secondi
C	25,3 secondi

Figura 3.36 Confronto fra le prestazioni delle versioni in linguaggio C ed assembler delle procedure `ordina` **e** `scambia` **analizzate nel Paragrafo 3.10.** La dimensione del vettore da ordinare è stata aumentata a 10000 elementi. I programmi sono stati eseguiti su un sistema DEC 5900 con 128 MB di memoria principale, processore R3000 a 40 MHz e versione 4.2a (revisione 47) del sistema operativo Ultrix. Il compilatore ha utilizzato l'opzione -O.

contenuto nel registro di una unità, invece che della dimensione della parola in byte. Uomo avvisato mezzo salvato!

Trabocchetto: usare il puntatore ad una variabile locale al di fuori della procedura in cui è definita.

Un errore comune quando si ha a che fare con i puntatori consiste nel passare come risultato di una procedura il puntatore ad un vettore locale alla procedura stessa. Secondo la filosofia che regola l'utilizzo dello stack (vedere figura 3.12), la memoria che conteneva il vettore locale probabilmente sarà riutilizzata non appena terminata la procedura. I puntatori a variabili locali possono generare il caos.

3.14 Note conclusive

Meno è di più.

Robert Browning, *Andrea del Sarto*, 1855

I calcolatori a programma memorizzato seguono due principi fondamentali: le istruzioni sono indistinguibili dai numeri e la memoria in cui sono scritti i programmi può essere modificata. Questi principi fanno sì che lo stesso calcolatore possa aiutare nella propria attività scienziati che si occupano dell'ambiente, operatori finanziari e romanzieri. La scelta del set delle istruzioni che il calcolatore è in grado di comprendere rappresenta un delicato compromesso fra il numero di istruzioni necessarie ad eseguire un programma, il numero di cicli di clock per istruzione e il periodo di clock. Quattro principi guida aiutano gli autori del set di istruzioni a svolgere il loro delicato compito:

1. *Semplicità e regolarità sono strettamente correlate.* La ricerca della regolarità è alla base di molte delle caratteristiche del set di istruzioni MIPS: stessa dimensione per tutte le istruzioni, sempre tre operandi di tipo registro nelle istruzioni aritmetiche, campi registro sempre nelle stesse posizioni all'interno di tutti i formati delle istruzioni.
2. *Minori sono le dimensioni, maggiore è la velocità.* La velocità di esecuzione è il motivo per cui il MIPS ha 32 registri anziché molti di più.
3. *Un buon progetto richiede buoni compromessi.* Un esempio relativo al MIPS è il compromesso che permette di specificare indirizzi e costanti di grandi dimensioni, mantenendo la stessa lunghezza per tutte le istruzioni.
4. *Rendere veloce l'evento più frequente.* Esempi di questo tipo che riguardano il MIPS includono il modo di indirizzamento PC-relative per i salti condizionati e l'indirizzamento immediato per gli operandi costanti.

Al di sopra del livello della macchina c'è il linguaggio assembler, che gli uomini sono in grado di leggere; l'assemblatore traduce questa notazione nei numeri binari comprensibili all'elaboratore ed "estende" persino il set di istruzioni creando istruzioni simboliche non implementate a livello hardware. Ad esempio le costanti o gli indirizzi troppo grandi sono suddivisi in frammenti di dimensione adeguata, i formati più frequenti di determinate istruzioni hanno un nome che li contraddistingue e così via. Le istruzioni MIPS viste finora (sia le istruzioni vere e proprie che le pseudoistruzioni) sono elencate in figura 3.37.

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
somma	add	R	move	move	R
sottrazione	sub	R	moltiplicazione	mult	R
somma immediata	addi	I	moltiplicazione immediata	multi	I
load word	lw	I	load immediata	li	I
store word	sw	I	branch less than	blt	I
load byte	lb	I	branch less than or equal	ble	I
store byte	sb	I	branch greater than	bgt	I
load upper immediato	lui	I	branch greater than or equal	bge	I
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediato	slti	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

Figura 3.37 Il set delle istruzioni MIPS esaminate finora, con a sinistra le istruzioni vere e proprie e a destra le pseudoistruzioni. L'Appendice A (Paragrafo A.10) descrive l'intera architettura MIPS. La figura 3.18 mostra ulteriori dettagli dell'architettura presentata in questo capitolo.

Classe di istruzioni	Esempi MIPS	Corrispondenza LAL	Frequenza	
			gcc	spice
Aritmetiche	add, sub, addi	operazioni nelle assegnazioni	48%	50%
Trasferimento dati	lw, sw, lb, sb, lui	riferimento a strutture dati, come i vettori	33%	41%
Salti condizionati	beq, bne, slt, slti	costrutti if e cicli	17%	8%
Salti incondizionati	j, jr, jal	chiamate a procedura, ritorno e costrutti case/switch	2%	1%

Figura 3.38 Classi di istruzioni MIPS, esempi, corrispondenza con i costrutti dei linguaggi di programmazione ad alto livello (LAL) e percentuale di istruzioni MIPS eseguite in due programmi, gcc e spice, per classe di appartenenza. La figura 4.54 mostra la percentuale delle singole istruzioni MIPS eseguite.

Non tutte le istruzioni sono uguali: alcune sono usate molto più frequentemente di altre. Ad esempio la figura 3.38 mostra l'utilizzo di ciascuna classe di istruzioni in due programmi, gcc e spice. La diversa percentuale di utilizzo delle istruzioni gioca un ruolo importante sulle prestazioni, sull'unità di elaborazione, sull'unità di controllo e sulla pipeline.

Le varie classi di istruzioni MIPS sono associate a costrutti dei linguaggi di programmazione ad alto livello:

- Le istruzioni aritmetiche corrispondono alle operazioni all'interno delle assegnazioni.
- Le istruzioni di trasferimento dati sono più frequenti quando si utilizzano strutture dati come i vettori o le strutture.
- Le istruzioni di salto condizionato sono usate nei costrutti *if* e nei cicli.
- Le istruzioni di salto incondizionato sono usate nelle chiamate a procedura e nel ritorno e nei costrutti *case/switch*.

Il Capitolo 4 presenta un'altra parte del set di istruzioni MIPS, dopo una spiegazione sull'aritmetica dei calcolatori.

3.15 Inquadramento storico e approfondimenti

accumulatore: Termine arcaico per registro. L'utilizzo del termine come sinonimo di "registro" è un indicatore attendibile del fatto che l'utente è sulla scena da parecchio tempo.

Eric Raymond, *The New Hacker's Dictionary*, 1991

Le architetture ad accumulatore

Nei primi calcolatori stored-program l'hardware era molto prezioso e di conseguenza i pionieri dell'informatica non potevano permettersi il numero di registri dei calcolatori di oggi: questi elaboratori avevano un solo registro per le istruzioni aritmetiche. Dato che in quell'unico registro venivano accumulati tutti i risultati delle operazioni esso venne chiamato *accumulatore* e lo stesso nome fu dato al set di istruzioni di quei calcolatori. Ad esempio l'EDSAC (1949) aveva un unico accumulatore.

Il formato a tre operandi delle istruzioni MIPS suggerisce che avendo un solo registro ne mancano come minimo due. Utilizzare l'accumulatore *sia* come operando sorgente *che* come operando destinazione dimezza il problema, ma lascia comunque scoperto un operando. L'ultimo operando è in memoria: i calcolatori ad accumulatore utilizzano il modo di indirizzamento in memoria descritto precedentemente. Ne segue che l'istruzione di somma di un set di istruzioni ad accumulatore è del tipo

```
add 200
```

Questa istruzione indica che viene eseguita la somma del contenuto dell'accumulatore con la parola che ha indirizzo di memoria 200 e che il risultato viene nuovamente memorizzato nell'accumulatore. Non è specificato nessun registro poiché è implicito che l'accumulatore sia uno degli operandi sorgente e l'operando destinazione dell'operazione.

Esempio

Compilare un'assegnazione in istruzioni ad accumulatore

In un'architettura ad accumulatore, quale sarebbe è il codice assembler corrispondente a questa istruzione C?

```
A = B + C;
```

Soluzione

L'assegnazione sarebbe tradotta nelle seguenti linee di codice appartenenti al set di istruzioni di un'architettura ad accumulatore:

```
load  IndirizzoB # Acc = Memoria[IndirizzoB], o Acc = B
add   IndirizzoC # Acc = B + Memoria[IndirizzoC], o Acc = B + C
store IndirizzoA # Memoria[IndirizzoA] = Acc, o A = B + C
```

Nei calcolatori ad accumulatore tutte le variabili dei programmi sono allocate in memoria e non nei registri come accadeva nel caso del MIPS; è come se in questo tipo di calcolatori le variabili fossero sempre riversate in memoria. Come è facile immaginare, nelle architetture a singolo accumulatore occorrono molte più istruzioni per eseguire i programmi (vedere l'Esercizio 3.19 per un altro esempio).

Il passo successivo nell'evoluzione del set di istruzioni fu l'aggiunta di registri dedicati ad operazioni specifiche. Potevano essere inclusi registri che fungevano da indice per l'accesso a vettori durante l'esecuzione di istruzioni di trasferimento dati, da accumulatore aggiuntivo per istruzioni di moltiplicazione o divisione o da puntatore alla cima dello stack. Forse l'esempio più noto di questo tipo di set di istruzioni è l'Intel 8086, il processore che ha costituito l'anima del Personal Computer IBM. Questo tipo di set di istruzioni prende il nome di *extended accumulator* (ad accumulatore esteso), *dedicated register* (a registri dedicati) o *special-purpose register* (a registri special-purpose). Nelle istruzioni aritmetiche uno degli operandi può essere in memoria, come nel caso dei calcolatori ad accumulatore singolo, ma ci sono altre istruzioni in cui tutti gli operandi sono registri, come nel caso dell'architettura MIPS.

Le architetture a registri general-purpose

La generalizzazione del concetto di registro dedicato conduce all'idea di introdurre registri che possano essere usati per qualunque tipo di funzione; da qui deriva l'espressione *general-purpose register*. Il MIPS è un esempio di questo tipo. Questi set di istruzioni possono essere ulteriormente suddivisi in due categorie: quelli che consentono che uno degli operandi si trovi in memoria, come avveniva nei calcolatori ad accumulatore, sono chiamati architetture *registro-memoria* (register-memory) e quelli che richiedono che gli operandi siano sempre nei registri sono chiamati architetture *load-store* o *registro-registro* (register-register). La figura 3.39 mostra come è evoluto il numero di registri all'interno di alcuni calcolatori di uso molto diffuso.

Calcolatore	Numero di registri general-purpose	Tipo di architettura	Anno
EDSAC	1	accumulatore	1949
IBM 701	1	accumulatore	1953
CDC 6600	8	load-store	1963
IBM 360	16	registro-memoria	1964
DEC PDP-8	1	accumulatore	1965
DEC PDP-11	8	registro-memoria	1970
Intel 8008	1	accumulatore	1972
Motorola 6800	2	accumulatore	1974
DEC VAX	16	registro-memoria, memoria-memoria	1977
Intel 8086	1	accumulatore esteso	1978
Motorola 68000	16	registro-memoria	1980
Intel 80386	8	registro-memoria	1985
MIPS	32	load-store	1985
HP PA-RISC	32	load-store	1986
SPARC	32	load-store	1987
PowerPC	32	load-store	1992
DEC Alpha	32	load-store	1992

Figura 3.39 Andamento negli anni del numero dei registri general-purpose all'interno di elaboratori di uso molto diffuso.

Il primo calcolatore di tipo load-store fu il CDC 6600 nel 1963, considerato da molti come il primo supercalcolatore. Il MIPS è un esempio più recente di architettura load-store.

L'80386 è il tentativo dell'Intel di trasformare l'80x86 in un processore con un set di istruzioni general-purpose e registro-memoria. Forse il primo esempio di set di istruzioni registro-memoria è l'architettura IBM 360, introdotta nel 1964; esso rappresenta ancora il cuore dei mainframe IBM, a cui si deve gran parte dell'attività commerciale della più grande azienda di calcolatori al mondo. Le architetture registro-memoria furono le più diffuse negli anni '60 e nella prima metà degli anni '70.

Nel 1977 l'architettura VAX della Digital Equipment Corporation rappresentò un altro passo in avanti, permettendo l'utilizzo all'interno delle istruzioni di qualsiasi combinazione di operandi nei registri o in memoria. Il tipo di architettura in cui tutti gli operandi sono in memoria si chiama *memoria-memoria*; in realtà il set di istruzioni VAX, come tutti quelli fino all'IBM 360, è un ibrido, avendo anche i registri general-purpose.

Compilare un'assegnazione in istruzioni memoria-memoria

Esempio Quale sarebbe il codice assembler memoria-memoria corrispondente a questa istruzione C?

```
A = B + C;
```

Soluzione L'assegnazione sarebbe tradotta nella seguente istruzione di un set di tipo memoria-memoria:

```
add     IndirizzoA, IndirizzoB, IndirizzoC
```

Si veda l'Esercizio 3.19 per un altro esempio.

Mentre il MIPS ha una sola istruzione di somma con operandi a 32 bit, l'Intel 80x86 ne ha diverse, per specificare se un operando si trova in un registro o in memoria; in più è possibile accedere all'operando in memoria con più di sette modi di indirizzamento. Le combinazioni fra i modi di indirizzamento e gli operandi registro-memoria fanno sì che nell'80x86 esistano decine di varianti dell'istruzione di somma; chiaramente tutta questa varietà rende l'implementazione hardware dell'80x86 molto più impegnativa.

Codice compatto e architetture a stack

Quando la memoria era scarsa era molto importante che i programmi fossero di piccole dimensioni; calcolatori come l'Intel 8086, l'IBM 360 o il VAX hanno una lunghezza variabile delle istruzioni, da un lato per adattarsi alle diverse modalità di specifica degli operandi, dall'altro per minimizzare le dimensioni del codice. Le istruzioni Intel 80x86 possono essere lunghe da 1 a 17 byte, quelle IBM 360 2, 4 o 6 byte, quelle VAX un qualunque numero di byte compreso fra 1 e 54. Se lo spazio di memoria dedicato alle istruzioni dovesse nuovamente diventare una risorsa preziosa, architetture come queste ritornerebbero popolari.

Negli anni '60 alcune aziende seguirono un approccio drastico nei confronti del set di istruzioni: credendo che fosse troppo difficile per i compilatori utilizzare i registri in modo efficiente, ne abbandonarono del tutto l'uso! Il set di istruzioni si basava su un modello di esecuzione *a stack*, come quello delle vecchie calcolatrici Hewlett-Packard. Gli operandi erano trasferiti dalla memoria allo stack o dallo stack alla memoria e le operazioni cercavano gli operandi e mettevano i risultati nello stack. Oltre a semplificare i compilatori per il fatto di eliminare l'allocazione delle variabili nei registri, i calcolatori a stack si prestavano a compattare la codifica delle istruzioni, rendendo la scarsità di memoria una scusa non più valida per non programmare ad alto livello.

Compilare un'assegnazione in istruzioni a stack

Esempio In un'architettura a stack, quale sarebbe il codice assembler corrispondente a questa istruzione C?

```
A = B + C;
```

Soluzione In un'architettura a stack l'assegnazione sarebbe tradotta nelle seguenti istruzioni:

```
push IndirizzoC # Top=Top+4; Stack[Top]=Memoria[IndirizzoC]
push IndirizzoB # Top=Top+4; Stack[Top]=Memoria[IndirizzoB]
add             # Stack[Top-4]= Stack[Top]+ Stack[Top-4];
                # Top=Top-4
pop  IndirizzoA # Memoria[IndirizzoA]=Stack[Top]; Top=Top-4
```

Per ottenere l'indirizzo corretto in byte, lo stack viene aggiornato a multipli di 4. Il lato negativo dello stack, confrontato con i registri, è che è difficile riutilizzare i dati prelevati o calcolati senza accedere ripetutamente alla memoria (vedere l'Esercizio 3.19 per un altro esempio).

Lo spazio di memoria può essere nuovamente considerato una risorsa preziosa per il tanto annunciato Network Computer (NC), sia perché tale spazio è limitato al fine di mantenere bassi i costi, sia perché i programmi sono scaricati da Internet e più un programma è piccolo meno tempo impiega per essere trasferito. La compattezza del set di istruzioni è quindi un requisito per il Network Computer; queste motivazioni sono state adottate per giustificare la costruzione di un interprete hardware per il linguaggio Java, basato sullo stack. Il tempo dirà se le motivazioni addotte avranno un riscontro tecnico e/o commerciale.

Architetture per linguaggi ad alto livello

Negli anni '60 il software di sistema era raramente scritto in linguaggio ad alto livello; per citare un esempio, praticamente tutti i sistemi operativi commerciali precedenti a Unix erano scritti in assembler e più di recente anche OS/2 fu scritto originariamente a basso livello. Alcuni imputarono il problema alla densità di codice del set di istruzioni, piuttosto che alla tecnologia dei linguaggi di programmazione e dei compilatori.

Venne quindi sostenuta una filosofia di progetto detta architettura dei calcolatori *high-level-language* (per linguaggi ad alto livello), con l'obiettivo di rendere l'hardware più vicino ai linguaggi di programmazione. L'avvento di linguaggi di programmazione e di compilatori più efficienti, insieme all'espansione delle memorie, relegarono questo movimento ad una semplice nota storica. Il Burroughs B500 fu l'unico sbocco commerciale di tale filosofia di progetto e ad oggi non ci sono in commercio esempi significativi derivati da questa idea datata anni '60.

Architetture con set di istruzioni ridotto (RISC)

La filosofia di progetto orientata al linguaggio fu sostituita negli anni '80 da quella dei *RISC* (*reduced instruction set computer*, calcolatori con set di istruzioni ridotto). Miglioramenti nei linguaggi di programmazione, nei compilatori e relativi al costo delle memorie fecero sì che sempre meno codice fosse scritto direttamente in assembler e che quindi la bontà di un set di istruzioni potesse essere misurata da quanto efficacemente fosse sfruttabile da un compilatore piuttosto che da un programmatore umano.

Praticamente tutti i nuovi set di istruzioni sviluppati a partire dal 1982 seguono la filosofia RISC: lunghezza fissa delle istruzioni, set di istruzioni di tipo load-store, modi di indirizzamento ed operazioni limitati. MIPS, Sun SPARC, Hewlett-Packard PA-RISC, IBM PowerPC e DEC Alpha sono tutti esempi di architetture RISC.

Breve storia dell'80x86

I progenitori dell'80x86 furono i primi microprocessori, prodotti intorno alla metà degli anni '70. Gli Intel 4004 e 8008 erano processori ad accumulatore molto semplici, rispettivamente a 4 e a 8 bit. Morse et al. [1980] descrive l'evoluzione dell'8086 a partire dall'8080, verificatasi alla fine degli anni '70, nel tentativo di ottenere un elaboratore a 16 bit con un throughput migliore. In quel periodo quasi tutta la programmazione dei microprocessori era fatta in assembler (sia la memoria che i compilatori scarseggiavano). L'Intel era interessata a mantenere la sua base di utenti 8080, quindi l'8086 fu progettato in modo da essere "compatibile" con l'8080; l'8086 non fu *mai* compatibile con l'8080 a livello di modulo oggetto, ma i calcolatori erano sufficientemente simili da consentire che la conversione fra i due linguaggi assembler fosse fatta in modo automatico.

Nei primi anni '80 l'IBM scelse la versione dell'8086 con il bus esterno da 8 bit, chiamata 8088, per inserirla nel suo PC IBM; la versione a 8 bit fu preferita per mantenere basso il costo del sistema. Questa scelta, unita allo straordinario successo del PC IBM, ha reso l'architettura 8086 onnipresente. Il successo del PC fu in parte dovuto al fatto che l'IBM rese la sua architettura aperta, rendendo fiorente l'industria dei cloni. Come discusso nel Paragrafo 3.12, l'80286, l'80386, l'80486, il Pentium ed il Pentium Pro hanno esteso l'architettura e fornito una serie di miglioramenti delle prestazioni.

Benché il Motorola 68000 fosse stato scelto per il Macintosh, il Mac non invase mai il mercato allo stesso modo del PC (in parte perché la Apple non consentì la produzione di cloni Mac basati sul 68000) ed il 68000 non acquisì mai la stessa influenza sul software di cui gode l'8086. Il Motorola 68000 può anche essere *tecnicamente* migliore dell'8086, ma l'impatto della scelta dell'IBM e la strategia di architettura aperta perseguita, nel mercato hanno avuto ragione delle qualità tecniche del 68000.

Alcuni sostengono che la scarsa eleganza del set di istruzioni dell'80x86 sia inevitabile e rappresenti il prezzo da pagare per il dilagante successo di qualunque architettura. Gli autori non condividono questa opinione: ovviamente nessuna architettura di successo può buttare a mare le caratteristiche introdotte nelle implementazioni precedenti e dopo un certo tempo alcune di esse possono diventare indesiderabili. La mancanza di eleganza dell'80x86 ha già origine dal set di istruzioni dell'8086 ed è esacerbata dalle espansioni architetturalmente incoerenti che si trovano nell'8087, nell'80286, nell'80386 e nell'MMX.

Un esempio opposto è l'architettura dell'IBM 360/370, molto più vecchia dell'80x86, che ha dominato il mercato dei mainframe proprio come l'80x86 ha dominato quello del personal computer. Il suo set istruzioni, senza dubbio grazie a fondamenta più solide e a sviluppi più attenti alla compatibilità, anche dopo 30 anni dalla prima implementazione è decisamente più sensato di quello dell'80x86.

Hewlett-Packard e Intel annunceranno un nuovo set di istruzioni comune nel 1998; anch'esso sarà compatibile verso l'alto con l'80x86, così il set di istruzioni dell'80x86 sarà in qualche modo presente anche nei calcolatori del prossimo secolo.

Gli antropologi informatici del 21esimo secolo che analizzeranno il set di istruzioni di questi calcolatori, sfogliandolo strato dopo strato scopriranno i manufatti a partire dal primo microprocessore. Con queste scoperte, come giudicheranno l'architettura degli elaboratori del 20esimo secolo?

Letture consigliate

Bayko, J. [1996]. "Great Microprocessors of the Past and Present," disponibile all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*.
Un punto di vista personale nella storia di microprocessori tipici e insoliti, dall'Intel 4004 al Patriot Scientific ShBoom!

Kane, G., e J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ.
Il libro descrive l'architettura MIPS in maggior dettaglio rispetto all'Appendice A.

Levy, H., e R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX*, Digital Press, Boston.
Il libro si concentra sul VAX, ma include anche descrizioni dell'Intel 80x86, dell'IBM 360 e del CDC 6000.

Morse, S., B. Ravenal, S. Mazor, e W. Pohlman [1980]. "Intel Microprocessors – 8080 to 8086," *Computer* 13:10 (Ottobre).
La storia delle architetture Intel dal 4004 all'8086, secondo le persone che hanno partecipato al loro progetto.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming*, Wiley, New York.
Il libro è focalizzato sul Motorola 68x00, ma include anche l'Intel 8086, il Motorola 6809, il TI 9900, e lo Zilog Z8000.

3.16 Parole chiave

I termini elencati nel seguito riflettono le idee chiave discusse in questo capitolo. Coloro che non sono sicuri del significato dei termini sono invitati a fare riferimento al Glossario, in cui compaiono le definizioni complete.

blocco base
calcolatore load-store o registro-registro
calcolatore a programma memorizzato
chiamante
chiamato
codice operativo
concetto di programma memorizzato
file eseguibile
finestra di attivazione
formato delle istruzioni
frame della procedura
frame pointer

global pointer
indirizzamento di tipo registro
indirizzamento immediato
indirizzamento PC-relative
indirizzamento tramite base o spiazzamento
indirizzo
indirizzo di ritorno
istruzione di trasferimento dati
istruzione jump-and-link
linker o link editor
loader
modo di indirizzamento
parola
procedura
program counter (PC)
programma oggetto
pseudoistruzione
puntatore allo stack
registro general-purpose (GPR)
salto condizionato
set di istruzioni
stack
tabella degli indirizzi di salto

3.17 Esercizi

L'Appendice A descrive il simulatore MIPS, utile per svolgere questi esercizi. Benché il simulatore accetti le pseudoistruzioni, si cerchi di non usarle in tutti gli esercizi che richiedono di produrre codice MIPS. L'obiettivo è quello di imparare il vero set di istruzioni MIPS ed il conteggio delle istruzioni dovrebbe riflettere il numero di istruzioni realmente eseguite, non le pseudoistruzioni.

Ci sono però casi in cui è necessario usare le pseudoistruzioni, ad esempio l'istruzione `la` quando il valore reale non è conosciuto in fase di codifica assembler; in molti casi esse sono piuttosto comode ed il risultato è un codice più leggibile (ad esempio nel caso delle istruzioni `li` e `move`). Nel caso il lettore scelga di usare le pseudoistruzioni per questi motivi, è invitato ad aggiungere dei commenti alla soluzione per spiegare quali pseudoistruzioni ha usato e perché.

3.1 [5] <§§3.3, 3.5, 3.8> Aggiungere i commenti al seguente codice MIPS e descrivere in una frase che cosa calcola. Si assuma che `$a0` sia usato per l'ingresso dei dati e che inizialmente contenga `n`, un intero positivo. Si assuma che `$v0` sia usato per l'uscita.

```
inizio:  addi    $t0, $zero, 0
         addi    $t1, $zero, 1
ciclo:   slt     $t2, $a0, $t1
         bne     $t2, $zero, fine
         add     $t0, $t0, $t1
         addi    $t1, $t1, 2
         j       ciclo
fine:    add $v0, $t0, $zero
```

3.2 [12] <§§3.3, 3.5, 3.8> Il seguente frammento di codice lavora su un vettore e genera due valori rilevanti nei registri `$v0` e `$v1`. Si assuma che il vettore sia lungo 5000 parole, con l'indice che varia da 0 a 4999, che il suo indirizzo di base si trovi in `$a0` e la sua dimensione (5000) in `$a1`. Descrivere in una frase che cosa esegue questo codice ed in particolare che cosa viene restituito in `$v0` e `$v1`.

```
         add     $a1, $a1, $a1
         add     $a1, $a1, $a1
         add     $v0, $zero, $zero
         add     $t0, $zero, $zero
esterno: add     $t4, $a0, $t0
         lw      $t4, 0($t4)
         add     $t5, $zero, $zero
         add     $t1, $zero, $zero
```



```
interno: add     $t3, $a0, $t1
         lw      $t3, 0($t3)
         bne     $t3, $t4, salta
         addi    $t5, $t5, 1
salta:   addi    $t1, $t1, 4
         bne     $t1, $a1, interno
         slt     $t2, $t5, $v0
         bne     $t2, $zero, avanti
         add     $v0, $t5, $zero
         add     $v1, $t4, $zero
avanti:  addi    $t0, $t0, 4
         bne     $t0, $a1, esterno
```

3.3 [10] <§§3.3, 3.5, 3.8> Si assuma che il codice dell'Esercizio 3.2 sia eseguito su un calcolatore con frequenza di clock pari a 500 MHz, che richiede il seguente numero di cicli per ogni istruzione:

Istruzione	Cicli
add, addi, slt	1
lw, bne	2

Nel caso peggiore, in quanti secondi sarà eseguito il codice?

3.4 [5] <§3.8> Scrivere la più breve sequenza di istruzioni MIPS che codifica la seguente espressione C:

```
a = b + 100;
```

Si assuma che `a` corrisponda al registro `$t0` e `b` a `$t1`.

3.5 [10] <§3.8> Scrivere la più breve sequenza di istruzioni MIPS che codifica la seguente espressione C:

```
x[10] = x[11] + c;
```

Si assuma che `c` corrisponda al registro `$t0` e che il vettore `x` abbia l'indirizzo di base pari a $4.000.000_{dieci}$.

3.6 [10] <§§3.3, 3.5, 3.8> Il seguente programma cerca di copiare dati di tipo parola dall'indirizzo memorizzato in `$a0` all'indirizzo che si trova in `$a1`, scrivendo il numero di parole copiate nel registro `$v0`. Il programma termina quando viene trovata una parola che ha valore 0. Non è necessario preservare il contenuto dei registri `$v1`, `$a0` e `$a1`. L'ultima parola deve essere copiata ma non contata.

```
ciclo: lw   $v1, 0($a0)        # legge la parola successiva dal
                               # sorgente
       addi $v0, $v0, 1        # incrementa il contatore delle
                               # parole copiate
       sw   $v1, 0($a1)        # scrive la parola nella
                               # destinazione
       addi $a0, $a0, 1        # fa avanzare il puntatore al
                               # prossimo sorgente
       addi $a1, $a1, 1        # fa avanzare il puntatore alla
                               # prossima destinazione
       bne  $v1, $zero, ciclo  # va a ciclo se la parola copiata
                               # è diversa da 0
```

Ci sono molti errori in questo programma MIPS: correggerli e scrivere il programma nella versione corretta. Come nel caso di molti altri esercizi di questo capitolo, il modo più semplice per scrivere programmi MIPS è usare il simulatore descritto in Appendice A. Una copia di questo programma si può trovare all'indirizzo *www.mkp.com/cod2e.htm*.

3.7 [15] <§3.8> Utilizzando il programma MIPS dell'Esercizio 3.6 (errori compresi), determinare il formato di ciascuna delle istruzioni ed i valori decimali di tutti i campi delle istruzioni.

3.8 [10] <§§3.2, 3.3, 3.5, 3.8> {Es. 3.7} A partire dal programma corretto ottenuto risolvendo l'Esercizio 3.6, scrivere il frammento di codice C che potrebbe aver generato tale codice. Si assuma che la variabile `sorg` corrisponda al registro `$a0`, la variabile `dest` al registro `$a1` e la variabile `cont` al registro `$v0`. Si mostri la dichiarazione delle variabili, ma si assuma che `sorg` e `dest` siano già state inizializzate con gli indirizzi opportuni.

3.9 [10] <§3.5> Il frammento C

```
while (salva[i] == k)
  i = i + j;
```

che compare nel quarto esempio del Paragrafo 3.5 utilizza due salti all'interno del ciclo, uno condizionato ed uno incondizionato. Solo compilatori molto limitati produrrebbero codice di questo tipo. Riscrivere il codice assembler in modo tale che ad ogni iterazione venga eseguito al più un salto (condizionato o incondizionato). Quante istruzioni sono eseguite, prima e dopo l'ottimizzazione, nel caso in cui il numero di iterazioni è 10 (cioè nel caso in cui `salva[i+10*j]` è uguale a `k` e `salva[i], ..., salva[i+9*j]` sono diversi da `k`)?

3.10 [25] <§3.9> Come discusso nel Paragrafo 3.9 e riassunto in figura 3.37, le pseudoistruzioni non fanno parte del set di istruzioni MIPS, ma appaiono frequentemente nei programmi MIPS. Per ciascuna delle pseudoistruzioni della seguente tabella riportare la più breve sequenza di istruzioni MIPS che produce lo stesso risultato. In alcuni casi può essere necessario usare il registro `$at`. Nella tabella, `grande` si riferisce ad un numero rappresentabile su 32 bit e `piccolo` ad un numero su 16 bit.

Pseudoistruzione	Risultato
`move $t5, $t3`	`$t5 = $t3`
`clear $t5`	`$t5 = 0`
`li $t5, piccolo`	`$t5 = piccolo`
`li $t5, grande`	`$t5 = grande`
`lw $t5, grande($t3)`	`$t5 = Memoria[$t5 + grande]`
`addi $t5, $t3, grande`	`$t5 = $t3 + grande`
`beq $t5, piccolo, L`	`if ($t5 = piccolo) go to L`
`beq $t5, grande, L`	`if ($t5 = grande) go to L`
`ble $t5, $t3, L`	`if ($t5 <= $t3) go to L`
`bgt $t5, $t3, L`	`if ($t5 > $t3) go to L`
`bge $t5, $t3, L`	`if ($t5 >= $t3) go to L`

3.11 [30] <§3.5> Si consideri il seguente frammento di codice C:

```
for (i=0; i<=100; i=i+1) { a[i] = b[i] + c; }
```


Si assuma che `a` e `b` siano vettori di parole, che l'indirizzo di base di `a` sia in `$a0` e che l'indirizzo di base di `b` sia in `$a1`; inoltre il registro `$t0` è associato alla variabile `i` ed il registro `$s0` a `c`. Si scriva il codice MIPS corrispondente. Quante istruzioni sono eseguite? Quanti sono i riferimenti ai dati in memoria durante l'esecuzione?

3.12 [5] <§§3.8, 3.9> Tenendo presente il modo di indirizzamento PC-relative, spiegare perché potrebbe essere problematico per un assemblatore implementare direttamente l'istruzione di salto condizionato presente nel seguente frammento di codice:

```
qui:  beq  $t1, $t2, là
...
là:   add  $t1, $t1, $t1
```

Mostrare come l'assemblatore dovrebbe riscrivere questa sequenza di codice per risolvere il problema.

3.13 [10] <§3.12> Si consideri un'architettura simile a quella MIPS, che però supporti anche l'indirizzamento di tipo update (come il PowerPC) per le istruzioni di trasferimento dati. Eseguendo il programma gcc con questa architettura, una certa percentuale delle istruzioni di trasferimento dati mostrate in figura 3.38 potrebbe usare le nuove istruzioni e per ciascuna delle modifiche potrebbe essere eliminata un'istruzione aritmetica. Se il 25% delle istruzioni potessero essere modificate, quale delle due architetture consentirebbe un tempo di esecuzione inferiore per il gcc, la MIPS modificata o quella originale? E di quanto inferiore? Si assuma che entrambe le architetture abbiano i valori di CPI riportati nell'Esercizio 3.16 e che l'architettura modificata abbia una durata del ciclo di clock maggiore del 10% per consentire le nuove istruzioni.

3.14 [10] <§3.14> Quando si progetta un sistema di memoria, diventa utile conoscere la frequenza con cui avvengono le operazioni di lettura rispetto a quelle di scrittura e la frequenza di accesso alle istruzioni rispetto alla frequenza di accesso ai dati. Usando le informazioni relative alla combinazione delle istruzioni MIPS per il programma gcc riportate in figura 3.38, si trovino:

a. La percentuale di *tutti* gli accessi in memoria che sono relativi ai dati (rispetto alle istruzioni).

b. La percentuale di *tutti* gli accessi in memoria che avvengono in lettura (rispetto a quelli in scrittura). Si assuma che due terzi dei trasferimenti dati siano load.

3.15 [10] <§3.14> Eseguire gli stessi calcoli dell'Esercizio 3.14, ma relativi al programma spice invece di gcc.

3.16 [15] <§3.14> Si supponga di aver ottenuto le seguenti misure per quanto riguarda il CPI medio delle istruzioni:

Istruzione	CPI medio
Aritmetiche	1,0 cicli di clock
Trasferimento dati	1,4 cicli di clock
Salti condizionati	1,7 cicli di clock
Salti incondizionati	1,2 cicli di clock

Si calcoli l'effettivo CPI del MIPS. Per ottenere la combinazione delle istruzioni si faccia la media della frequenza delle istruzioni nei programmi gcc e spice, riportate in figura 3.38.

3.17 [20] <§3.10> In questo esercizio verranno esaminati in modo quantitativo i pro e i contro di aggiungere un modo di indirizzamento al MIPS che consenta alle istruzioni aritmetiche di accedere direttamente in memoria, come avviene nell'80x86. Il principale beneficio è un minor numero di istruzioni eseguite, dato che non sarebbe necessario il caricamento preliminare di un registro. Il principale svantaggio è che il ciclo di clock dovrebbe avere durata maggiore, per tener conto del tempo addizionale di lettura dalla memoria. Si consideri l'aggiunta di una nuova istruzione:

```
addm   $t2, 100($t3)   # $t2 = $t2 + Memoria[$t3+100]
```

Si assuma che la nuova istruzione provochi un aumento della durata del ciclo di clock pari al 10%. Si utilizzi la frequenza delle istruzioni relativa al benchmark gcc, riportata in figura 3.38, e si assuma che due terzi dei trasferimenti dati siano load ed i restanti store; si assuma inoltre che l'aggiunta della nuova istruzione influisca solo sulla velocità del clock e non sul CPI. Quale percentuale di load deve essere eliminata, perché l'elaboratore con la nuova istruzione abbia almeno le stesse prestazioni?

3.18 [10] <§3.10> Usando le informazioni dell'Esercizio 3.17, scrivere una sequenza di istruzioni in cui una load di `$t0`, immediatamente seguita dall'utilizzo di `$t0` (ad esempio in una somma), non possa essere sostituita da una singola istruzione del tipo proposto.

In maggior dettaglio

Confrontare set di istruzioni di tipo diverso

Nei prossimi due esercizi, il compito assegnato sarà quello di confrontare l'efficienza di utilizzo della memoria da parte di quattro tipi diversi di set di istruzioni nel caso di due sequenze di codice. I set di istruzioni sono i seguenti:

- *Ad accumulatore.*
- *Memoria-memoria*: Tutti e tre gli operandi di ogni istruzione sono in memoria.
- *A stack*: Tutte le operazioni coinvolgono la cima dello stack. Gli accessi in memoria sono solo di tipo push e pop e tutte le altre istruzioni prelevano gli operandi dallo stack sostituendoli con il risultato. L'implementazione usa uno stack per gli ultimi due dati; gli accessi ad altre posizioni dello stack corrispondono a riferimenti alla memoria.
- *Load-store*: Tutte le operazioni sono fatte sui registri e le operazioni registro-registro hanno tre operandi. Ci sono 16 registri general-purpose e l'identificatore del registro è lungo 4 bit.

Si consideri il seguente codice C:

```
a = b + c;        #a, b e c sono variabili in memoria
```

Il Paragrafo 3.15 riporta il codice assembler equivalente per i diversi tipi del set di istruzioni. Data una sequenza di codice, si possono calcolare i byte relativi alle istruzioni prelevate dalla memoria ed i byte di dati trasferiti da e verso la memoria, facendo le seguenti assunzioni valide per tutti e quattro i set di istruzioni:

- Il codice operativo è sempre lungo 1 byte (8 bit).
- Tutti gli indirizzi di memoria sono lunghi 2 byte (16 bit).
- Tutti gli operandi sono lunghi 4 byte (32 bit).
- Tutte le istruzioni hanno dimensioni pari ad un numero intero di byte.
- Non ci sono ottimizzazioni volte a ridurre l'accesso alla memoria.

Ad esempio il caricamento di un registro richiede un'istruzione di quattro byte (uno per il codice operativo, uno per il registro destinazione e due per l'indirizzo di memoria) da prelevare dalla memoria, più quattro byte di dati. Un'istruzione di somma di tipo memoria-memoria richiede invece sette byte di istruzione (uno per il codice operativo e due per ciascuno dei tre indirizzi di memoria) da prelevare dalla memoria ed il trasferimento di 12 byte da o verso la memoria (otto dalla memoria al processore e quattro dal processore alla memoria). La seguente tabella mostra un riassunto di queste informazioni per ciascuno degli stili di architettura che compaiono sopra e nel Paragrafo 3.15:

Stile	Istruzioni per a = b+c	Byte di codice	Byte di dati
Accumulatore	3	3+3+3	4+4+4
Memoria-memoria	1	7	12
Stack	4	3+3+1+3	4+4+0+4
Load-store	4	4+4+3+4	4+4+0+4

3.19 [20] <§3.15> Per il seguente codice C, scrivere un programma equivalente in linguaggio assembler per ciascuno dei tipi di architettura (si assuma che inizialmente tutte le variabili siano in memoria):

```
a = b + c;
b = a + c;
d = a - b;
```

Per ciascuna sequenza di codice calcolare il numero di byte di istruzioni prelevate dalla memoria ed i byte di dati trasferiti (in lettura o in scrittura). Quale delle architetture è più efficiente, tenendo conto delle dimensioni del codice? Quale tenendo conto della banda totale di memoria richiesta (codice + dati)? Se la risposta non è la stessa, perché questa diversità?

3.20 [5] <§3.15> Talvolta le architetture sono caratterizzate dal numero tipico di indirizzi di memoria contenuti nelle istruzioni. I numeri più comuni sono 0, 1, 2 e 3 indirizzi per istruzione: si associno questi valori a ciascuna delle categorie.

3.21 [10] <§3.7> Calcolare il valore decimale dei byte che costituiscono la rappresentazione della seguente stringa ASCII (terminata dal carattere Null):

```
Un byte misura 8 bit
```

3.22 [30] <§§3.6, 3.7> Scrivere un programma in linguaggio assembler MIPS che converta una stringa ASCII decimale in un numero intero. Il programma deve aspettarsi che il registro `$a0` contenga l'indirizzo di partenza di una stringa termi-

nata dal carattere Null, composta da una combinazione di cifre da 0 a 9. Il programma deve calcolare il valore intero equivalente alla stringa e metterlo nel registro `$v0`. Il programma non deve gestire i numeri negativi. Se in qualunque punto della stringa compare un carattere diverso da una cifra, il programma deve terminare, restituendo il valore -1 nel registro `$v0`. Se ad esempio il registro `$a0` punta alla sequenza dei tre byte 50_{dieci}, 52_{dieci}, 0_{dieci}, (cioè la stringa "24" terminata dal carattere Null), quando il programma termina il registro `$v0` deve contenere il valore 24_{dieci} (il pedice "dieci" indica la base 10).

3.23 [20] <§§3.6, 3.7> Scrivere una procedura `trovab` in linguaggio assembler MIPS. La procedura ha in ingresso un solo argomento (nel registro `$a0`), che punta ad una stringa terminata dal carattere Null, e deve trovare la prima occorrenza del carattere `b` all'interno della stringa e restituirne l'indirizzo nel registro `$v0`. Se nella stringa non compaiono delle `b`, allora `trovab` deve restituire un puntatore al carattere Null che termina la stringa. Se ad esempio l'argomento di `trovab` punta alla stringa "`imbibe`", allora il valore di ritorno è il puntatore al terzo carattere della stringa.

3.24 [20] <§§3.6, 3.7> {Es. 3.23} Scrivere una procedura `contab` in linguaggio assembler MIPS. La procedura ha in ingresso un solo argomento (nel registro `$a0`) che punta ad una stringa e restituisce nel registro `$v0` il numero totale dei caratteri `b` che compaiono all'interno della stringa. Nell'implementazione di `contab` deve essere utilizzata la procedura `trovab` dell'Esercizio 3.23.

3.25 [30] <§§3.6, 3.7> Scrivere una procedura `iada` in linguaggio assembler MIPS, che converta un numero intero in una stringa ASCII di cifre decimali. La procedura accetta due argomenti: il primo (nel registro `$a0`) è un numero intero ed il secondo (nel registro `$a1`) è l'indirizzo in cui scrivere la stringa risultato. `iada` deve convertire il primo argomento in una stringa decimale ASCII terminata dal carattere Null e memorizzare tale stringa a partire dalla locazione data. Il valore di ritorno della procedura `iada` (nel registro `$v0`) deve essere il numero di caratteri diversi da Null memorizzati nella destinazione.

In maggior dettaglio

Ricorsione terminale

Alcune procedure ricorsive possono essere implementate in modo iterativo senza utilizzare la ricorsione. L'iterazione può migliorare in modo significativo le prestazioni, rimuovendo l'overhead dovuto alle chiamate a procedura. Si consideri ad esempio una procedura usata per accumulare una somma:

```
int somma (int n, int acc) {
  if (n > 0)
      return somma (n-1, acc + n);
  else
      return acc;
}
```

Si consideri la chiamata a procedura `somma(3, 0)`. Questa provoca le chiamate ricorsive `somma(2, 3)`, `somma(1, 5)` e `somma(0, 6)` ed il risultato 6 è restituito quattro volte. La chiamata ricorsiva della somma è detta *chiamata terminale* (tail call) e questo tipo di ricorsione terminale (tail recursion) può essere implementato in modo molto efficiente (si assuma `$a0 = n` e `$a1 = acc`):

```
somma: beq   $a0, $zero, esci_somma # va a esci_somma se n
                                    # vale 0
       add   $a1, $a1, $a0          # somma n ad acc
       addi  $a0, $a0, -1           # sottrae 1 da n
       j     somma                  # va a somma
esci_somma:
       move  $v0, $a1               # restituisce il valore
                                    # acc
       jr    $ra                    # ritorna al chiamante
```

3.26 [30] <§3.6> Scrivere una procedura MIPS che calcoli l'n-esimo numero di Fibonacci `F(n)` dove

```
F(n) = 0,      se n = 0;
       1,      se n = 1;
       F(n-1) + F(n-2),      altrimenti
```

Basare l'algoritmo sulla seguente procedura (diretta ma terribilmente inefficiente), che genera un processo ricorsivo:

```
int fib (int n) {
  if (n == 0)
      return 0;
  else if (n == 1)
      return 1;
  else
return fib(n-1) + fib(n-2); }
```

3.27 [30] <§3.6> Scrivere un programma analogo a quello dell'Esercizio 3.26, che però si basi sulla procedura riportata nel seguito e che ottimizzi la chiamata terminale in modo da rendere più efficiente l'implementazione:

```
int fib_iter (int a, int b, int cont) {
  if (cont == 0)
      return b;
  else
      return fib_iter (a + b, a, cont - 1); }
```

In questo caso i primi due parametri tengono traccia dei due numeri di Fibonacci precedentemente calcolati. Per calcolare `F(n)` occorre eseguire la chiamata a procedura `fib_iter (1, 0, n)`.

3.28 [20] <§3.6> Stimare la differenza di prestazioni fra la soluzione dell'Esercizio 3.26 e la soluzione dell'Esercizio 3.27.

Il calcolatore a singola istruzione

L'architettura MIPS considerata in questo capitolo ha uno dei set di istruzioni più semplici esistenti; è comunque possibile immaginare set di istruzioni ancora più semplici. Si consideri un elaboratore ipotetico chiamato SIC, da Single Instruction Computer (calcolatore a singola istruzione); come suggerisce il nome, il SIC possiede una sola istruzione, la subtract and branch if negative (esegui una differenza e salta se è minore di zero), abbreviata in `sbn`. L'istruzione `sbn` ha tre operandi, ciascuno dei quali è l'indirizzo di una parola di memoria:

```
sbn a, b, c # Mem[a] = Mem[a] - Mem[b]; se (Mem[a]<0) vai a c
```

L'istruzione sottrae il numero che si trova nella locazione di memoria `b` al numero che si trova in `a` e pone il risultato di nuovo in `a`, sovrascrivendo il valore precedente. Se il risultato della sottrazione è maggiore o uguale a 0, il calcolatore esegue l'istruzione che si trova nella locazione di memoria immediatamente successiva a quella corrente; se invece è minore di 0, l'istruzione seguente è quella memorizzata nella locazione `c`. SIC non ha né registri, né altre istruzioni oltre a `sbn`.

Benché possieda una sola istruzione, SIC può emulare molte delle operazioni di set di istruzioni più complessi, tramite sequenze opportune di istruzioni `sbn`. Ad esempio il programma seguente copia un numero dalla locazione `a` alla locazione `b`:

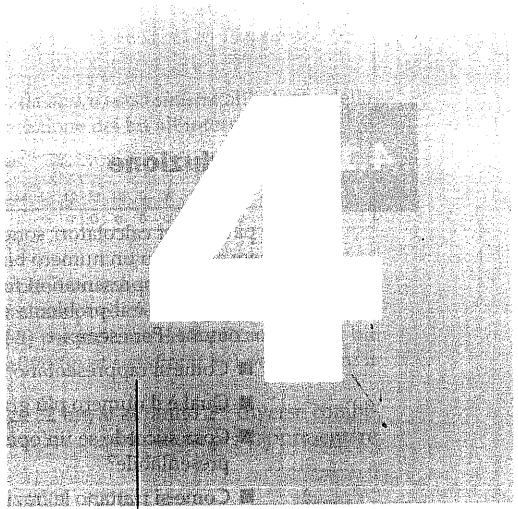
```
inizio:sbn  temp, temp, .+1  # pone temp a zero
       sbn  temp, a, .+1     # pone temp a -a
       sbn  b, b, .+1        # pone b a zero
       sbn  b, temp, .+1     # pone b a -temp, cioè ad a
```

Nel programma sopra riportato, la notazione .+1 indica "l'indirizzo che segue l'indirizzo corrente", così che ciascuna istruzione va a alla linea di codice successiva qualunque sia il risultato della sottrazione. Qui è stato assunto che `temp` sia l'indirizzo di una parola di memoria da usare per memorizzare temporaneamente i risultati.

3.29 [10] <§3.15> Scrivere un programma SIC che sommi `a` e `b`, ponendo il risultato in `a` e lasciando inalterato `b`.

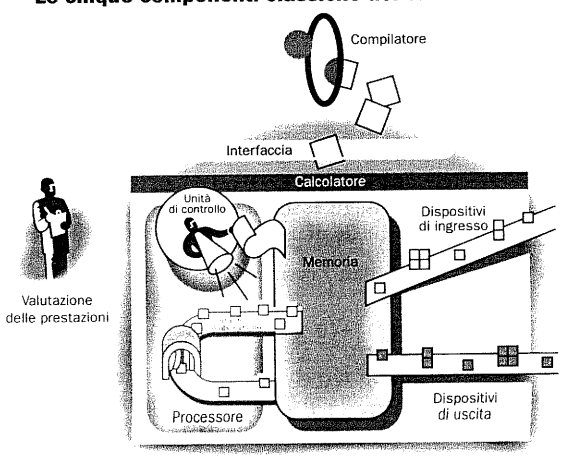
3.30 [20] <§3.15> Scrivere un programma SIC che moltiplichi `a` e `b`, ponendo il risultato in `c`. Si assuma che la locazione di memoria `uno` contenga il numero 1, che `a` e `b` siano maggiori di 0 e che sia possibile modificare `a` o `b`. Un suggerimento: che cosa calcola il seguente programma?

```
c = 0; while (b > 0) {b = b - 1; c = c + a; }
```

L'aritmetica dei calcolatori

Le cinque componenti classiche del calcolatore



La precisione numerica è la vera anima della scienza.

Sir D'arcy Wentworth Thompson
On Growth and Form, 1917

In maggior dettaglio

Il calcolatore a singola istruzione

L'architettura MIPS considerata in questo capitolo ha uno dei set di istruzioni più semplici esistenti; è comunque possibile immaginare set di istruzioni ancora più semplici. Si consideri un elaboratore ipotetico chiamato SIC, da Single Instruction Computer (calcolatore a singola istruzione); come suggerisce il nome, il SIC possiede una sola istruzione, la subtract and branch if negative (esegui una differenza e salta se è minore di zero), abbreviata in `sbn`. L'istruzione `sbn` ha tre operandi, ciascuno dei quali è l'indirizzo di una parola di memoria:

```
sbn a, b, c # Mem[a] = Mem[a] - Mem[b]; se (Mem[a]<0) vai a c
```

L'istruzione sottrae il numero che si trova nella locazione di memoria `b` al numero che si trova in `a` e pone il risultato di nuovo in `a`, sovrascrivendo il valore precedente. Se il risultato della sottrazione è maggiore o uguale a 0, il calcolatore esegue l'istruzione che si trova nella locazione di memoria immediatamente successiva a quella corrente; se invece è minore di 0, l'istruzione seguente è quella memorizzata nella locazione `c`. SIC non ha né registri, né altre istruzioni oltre a `sbn`.

Benché possieda una sola istruzione, SIC può emulare molte delle operazioni di set di istruzioni più complessi, tramite sequenze opportune di istruzioni `sbn`. Ad esempio il programma seguente copia un numero dalla locazione `a` alla locazione `b`:

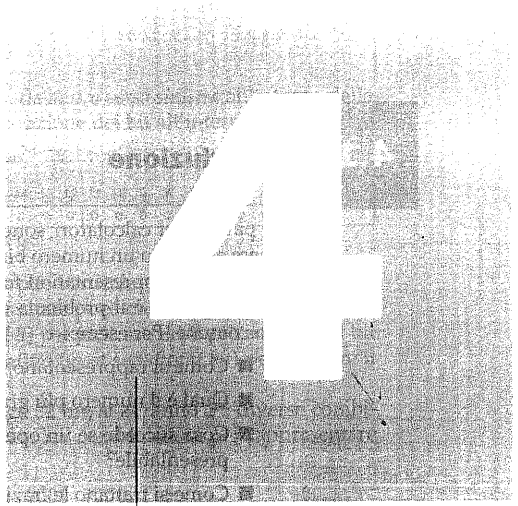
```
inizio:sbn  temp, temp, .+1  # pone temp a zero
       sbn  temp, a, .+1     # pone temp a -a
       sbn  b, b, .+1        # pone b a zero
       sbn  b, temp, .+1     # pone b a -temp, cioè ad a
```

Nel programma sopra riportato, la notazione .+1 indica "l'indirizzo che segue l'indirizzo corrente", così che ciascuna istruzione va a alla linea di codice successiva qualunque sia il risultato della sottrazione. Qui è stato assunto che `temp` sia l'indirizzo di una parola di memoria da usare per memorizzare temporaneamente i risultati.

3.29 [10] <§3.15> Scrivere un programma SIC che sommi `a` e `b`, ponendo il risultato in `a` e lasciando inalterato `b`.

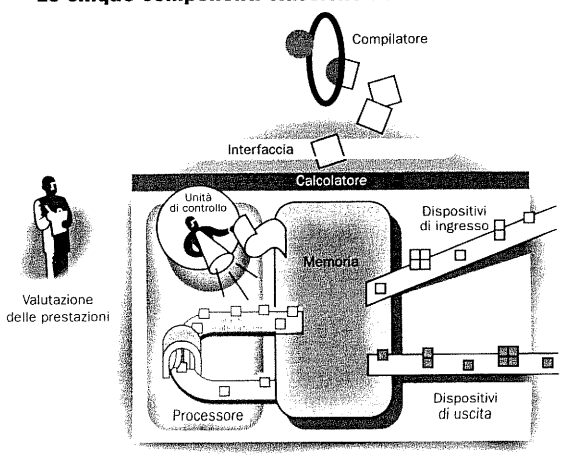
3.30 [20] <§3.15> Scrivere un programma SIC che moltiplichi `a` e `b`, ponendo il risultato in `c`. Si assuma che la locazione di memoria `uno` contenga il numero 1, che `a` e `b` siano maggiori di 0 e che sia possibile modificare `a` o `b`. Un suggerimento: che cosa calcola il seguente programma?

```
c = 0; while (b > 0) {b = b - 1; c = c + a; }
```

L'aritmetica dei calcolatori

Le cinque componenti classiche del calcolatore



La precisione numerica è la vera anima della scienza.

Sir D'arcy Wentworth Thompson
On Growth and Form, 1917

4.1 Introduzione

Le parole dei calcolatori sono composte di bit e pertanto a ciascuna di esse può essere associato un numero binario. I numeri naturali zero, uno, due e così via possono essere rappresentati in forma decimale ed in forma binaria in modo semplice; più complesso è il problema di rappresentare in binario gli altri tipi di numeri di uso comune. Per esempio:

- Come si rappresentano i numeri negativi?
- Qual è il numero più grande rappresentabile in una parola del calcolatore?
- Cosa succede se un'operazione genera un numero più grande di quello rappresentabile?
- Come si trattano le frazioni ed i numeri reali?

Ci si potrebbe anche chiedere qual è la vera storia del famoso baco nel Pentium. E dietro a tutte queste domande vi è un mistero: come può l'hardware eseguire le operazioni di somma, sottrazione, moltiplicazione e divisione sui numeri?

Trattando di rappresentazione dei numeri binari, lo scopo di questo capitolo è svelare il mistero degli algoritmi aritmetici, dell'hardware che implementa questi algoritmi e delle implicazioni che tutto questo ha per il set di istruzioni. I temi trattati potrebbero chiarire anche alcuni dubbi sul funzionamento dei calcolatori. I lettori che già conoscono i numeri binari con segno possono passare al prossimo paragrafo e andare al Paragrafo 4.3.

4.2 Numeri con segno e senza segno

I numeri possono essere rappresentati in una base qualsiasi; gli esseri umani preferiscono la base 10, mentre nel Capitolo 3 si è visto che la base 2 è la più adatta ai calcolatori. Dal momento che si farà frequentemente riferimento sia a numeri decimali sia a numeri binari, per evitare confusioni si aggiungerà ai numeri decimali il pedice *dieci* ed ai numeri binari il pedice *due*.

Qualunque sia la base, il valore della i-esima cifra d di un numero è pari a

$$d \times \text{Base}^i$$

dove i ha un valore che inizia da 0 e viene incrementato da destra verso sinistra. Come conseguenza, un modo immediato per identificare i bit all'interno di una parola è quello di usare la potenza della base relativa a quel bit. Per esempio,

$$1011_{due}$$

rappresenta

$$\begin{aligned}&(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{dieci}\\ &= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{dieci}\\ &= 8 + 0 + 2 + 1_{dieci}\\ &= 11_{dieci}\end{aligned}$$

In conclusione, i bit sono numerati 0, 1, 2, 3... da *destra verso sinistra* all'interno della parola. Il disegno sottostante illustra la numerazione dei bit all'interno di una parola del MIPS e la posizione del numero 1011_{due}:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

ampiezza 32 bit

Dal momento che le parole sono riportate a volte verticalmente e a volte orizzontalmente, i termini destra e sinistra possono risultare poco chiari. Di qui l'uso dei termini bit meno significativo (*least significant bit*) per indicare il bit più a destra (il bit 0 nell'esempio precedente) e bit più significativo (*most significant bit*) per indicare il bit più a sinistra (il bit 31).

Poiché la generica parola del MIPS è lunga 32 bit, si avranno 2^{32} diverse combinazioni di 32 bit ciascuna. È naturale utilizzare tali combinazioni per rappresentare i numeri da 0 a $2^{32}-1$ (4292967295_{dieci}):

$$\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{due} &&= 0_{dieci}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{due} &&= 1_{dieci}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{due} &&= 2_{dieci}\\
&\ldots \qquad \ldots\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{due} &&= 4294967293_{dieci}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{due} &&= 4294967294_{dieci}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{due} &&= 4294967295_{dieci}
\end{aligned}$$

Interfaccia Hardware Software

L'uso della base 2 non è immediato per gli esseri umani; possediamo 10 dita e per questo troviamo più naturale la base 10. Perché i calcolatori non usano la base 10? In realtà, i primi calcolatori commerciali *usavano* l'aritmetica decimale. Il problema consiste nel fatto che i calcolatori utilizzano segnali di tipo on/off, e quindi ciascuna cifra decimale veniva rappresentata tramite una serie di cifre binarie. L'aritmetica decimale si dimostrò così inefficiente che gli elaboratori successivi ritornarono ad essere completamente binari, eseguendo la conversione in base 10 solo per le operazioni (relativamente poco frequenti) di input ed output.

Codice ASCII e numeri binari

Esempio Anziché come interi in complemento a due (si veda la figura 3.15), i numeri possono essere rappresentati come stringhe di cifre ASCII. Di quanto aumenta la necessità di memoria per rappresentare il numero 1 miliardo in ASCII anziché come un intero su 32 bit?

Soluzione Un miliardo corrisponde a 1000000000 e quindi richiede 10 cifre ASCII, ciascuna corrispondente a 8 bit. Dunque l'aumento nella quantità di memoria richiesta è pari a $(10 \times 8)/32$, ossia 2,5. Oltre all'aumento nella quantità di memoria

richiesta, si deve anche considerare la difficoltà nel realizzare l'hardware necessario per sommare, sottrarre, moltiplicare e dividere tali numeri. Tale difficoltà spiega perché coloro che si occupano di calcolatori ritengono che la rappresentazione binaria sia naturale, e che un eventuale elaboratore con aritmetica decimale sia una stranezza.

Si tenga presente che le combinazioni di bit introdotte precedentemente sono semplicemente delle *rappresentazioni* dei numeri. In realtà i numeri hanno un numero infinito di cifre, delle quali quasi tutte assumono il valore 0 tranne alcune nelle posizioni più a destra. Normalmente non si considerano gli 0 iniziali.

Come si vedrà nei paragrafi dal 4.5 al 4.7, è possibile progettare un circuito logico che somma, sottrae, moltiplica e divide queste combinazioni di bit. Se il numero corrispondente al risultato di tali operazioni non può essere rappresentato tramite i bit resi disponibili dall'hardware, si dice che si è verificato un *overflow* (che in inglese significa tracimazione). È compito del sistema operativo e del programma determinare cosa fare quando si verifica un overflow.

I programmi dei calcolatori eseguono calcoli su numeri sia positivi che negativi, ed è quindi necessaria una rappresentazione che permetta di distinguere tra numeri positivi e numeri negativi. La soluzione più ovvia consiste nell'aggiungere un'informazione di segno, che può essere convenientemente rappresentata da un singolo bit; il nome di questa rappresentazione è *modulo e segno*.

Purtroppo la rappresentazione in modulo e segno ha diversi limiti. Innanzitutto, non è chiaro dove mettere il bit di segno. A destra? A sinistra? I primi elaboratori sperimentarono ambedue le soluzioni. In secondo luogo i sommatori che sfruttano questa rappresentazione richiedono un passo aggiuntivo per determinare il segno, dal momento che questo non può essere calcolato a priori. Infine un bit di segno separato dagli altri significa che la rappresentazione in modulo e segno possiede sia uno zero positivo che uno zero negativo, cosa che può creare dei problemi ai programmatori. La conseguenza di questi limiti è che la rappresentazione in modulo e segno fu abbandonata molto presto.

Mentre si cercavano alternative più convenienti, si considerò il problema di come potesse essere rappresentato il risultato di un'operazione di sottrazione tra numeri senza segno quando si sottrae un numero grande da uno piccolo. Applicando le normali regole della sottrazione si ottiene un risultato con una stringa di 1 iniziali, determinati da riporti tra i due operandi.

Dal momento che tra le soluzioni a portata di mano non ve ne era una migliore, la soluzione fu quella di adottare la rappresentazione che rende più semplice l'hardware, ossia prevedere che una sequenza di 0 iniziali indicasse un valore positivo, ed una sequenza di 1 un valore negativo. Questa convenzione per la rappresentazione dei numeri binari prende il nome di rappresentazione *in complemento a due*:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{due} = 0_{dieci}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{due} = 1_{dieci}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{due} = 2_{dieci}$
...
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{due} = 2147483645_{dieci}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{due} = 2147483646_{dieci}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{due} = 2147483647_{dieci}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{due} = -2147483648_{dieci}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{due} = -2147483647_{dieci}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{due} = -2147483646_{dieci}$
...
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{due} = -3_{dieci}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{due} = -2_{dieci}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{due} = -1_{dieci}$

La metà positiva dei numeri, da 0 a 2147483647_{dieci} ($2^{31}-1$), usa la stessa rappresentazione di prima. La combinazione successiva ($1000...0000_{due}$) rappresenta il numero negativo di valore assoluto maggiore, ossia -2147483648_{dieci} (-2^{31}). Subito dopo vi è una sequenza decrescente di numeri negativi, da -2147483647_{dieci} ($1000...0001_{due}$) fino a -1_{dieci} ($1111...1111_{due}$).

La rappresentazione in complemento a due comprende un numero negativo, -2147483648_{dieci}, che non ha un corrispondente numero positivo. Questo sbilanciamento richiede una certa attenzione ai programmatori, ma d'altronde la rappresentazione in modulo e segno dava problemi sia ai programmatori che ai progettisti dell'hardware. Pertanto tutti i calcolatori di oggi usano la rappresentazione binaria in complemento a due per i numeri con segno.

La rappresentazione in complemento a due ha il vantaggio che tutti i numeri negativi hanno il bit più significativo uguale a 1. Perciò l'hardware deve controllare soltanto questo bit per verificare se il numero è positivo o negativo (lo zero viene considerato positivo). Questo bit particolare è a volte chiamato *bit di segno*. Considerato il ruolo particolare assunto da questo bit, si possono rappresentare numeri positivi e negativi associando a ciascun bit un valore funzione della corrispondente potenza di 2 (qui xi indica l'i-esimo bit di x):

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + ... + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

Il bit di segno è moltiplicato per -2^{31}, mentre gli altri bit sono moltiplicati per i rispettivi valori positivi della base.

Conversione da binario a decimale

Esempio Qual è il valore decimale del seguente numero su 32 bit in complemento a due?

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{due}$

Soluzione Sostituendo i valori dei bit nella formula precedentemente introdotta si ha:

$$(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + ... + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0})$$
$$= -2^{31} + 2^{30} + 2^{29} + ... + 2^{2} + 0 + 0$$
$$= -2147483648_{dieci} + 2147483644_{dieci}$$
$$= -4_{dieci}$$

Verrà introdotta tra breve una scorciatoia per rendere più semplice la conversione.

Interfaccia Hardware Software

La differenza tra numeri con segno e senza segno influenza le operazioni di caricamento oltre a quelle aritmetiche. Nella pratica, quando si carica un numero con segno in un registro si copia ripetutamente il bit di segno per riempire il resto del registro (operazione chiamata *estensione del segno*). Così facendo si realizza lo *scopo* di scrivere la rappresentazione corretta del numero in tale registro. Le operazioni di caricamento di numeri senza segno si limitano a riempire di zeri la parte a sinistra del dato, dal momento che il numero rappresentato dalla combinazione di bit è senza segno.

Quando si carica una parola di 32 bit in un registro di 32 bit, la questione è puramente accademica: le operazioni di caricamento di numeri con segno e senza segno sono identiche. Il MIPS offre due tipi di operazioni di caricamento di byte: *load byte* (`lb`) considera il byte come un numero con segno e di conseguenza estende il segno in maniera da riempire i 24 bit più significativi del registro, mentre *load byte unsigned* (`lbu`) lavora con interi senza segno. I programmi usano i byte quasi esclusivamente per rappresentare i caratteri e non per contenere interi con segno di piccolo valore. Pertanto, l'istruzione `lbu` è impiegata soltanto per il caricamento di byte.

Così come un'operazione su numeri senza segno può eccedere le capacità di rappresentazione del risultato da parte dell'hardware, anche un'operazione su numeri in complemento a due può portare allo stesso risultato. La condizione di overflow si verifica quando il bit più a sinistra tra quelli della combinazione di bit che rappresenta il numero non è uguale al valore assunto dall'infinito numero di cifre che vi sono a sinistra (ossia il bit di segno non è corretto): uno 0 a sinistra nella combinazione di bit che rappresentano un numero negativo oppure un 1 in quella che rappresenta un numero positivo.

Interfaccia Hardware Software

Diversamente dai numeri di cui si è parlato prima, gli indirizzi di memoria iniziano dal valore 0 e continuano verso i valori più grandi; in altre parole gli indirizzi negativi non hanno senso. Perciò i programmi lavorano a volte su numeri che possono essere positivi o negativi, a volte su numeri che possono essere solo positivi. Questa distinzione si riflette sui linguaggi di programmazione: il C per esempio chiama i primi *interi* (dichiarati come `int` nel programma) ed i secondi *interi senza segno* (`unsigned int`).

Nelle istruzioni di confronto si deve tenere conto di questa dicotomia. A volte una combinazione di bit con un 1 nel bit più significativo rappresenta un numero negativo e, ovviamente, è minore di qualunque numero positivo, che deve avere uno 0 nel bit più significativo. Con gli interi senza segno, d'altro canto, un 1 nel bit più significativo rappresenta un numero che è *più grande* di qualunque altro che inizi con uno 0.

Il MIPS prevede due versioni delle operazioni di "set on less than" per gestire questa alternativa. *Set on less than* (`slt`) e *set on less than immediate* (`slti`) lavorano su interi con segno. Gli interi senza segno sono confrontati utilizzando *set on less than unsigned* (`sltu`) e *set on less than immediate unsigned* (`sltiu`).

Esempio

Confronti con segno e senza segno

Si supponga che il registro `$s0` contenga il numero binario

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{due}$

e che il registro $s1 contenga il numero binario

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{due}$

Quali sono i valori dei registri $t0 e $t1 dopo le seguenti due istruzioni?

```
slt     $t0, $s0, $s1 # confronto con segno
sltu    $t1, $s0, $s1 # confronto senza segno
```

Soluzione

Il valore nel registro `$s0` rappresenta il numero –1 se è un intero ed il numero 4294967295_{dieci} se è un intero senza segno. Il valore nel registro `$s1` rappresenta il valore 1 in entrambi i casi. Quindi il registro `$t0` contiene il valore 1, dal momento che $-1_{dieci} < 1_{dieci}$, ed il registro `$t1` contiene il valore 0, dal momento che $4294967295_{dieci} > 1_{dieci}$.

Prima di procedere a trattare le operazioni di somma e sottrazione, verranno esaminate alcune utili scorciatoie per lavorare con i numeri in complemento a due.

La prima scorciatoia è un modo veloce per cambiare il segno ad un numero binario in complemento a due. Basta invertire ogni 0 in 1 ed ogni 1 in 0 e poi sommare uno al risultato. Questa scorciatoia si basa sull'osservazione che la somma di un numero e della sua rappresentazione invertita deve dare sempre $111...111_{due}$, che rappresenta –1. Poiché $x + \bar{x} \equiv -1$, allora $x + \bar{x} + 1 = 0$, ossia $\bar{x} + 1 = -x$.

Esempio

Scorciatoia per cambiare il segno di un numero

Si cambi il segno del numero 2_{dieci} e si verifichi poi il risultato facendo lo stesso con il numero -2_{dieci}.

Soluzione

$2_{dieci} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{due}$

Negando questo numero tramite l'inversione dei bit e la somma di un 1 si ottiene

$$\begin{array}{ll} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{due} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{due} \\ \hline = & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{due} \\ = & -2_{dieci} \end{array}$$

Procedendo nell'altra direzione si parte da:

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{due}$

Si inverte e poi si incrementa:

$$
\begin{array}{rl}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{due} \\
+ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{due} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{due} \\
= & 2_{dieci}
\end{array}
$$

La seconda scorciatoia serve per convertire un numero binario rappresentato su n bit in un numero rappresentato su più di n bit. Si consideri per esempio il campo immediato delle istruzioni di load, store, branch, add e set on less than, che contiene un numero su 16 bit in complemento a due, e che può quindi rappresentare i valori da -32768_{dieci} (-2^{15}) a 32767_{dieci} ($2^{15}-1$). Per sommare il campo immediato ad un registro di 32 bit, l'elaboratore deve convertire il numero su 16 bit nel suo equivalente su 32 bit. La scorciatoia consiste nel prendere il bit più significativo dell'entità minore (ossia il bit di segno) e nel replicarlo, in maniera da riempire i nuovi bit presenti nell'entità maggiore. I vecchi bit sono semplicemente copiati nella parte destra della nuova parola. Questa scorciatoia prende comunemente il nome di *estensione del segno*.

Scorciatoia per l'estensione del segno

Esempio Si converta la versione binaria su 16 bit di 2_{dieci} e di -2_{dieci} nei corrispondenti numeri binari su 32 bit.

Soluzione La versione binaria su 16 bit del numero 2 è

$$0000\ 0000\ 0000\ 0010_{due} = 2_{dieci}$$

La conversione in numero su 32 bit va fatta eseguendo 16 copie del valore del bit più significativo (0) e mettendole nella metà sinistra della parola.
La metà destra riceve invece il vecchio valore:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{due} = 2_{dieci}$$

Si esegue ora la negazione della versione su 16 bit del numero 2 utilizzando la precedente scorciatoia. Quindi

$$0000\ 0000\ 0000\ 0010_{due}$$

diventa

$$
\begin{array}{rl}
 & 1111\ 1111\ 1111\ 1101_{due} \\
+ & \qquad\qquad\qquad\qquad\qquad 1_{due} \\
\hline
= & 1111\ 1111\ 1111\ 1110_{due}
\end{array}
$$

Per creare la versione su 32 bit del numero negativo si deve copiare il bit di segno per 16 volte e metterlo nella parte sinistra:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{due} = -2_{dieci}$$

Esadecimale	Binario	Esadecimale	Binario	Esadecimale	Binario	Esadecimale	Binario
0_{esa}	0000_{due}	4_{esa}	0100_{due}	8_{esa}	1000_{due}	c_{esa}	1100_{due}
1_{esa}	0001_{due}	5_{esa}	0101_{due}	9_{esa}	1001_{due}	d_{esa}	1101_{due}
2_{esa}	0010_{due}	6_{esa}	0110_{due}	a_{esa}	1010_{due}	e_{esa}	1110_{due}
3_{esa}	0011_{due}	7_{esa}	0111_{due}	b_{esa}	1011_{due}	f_{esa}	1111_{due}

Figura 4.1 Tabella di conversione esadecimale-binario. Si rimpiazzi ciascuna cifra esadecimale con le corrispondenti quattro cifre binarie e viceversa. Se la lunghezza del numero binario non è un multiplo di quattro, si deve procedere da destra verso sinistra.

Il trucco funziona perché i numeri positivi in complemento a due possiedono effettivamente un numero infinito di 0 a sinistra, e quelli negativi un numero infinito di 1. La combinazione di bit che rappresenta il numero nasconde i bit iniziali per poter rientrare nell'ampiezza della parola resa disponibile dall'hardware; l'estensione del segno semplicemente ripristina alcuni di questi bit.

Infine esiste una scorciatoia, di cui si era già accennato nel Capitolo 3, che permette di risparmiare tempo nella lettura e scrittura di numeri binari particolarmente lunghi grazie all'uso di una base più grande di quella binaria, ma facilmente convertibile in quest'ultima. Dal momento che pressoché tutte le grandezze utilizzate dai calcolatori sono multiple di 4, sono molto diffusi i numeri *esadecimali* (ossia in base 16). Poiché 16 è una potenza di 2, si può banalmente eseguire la conversione sostituendo ad ogni gruppo di quattro cifre binarie una singola cifra esadecimale e viceversa. La figura 4.1 riporta la stele di Rosetta dei numeri esadecimali. Per denotare i numeri esadecimali si userà o il pedice *esa*, o la notazione C, cioè 0x*nnnnn*.

Scorciatoia per la conversione da binario ad esadecimale

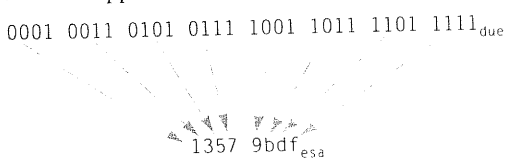
Esempio Si convertano i seguenti numeri esadecimali e binari nell'altra base:

$$eca8\ 6420_{esa}$$

$$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_{due}$$

Soluzione È sufficiente una tabella di conversione in una direzione:

ed una nella direzione opposta:



Riepilogo

Il punto principale trattato in questo paragrafo è la necessità di rappresentare nella parola di un calcolatore sia numeri interi positivi che negativi; benché in ogni decisione vi siano sempre dei pro e dei contro, la scelta dominante a partire dal 1965 è stata quella della rappresentazione in complemento a due. La figura 4.2 riporta le aggiunte al linguaggio assembler del MIPS introdotte in questo paragrafo. Il linguaggio dell'elaboratore MIPS è illustrato anche nelle ultime pagine di questo libro.

Operandi MIPS

Nome	Esempio	Commenti
32 registri	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. Il registro `$zero` ha sempre valore 0. Il registro `$at` è riservato all'assembler per la gestione di costanti di grandi dimensioni.
2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte e quindi le parole successive differiscono di 4 unità. La memoria mantiene le strutture dati, come i vettori, i registri riversati (spilled) e i registri salvati durante le chiamate a procedura.

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Tre operandi; dati nei registri
	sottrazione	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Tre operandi; dati nei registri
	somma immediata	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Usata per sommare costanti
Trasferimento dati	load word	`lw $s1, 100($s2)`	`$s1` = Memoria[`$s2` + 100]	Word dalla memoria al registro
	store word	`sw $s1, 100($s2)`	Memoria[`$s2` + 100] = `$s1`	Word dal registro alla memoria
	load byte unsigned	`lbu $s1, 100($s2)`	`$s1` = Memoria[`$s2` + 100]	Byte dalla memoria al registro
	store byte	`sb $s1, 100($s2)`	Memoria[`$s2` + 100] = `$s1`	Byte dal registro alla memoria
	load upper immediate	`lui $s1, 100`	`$s1` = 100 * 2^{16}	Carica una costante nei 16 bit più significativi
Salto condizionato	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Controllo dell'uguaglianza ed eventuale salto; salto PC relative
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Controllo della disuguaglianza ed eventuale salto; salto PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0;	Confronto del tipo "minore di"; usato con beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0;	Confronto con una costante
	set less than unsigned	`sltu $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0;	Confronto del tipo "minore di"; numeri senza segno
	set on less than immediate unsigned	`sltiu $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0;	Confronto con una costante; numeri senza segno
Salto incondizionato	jump	`j 2500`	go to 10000	Salto ad un indirizzo specificato
	jump register	`jr $ra`	go to `$ra`	Usato nei costrutti switch e nell'uscita dalle procedure
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	Per le chiamate a procedura

Figura 4.2 L'architettura del MIPS introdotta sinora. Sono evidenziate le parti di questa tabella aggiunte rispetto all'architettura del MIPS introdotta nel corso del Capitolo 3 (figura 3.20). Il linguaggio dell'elaboratore MIPS è riassunto nelle ultime pagine del libro.

Approfondimento: La rappresentazione in complemento a due deve il suo nome alla regola secondo la quale la somma senza segno di un numero di n bit e del suo negato è pari a 2^n, e quindi il complemento (o negazione) di un numero x in complemento a due è pari a 2^n-x.

Una terza alternativa in termini di rappresentazione è denominata *complemento a uno*. La negazione di un numero in complemento a uno si trova invertendo ciascun bit, da 0 a 1 e da 1 a 0, e questo spiega il suo nome, dal momento che il complemento di x è 2^n-x-1. Questa rappresentazione costituisce un ulteriore tentativo di determinare una soluzione migliore della rappresentazione in modulo e segno, e diversi calcolatori di tipo scientifico utilizzavano tale notazione. Essa è simile al complemento a due, se si esclude il fatto che possiede anch'essa due 0: lo 0 positivo $00...00_{due}$ e quello negativo $11...11_{due}$. Il numero negativo di valore assoluto maggiore $10...000_{due}$ rappresenta il valore -2147483647_{dieci}, ed in tal modo gli insiemi dei numeri positivi e quello dei numeri negativi sono bilanciati. I sommatori in complemento a uno richiedono un passo aggiuntivo per eseguire la sottrazione di un numero, e per questo il complemento a due domina la situazione attuale.

Come nota conclusiva, di cui si terrà conto quando si discuterà della virgola mobile, si ricordi la possibilità di rappresentare il valore più negativo con $00...00_{due}$ ed il valore più positivo con $11...11_{due}$, con lo 0 che assume il valore $10...000_{due}$. Questa notazione si chiama *polarizzata*, in quanto polarizza il numero in maniera tale che il numero stesso più il valore della polarizzazione corrisponda ad una rappresentazione non negativa.

4.3 Somma e sottrazione

La sottrazione: l'amica complicata della somma

No. 10, Top Ten Courses for Athletes at a Football Factory,
David Letterman et al., *Book of Top Ten Lists*, 1990

La somma in un calcolatore non è nulla di diverso da ciò che ci si aspetterebbe. Le cifre sono sommate bit a bit da destra verso sinistra, con il riporto passato verso la cifra a sinistra, esattamente come si farebbe a mano. La sottrazione è basata sulla somma: l'operando opportuno (*sottraendo*) è semplicemente negato prima di essere sommato.

Somma e sottrazione binarie

Esempio Si provi a sommare in binario 6_{dieci} e 7_{dieci} e poi a sottrarre 6_{dieci} da 7_{dieci}.

Soluzione

```
    0000 0000 0000 0000 0000 0000 0000 0111due = 7dieci
+   0000 0000 0000 0000 0000 0000 0000 0110due = 6dieci
    ----------------------------------------------------
=   0000 0000 0000 0000 0000 0000 0000 1101due =      13dieci
```

Tutto si svolge sui 4 bit a destra; la figura 4.3 mostra le operazioni di somma e di riporto. I riporti sono scritti in parentesi e le frecce mostrano la loro propagazione.

	(0)	(0)	(1)	(1)	(0)	(Riporti)
...	0	0	0	1	1	1
...	0	0	0	1	1	0
...	(0) 0	(0) 0	(0) 1	(1) 1	(1) 0	(0) 1

Figura 4.3 Somma binaria: sono visibili i riporti da destra verso sinistra. Nel bit più a destra si somma 1 a 0, ed il risultato della somma di questo bit è 1 mentre il riporto generato vale 0. Quindi l'operazione da farsi sul secondo bit a destra è 0 + 1 + 1. Questo genera uno 0 come somma ed un riporto pari a 1. Sul terzo bit si esegue la somma 1 + 1 + 1, producendo una somma pari a 1 ed un riporto pari a 1. Sul quarto bit si esegue 1 + 0 + 0, generando 1 come somma e nessun riporto.

La sottrazione di 6_{dieci} da 7_{dieci} può essere eseguita direttamente:

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{due} & = 7_{dieci} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{due} & = 6_{dieci} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{due} & = 1_{dieci} \end{array}$$

oppure attraverso la somma, usando la rappresentazione in complemento a due di –6:

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{due} & = 7_{dieci} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{due} & = -6_{dieci} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{due} & = 1_{dieci} \end{array}$$

Come già detto in precedenza, l'overflow si verifica quando il risultato di un'operazione non può essere rappresentato con l'hardware disponibile, in questo caso una parola di 32 bit. Quando si può verificare un overflow durante una somma? Quando si sommano operandi con segno opposto non può verificarsi overflow: la ragione è che in questo caso la somma non può essere più grande, in valore assoluto, di uno degli operandi; per esempio, –10+4 = –6. Poiché gli operandi sono rappresentabili in 32 bit e la somma non può essere più grande di un operando, anche la somma deve essere contenuta in 32 bit. Quindi non può verificarsi overflow quando si sommano un operando positivo ed uno negativo.

Vi sono delle regole analoghe per il verificarsi di un overflow nel corso di una sottrazione, ma il principio è semplicemente opposto: quando il segno degli operandi è *lo stesso*, allora l'overflow non può verificarsi. Per rendersene conto, si ricordi che $x-y = x+(-y)$, poiché si sottrae eseguendo la negazione del secondo operando e poi sommando. Quindi quando si sottraggono operandi con lo stesso segno si finisce per *sommare* operandi con segno *opposto*. Sulla base di quanto detto nel capoverso precedente, è chiaro che in questo caso l'overflow non può verificarsi.

Pur avendo esaminato quando l'overflow non può verificarsi nel corso della somma e della sottrazione, non è ancora stata data una risposta su come rilevarlo nel caso si verifichi. L'overflow si verifica quando si sommano due numeri positivi e la somma è negativa, o viceversa. Chiaramente, sommando o sottraendo due numeri su 32 bit si può ottenere un risultato che richiede 33 bit per essere completamente rappresentato; la mancanza del 33-esimo bit implica che quando si verifica l'overflow il bit di segno assume il *valore* del risultato anziché il segno del risultato. Dal

Operazione	Operando A	Operando B	Risultato
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A–B	≥ 0	< 0	< 0
A–B	< 0	≥ 0	≥ 0

Figura 4.4 Condizioni di overflow per somma e sottrazione.

momento che sarebbe necessario soltanto un bit in più, soltanto il bit di segno può essere sbagliato; questo significa che è stato generato un riporto sul bit di segno.

Nelle operazioni di sottrazione l'overflow si verifica quando si sottrae un numero negativo da un numero positivo e si ottiene un risultato negativo, o quando si sottrae un numero positivo da un numero negativo e si ottiene un numero positivo: questo significa che si è verificato un riporto a partire dal bit di segno. La figura 4.4 mostra le combinazioni di operazioni, operandi e risultati che indicano il verificarsi di un overflow. L'Esercizio 4.42 fornisce una scorciatoia per permettere un più semplice rilevamento dell'overflow via hardware.

Si è appena visto come rilevare l'overflow in un elaboratore quando si lavora con numeri in complemento a due. Cosa succede con i numeri senza segno? Gli interi senza segno sono normalmente usati per rappresentare gli indirizzi di memoria, per i quali l'overflow viene ignorato.

Il progettista di elaboratori deve quindi trovare un modo per ignorare l'overflow in alcuni casi e per riconoscerlo in altri. La soluzione adottata per il MIPS consiste nel disporre di due tipi di istruzioni aritmetiche, in grado quindi di riconoscere le due possibili situazioni:

- La somma (`add`), la somma con immediato (`addi`) e la sottrazione (`sub`) causano un'eccezione in caso di overflow.
- La somma senza segno (`addu`), la somma con immediato senza segno (`addiu`) e la sottrazione senza segno (`subu`) non causano eccezione in caso di overflow.

Dal momento che il C ignora gli overflow, i compilatori C del MIPS genereranno sempre le versioni senza segno delle istruzioni aritmetiche `addu`, `addiu` e `subu`, indipendentemente dal tipo delle variabili. I compilatori Fortran del MIPS, invece, scelgono l'istruzione aritmetica appropriata a seconda del tipo degli operandi.

Il progettista di un elaboratore deve decidere come gestire l'overflow aritmetico. Anche se alcuni linguaggi come il C lasciano la decisione al progettista dell'elaboratore, linguaggi come l'Ada ed il Fortran richiedono che il programma riceva una segnalazione. Saranno il programmatore o l'ambiente di programmazione a decidere poi che cosa fare in caso di overflow.

Il MIPS rileva l'overflow tramite una *eccezione*, anche chiamata *interrupt* in molti calcolatori. Un'eccezione o interrupt è essenzialmente una chiamata a procedura non preventivata. L'indirizzo dell'istruzione che ha causato l'overflow viene salvato in un registro ed il calcolatore salta ad un indirizzo predefinito per invocare la procedura appropriata per quella eccezione. L'indirizzo dell'i-

struzione interrotta viene salvato, in maniera tale che in alcune situazioni il programma possa continuare dopo che è stato eseguito del codice per la correzione. Il Paragrafo 5.6 tratta in maggior dettaglio le eccezioni; i Capitoli 7 e 8 descrivono altre situazioni in cui possono verificarsi eccezioni ed interrupt.

Il MIPS contiene un registro chiamato *exception program counter (EPC)*, che contiene l'indirizzo dell'istruzione che ha causato l'eccezione. L'istruzione *move from system control* (`mfc0`) può venire usata per copiare il registro EPC in un registro general-purpose, in maniera tale che il software del MIPS abbia la possibilità di ritornare all'istruzione che ha causato l'eccezione attraverso un'istruzione di salto tramite registro.

Riepilogo

Il punto principale di questo paragrafo è il fatto che, indipendentemente dalla rappresentazione, la dimensione finita delle parole dei calcolatori implica che le operazioni aritmetiche possano generare risultati troppo grandi per poter stare nella dimensione fissa delle parole. È facile rilevare l'overflow quando si lavora con i numeri senza segno, anche se in questo caso l'overflow viene quasi sempre ignorato poiché i programmi non richiedono di rilevare l'overflow nell'ambito dell'aritmetica degli indirizzi, che rappresenta l'uso più comune dei numeri naturali. I numeri in complemento a due rappresentano un problema maggiore, ed alcuni sistemi software richiedono la possibilità di rilevare l'overflow, così che oggi tutti gli elaboratori possiedono un meccanismo per rilevarlo. La figura 4.5 riporta le aggiunte fatte all'architettura MIPS durante questo paragrafo.

Approfondimento: Il MIPS può scatenare un'eccezione in caso di overflow, ma a differenza di molti altri elaboratori non possiede un'istruzione di salto condizionato per verificarne l'occorrenza. Una sequenza di istruzioni MIPS può scoprire un overflow. Per la somma tra numeri con segno, la sequenza è la seguente (si veda il dettaglio annesso agli Esercizi 4.40 e 4.41 per la definizione delle istruzioni `xor` e `nor`):

```
addu $t0, $t1,   $t2          # $t0 = somma, ma non attiva
                              # l'eccezione
xor  $t3, $t1,   $t2          # verifica se i segni sono diversi
slt  $t3, $t3,   $zero        # $t3 = 1 se i segni sono diversi
bne  $t3, $zero, No_overflow  # segni di $t1 e $t2
                              # diversi, no overflow
xor  $t3, $t0,   $t1          # segni uguali; anche segno somma
                              # uguale?
                              # $t3 negativo se segno somma diverso
slt  $t3, $t3,   $zero        # $t3 = 1 se segno somma diverso
bne  $t3, $zero, Overflow     # tre segni diversi; vai a overflow
```

Per la somma senza segno ($t0= $t1 + $t2), il test è

```
addu $t0, $t1,   $t2          # $t0 = somma
nor  $t3, $t1,   $zero        # $t3 = NOT $t1
                              # (compl. a 2 - 1: 2^32 - $t1 - 1)
```


Operandi MIPS

Nome	Esempio	Commenti
32 registri	`$s0-$s7, $t0-$t9, $gp, $fp, $zero, $sp, $ra, $at`	Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. Il registro `$zero` ha sempre valore 0. Il registro `$at` è riservato all'assembler per la gestione di costanti di grandi dimensioni.
2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte e quindi le parole successive differiscono di 4 unità. La memoria mantiene le strutture dati, come i vettori, i registri riversati (spilled) e i registri salvati durante le chiamate a procedura.

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Tre operandi; dati nei registri
	sottrazione	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Tre operandi; dati nei registri
	somma immediata	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Usata per sommare costanti
	somma senza segno	`addu $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Tre operandi; nessun test sull'overflow
	sottrazione senza segno	`subu $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Tre operandi; nessun test sull'overflow
	somma immediata senza segno	`addiu $s1, $s2, 100`	`$s1 = $s2 + 100`	Somma una costante; nessun test sull'overflow
	sposta dal registro del coprocessore	`mfc0 $s1, $epc`	`$s1 = $epc`	Copia l'Exception PC ed altri registri speciali
Trasferimento dati	load word	`lw $s1, 100($s2)`	`$s1` = Memoria[`$s2` + 100]	Word dalla memoria al registro
	store word	`sw $s1, 100($s2)`	Memoria[`$s2` + 100] = `$s1`	Word dal registro alla memoria
	load byte unsigned	`lbu $s1, 100($s2)`	`$s1` = Memoria[`$s2` + 100]	Byte dalla memoria al registro
	store byte	`sb $s1, 100($s2)`	Memoria[`$s2` + 100] = `$s1`	Byte dal registro alla memoria
	load upper immediate	`lui $s1, 100`	`$s1` = 100 * 216	Carica una costante nei 16 bit più significativi
Salto condizionato	branch on equal	`beq $s1, $s2, 25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Controllo dell'uguaglianza ed eventuale salto; salto PC-relative
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1` != `$s2`) go to PC + 4 + 100	Controllo della disuguaglianza ed eventuale salto; salto PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0;	Confronto del tipo "minore di"; usato con beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2` < 100) `$s1` = 1; else `$s1` = 0;	Confronto con una costante
	set less than unsigned	`sltu $s1, $s2, $s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0;	Confronto del tipo "minore di"; numeri senza segno
	set on less than immediate unsigned	`sltiu $s1, $s2, 100`	if (`$s2` < 100) `$s1` = 1; else `$s1` = 0;	Confronto con una costante; numeri senza segno
Salto incondizionato	jump	`j 2500`	go to 10000	Salto ad un indirizzo specificato
	jump register	`jr $ra`	go to `$ra`	Usato nei costrutti switch e nell'uscita dalle procedure
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	Per le chiamate a procedura

Figura 4.5 L'architettura del MIPS introdotta sinora. Il colore indica le parti aggiunte rispetto alla figura 4.2. Il linguaggio dell'elaboratore MIPS è riassunto nelle ultime pagine del libro.

```
sltu $t3, $t3,   $t2          # (2^32 - $t1 - 1) < $t2
                              # ⇒ 2^32 - 1 < $t1 + $t2
bne  $t3, $zero, Overflow     # se (2^32 - 1 < $t1 + $t2) vai a
                              # overflow
```

Approfondimento: In precedenza si è detto che si copiava EPC in un registro tramite `mfc0` e poi si ritornava al codice interrotto con un salto tramite registro. Questo conduce ad un'interessante problema: dal momento che si deve prima trasferire EPC in un registro per usarlo come registro di salto, come può il salto tramite registro ritornare al codice interrotto e ripristinare i valori originari di *tutti* i registri? O si ripristinano prima i vecchi registri, distruggendo così l'indirizzo di ritorno contenuto in EPC e che era stato copiato in un registro per essere usato in un salto tramite registro, o si ripristinano tutti i registri tranne quello con l'indirizzo di ritorno in maniera tale da poter saltare, con la conseguenza che un'eccezione comporta il cambiamento di quel registro in un qualunque istante di esecuzione di un programma! Nessuna delle due opzioni è accettabile.

Per risolvere questo problema, i programmatori del MIPS hanno scelto di riservare i registri `$k0` e `$k1` per il sistema operativo; questi registri *non* sono ripristinati durante le eccezioni. Esattamente come i compilatori MIPS evitano l'uso del registro `$at`, così che l'assemblatore possa utilizzarlo come registro temporaneo (si veda Interfaccia Hardware Software nel Paragrafo 3.8), allo stesso modo i compilatori si astengono dall'usare i registri `$k0` e `$k1` per lasciarli disponibili per il sistema operativo. Le procedure di eccezione scrivono l'indirizzo di ritorno in uno di questi registri e poi usano un salto tramite registro per ripristinare l'indirizzo dell'istruzione.

4.4 Operazioni logiche

"Al contrario," continuò Tweedledee, "se sarà così, bene; se fosse stato così, sarebbe stato bene; ma poiché non è così, non lo è. Questa è logica."

Lewis Carroll, *Alice's Adventures in Wonderland*, 1865

Benché i primi calcolatori trattassero principalmente parole intere, divenne ben presto chiaro che era utile lavorare anche su campi di bit all'interno di una parola, o addirittura su singoli bit. L'analisi di singoli caratteri all'interno di una parola, ciascuno memorizzato su 8 bit, costituisce un esempio di tali operazioni. Di conseguenza vennero aggiunte delle istruzioni per contribuire, tra le altre cose, a compattare e a scompattare i bit all'interno delle parole.

Una classe di tali operazioni è denominata *scalamento* (*shift*). Le operazioni di scalamento spostano tutti i bit di una parola verso sinistra o verso destra, riempiendo con degli 0 i bit rimasti vuoti. Per esempio, se il registro `$s0` conteneva

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{due}$$

ed è stata eseguita l'operazione di scalamento a sinistra di otto posizioni, il nuovo valore sarà

$$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{due}$$

Il duale dello scalamento a sinistra è lo scalamento a destra. Il nome effettivo delle due istruzioni di scalmento del MIPS è *shift left logical* (`sll`) e *shift right logical* (`srl`). La seguente istruzione esegue l'operazione descritta sopra, assumendo che il risultato venga memorizzato nel registro `$t2`:

```
sll  $t2,  $s0,  8      # reg $t2 = reg $s0 << 8 bit
```


Nel Capitolo 3 si è rinviata la spiegazione del campo *shamt* nel formato-R. Il nome sta per *shift amount* (dimensione dello scalamento) ed è usato nelle istruzioni di scalamento. Quindi la versione in linguaggio macchina dell'istruzione precedente è:

op	rs	rt	rd	shamt	funct
0	0	16	10	8	0

La codifica di `sll` prevede uno 0 sia nel campo op che nel campo funct, mentre rd specifica `$t2`, rt specifica `$s0`, e shamt contiene 8. Il campo rs non è utilizzato ed è quindi messo a 0.

Un'altra operazione utile per isolare i campi è quella di AND. AND è un'operazione bit a bit che genera un 1 nel risultato solo se ambedue i bit degli operandi valgono 1. Per esempio, se il registro `$t2` contiene ancora

$$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{due}$$

ed il registro `$t1` contiene

$$0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{due}$$

allora dopo l'esecuzione della seguente istruzione MIPS

```
and  $t0,  $t1,  $t2   # reg $t0 = reg $t1 & reg $t2
```

il valore del registro `$t0` sarà:

$$0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000_{due}$$

Come si può vedere, l'operazione AND può essere utilizzata per sovrapporre una certa combinazione di bit ad un gruppo di bit preesistente, in maniera tale da ottenere uno 0 laddove vi è uno 0 nella combinazione di bit. L'utilizzo di una combinazione di bit in congiunzione con l'operazione AND è chiamata tradizionalmente *maschera*, dal momento che ha l'effetto di "nascondere" alcuni bit.

Esiste un'operazione duale a quella di AND, denominata OR. Si tratta di un'operazione bit a bit che scrive un 1 nel risultato se l'uno o l'altro dei bit degli operandi è a 1. Per sviluppare il concetto, si supponga che i registri `$t1` e `$t2` non siano cambiati rispetto all'esempio precedente; in tal caso il risultato dell'istruzione MIPS

```
or  $t0,  $t1,  $t2   # reg $t0 = reg $t1 | reg $t2
```

è il seguente valore nel registro `$t0`:

$$00000000\ \ 0000\ \ 0000\ \ 0011\ \ 1101\ \ 0000\ \ 0000_{due}$$

La figura 4.6 mostra le operazioni logiche del linguaggio C e le istruzioni MIPS corrispondenti. Le costanti sono utili per le operazioni logiche quanto per quelle aritmetiche e quindi il MIPS fornisce anche le istruzioni *and immediate* (`andi`) e *or immediate* (`ori`). Questo paragrafo descrive le operazioni logiche AND, OR e di scalamento che oggi si possono trovare in ogni calcolatore. Le istruzioni logiche sono evidenziate nella figura 4.7, che riassume le istruzioni MIPS viste sinora.

Operazione logica	Operatore C	Istruzione MIPS
Scalamento a sinistra	<<	sll
Scalamento a destra	>>	srl
AND bit a bit	&	and, andi
OR bit a bit	|	or, ori

Figura 4.6 Le operazioni logiche e le corrispondenti operazioni in C e nel MIPS.

Operandi MIPS

Nome	Esempio	Commenti
32 registri	$s0-$s7, $t0-$t9, $gp, $fp, $zero, $sp, $ra, $at	Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. Il registro $zero ha sempre valore 0. Il registro $at è riservato all'assembler per la gestione di costanti di grandi dimensioni.
2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte e quindi le parole successive differiscono di 4 unità. La memoria mantiene le strutture dati, come i vettori, i registri riversati (spilled) e i registri salvati durante le chiamate a procedura.

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	add $s1, $s2, $s3	$s1 = $s2 + $s3	Tre operandi; dati nei registri
	sottrazione	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Tre operandi; dati nei registri
	somma immediata	addi $s1, $s2, 100	$s1 = $s2 + 100	Usata per sommare costanti
	somma senza segno	addu $s1, $s2, $s3	$s1 = $s2 + $s3	Tre operandi; nessun test sull'overflow
	sottrazione senza segno	subu $s1, $s2, $s3	$s1 = $s2 - $s3	Tre operandi; nessun test sull'overflow
	somma immediata senza segno	addiu $s1, $s2, 100	$s1 = $s2 + 100	Somma una costante; nessun test sull'overflow
	sposta dal registro del coprocessore	mfc0 $s1, $epc	$s1 = $epc	Copia l'Exception PC ed altri registri speciali
Logica	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	Tre operandi registri; AND bit a bit
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	Tre operandi registri; OR bit a bit
	and immediato	andi $s1, $s2, 100	$s1 = $s2 & 100	AND bit a bit tra registro e costante
	or immediato	ori $s1, $s2, 100	$s1 = $s2 | 100	OR bit a bit tra registro e costante
	scalamento logico a sinistra	sll $s1, $s2, 10	$s1 = $s2 << 10	Scalamento a sinistra di una costante
	scalamento logico a destra	srl $s1, $s2, 10	$s1 = $s2 >> 10	Scalamento a destra di una costante
Trasferimento dati	load word	lw $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Word dalla memoria al registro
	store word	sw $s1, 100($s2)	Memoria[$s2 + 100] = $s1	Word dal registro alla memoria
	load byte unsigned	lbu $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Byte dalla memoria al registro
	store byte	sb $s1, 100($s2)	Memoria[$s2 + 100] = $s1	Byte dal registro alla memoria
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Carica una costante nei 16 bit più significativi
Salto condizionato	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Controllo dell'uguaglianza ed eventuale salto; salto PC-relative
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Controllo dell'uguaglianza ed eventuale salto; salto PC-relative
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0;	Confronto del tipo "minore di"; complemento a due
	set less than immediate	slti $s1, $s2, 100	if ($s2 < $s3) $s1 = 1; else $s1 = 0;	Confronto con una costante; complemento a due
	set less than unsigned	sltu $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0;	Confronto del tipo "minore di"; numeri senza segno
	set on less than immediate unsigned	sltiu $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0;	Confronto con una costante; numeri senza segno
Salto incondizionato	jump	j 2500	go to 10000	Salto ad un indirizzo specificato
	jump register	jr $ra	go to $ra	Usato nei costrutti switch e nell'uscita dalle procedure
	jump and link	jal 2500	$ra = PC + 4; go to 10000	Per le chiamate a procedura

Figura 4.7 L'architettura del MIPS introdotta sinora. Sono evidenziate le parti aggiunte rispetto alla figura 4.5. Il linguaggio dell'elaboratore MIPS è riassunto nelle ultime pagine del libro.

Interfaccia Hardware Software

Il C prevede la possibilità di definire *campi di bit* (bit field) o *campi* (field) all'interno delle parole, permettendo così sia di compattare oggetti all'interno di una parola, sia di rispettare un'interfaccia imposta dall'esterno, come quella di un dispositivo di I/O. Tutti i campi devono essere contenuti all'interno di una singola parola. I campi possono essere numeri interi senza segno lunghi anche un solo bit. I compilatori C del MIPS inseriscono ed estraggono i campi dalle parole usando le istruzioni logiche and, or, sll e srl.

Esempio

I campi di bit in C

Il seguente codice C alloca tre campi e li associa all'etichetta receiver: un campo da 1 bit denominato ready, un campo da 1 bit denominato enable ed un campo da 8 bit denominato receivedByte. Il codice copia il campo receivedByte nella variabile data, forza ready a 0 e pone enable a 1.

```
int data;
struct
{
    unsigned int ready:       1;
    unsigned int enable:      1;
    unsigned int receivedByte:8;
}receiver;
...
    data = receiver.receivedByte;
    receiver.ready = 0;
    receiver.enable = 1;
```

Qual è il codice MIPS compilato? Si supponga che data e receiver siano allocati nei registri $s0 e $s1.

Soluzione

I campi considerati avranno la seguente posizione all'interno di una word (il C allinea i campi a destra):

31 ... 10	9 ... 2	1	0
	receivedByte	enable	ready

Il primo passo serve per isolare il campo di 8 bit (receivedByte), prima facendolo scalare a sinistra il più possibile, e poi a destra il più possibile:

```
sll   $s0, $s1, 22      # sposta il campo di 8 bit
                        # al margine sinistro
srl   $s0, $s0, 24      # sposta il campo da 8 bit
                        # al margine destro
```

La terza istruzione ripulisce il bit meno significativo utilizzando la maschera $fffe_{esa}$, e l'ultima istruzione forza il bit vicino a 1:

```
andi  $s1, $s1, fffe_esa  # forza il bit 0 a 0
ori   $s1, $s1, 0002_esa  # forza il bit 1 a 1
```

Approfondimento: La seguente sequenza funziona altrettanto bene nell'esempio riportato:

```
srl   $s0,  $s1,  2
andi  $s0,  $s0,  0x00ff
```

Il campo si trova nei 16 bit meno significativi della parola e si vogliono avere degli 0 nei bit più significativi del risultato dell'istruzione andi. In generale, uno scalamento a sinistra di *32–(n+m)* seguito da uno scalamento a destra di *32-n* isolerà un qualunque campo di *n* bit il cui bit meno significativo si trovi nella posizione *m*-esima.

Poiché `addi` e `slti` sono pensati per i numeri con segno, non deve stupire se i loro campi immediato sono sottoposti all'estensione del segno prima dell'uso. Anche i campi che contengono indirizzi di salto e di trasferimento dati subiscono l'estensione del segno.

Forse stupirà invece che `addiu` e `sltiu` eseguano l'estensione del segno sui loro operandi, ma così è. La `u` sta per senza segno, ma in realtà `addiu` è spesso usata semplicemente come fosse un'istruzione `add` che non può causare overflow, e quindi viene frequentemente usata per sommare numeri negativi. È invece molto più difficile trovare una ragione per cui `sltiu` non debba eseguire l'estensione del segno sul proprio operando immediato.

Dal momento che `andi` e `ori` lavorano normalmente su interi senza segno, gli operandi immediati sono trattati anch'essi come interi senza segno, e quindi sono estesi su 32 bit riempiendo lo spazio vuoto con degli 0 iniziali anziché estendendo il segno. Di conseguenza se i campi di bit nella terza riga dell'esempio si fossero estesi al di là dei 16 bit meno significativi, l'istruzione `andi` avrebbe richiesto una costante su 32 bit per evitare di dover ripulire la parte superiore dei campi.

L'assemblatore MIPS crea delle costanti su 32 bit tramite le istruzioni `lui` e `ori`; si veda il Paragrafo 3.8 per un esempio sulla creazione di costanti su 32 bit utilizzando `lui` e `ori`.

4.5 La costruzione di una unità aritmetico logica

*ALU [**A**rthritic **L**ogic **U**nit (**U**nità **L**ogica **A**rtritica) o (raro) Arithmetic Logic Unit (Unità Aritmetico Logica)] Un generatore di numeri casuali fornito regolarmente con tutti i sistemi di elaborazione.*

Stan Kelly-Bootle, *The Devil's DP Dictionary*, 1981

L'*Unità Aritmetico Logica* o ALU (*Arithmetic Logic Unit*) rappresenta l'apparato muscolare di un calcolatore, il dispositivo cioè che esegue le operazioni aritmetiche, come la somma o la sottrazione, o quelle logiche, come l'AND o l'OR. Questo paragrafo illustra come costruire una ALU a partire dai quattro blocchi elementari riportati nella figura 4.8 (si veda l'Appendice B per maggiori dettagli su tali blocchi). I casi 1, 2 e 4 nella figura 4.8 hanno sempre due ingressi. Si useranno a volte delle versioni di questi stessi componenti caratterizzate da più di due ingressi, nella certezza che il lettore sia in grado di generalizzare a partire da questi semplici esempi. In ogni caso l'Appendice B fornisce degli esempi in cui compaiono più ingressi. Potrebbe essere opportuno rivedere i paragrafi dal B.1 al B.3 prima di procedere oltre.

Dal momento che la parola del MIPS ha ampiezza pari a 32 bit, sarà necessario considerare una ALU a 32 bit. Si assuma di collegare 32 ALU a 1 bit per ottenere la ALU desiderata; si partirà quindi costruendo una ALU a 1 bit.

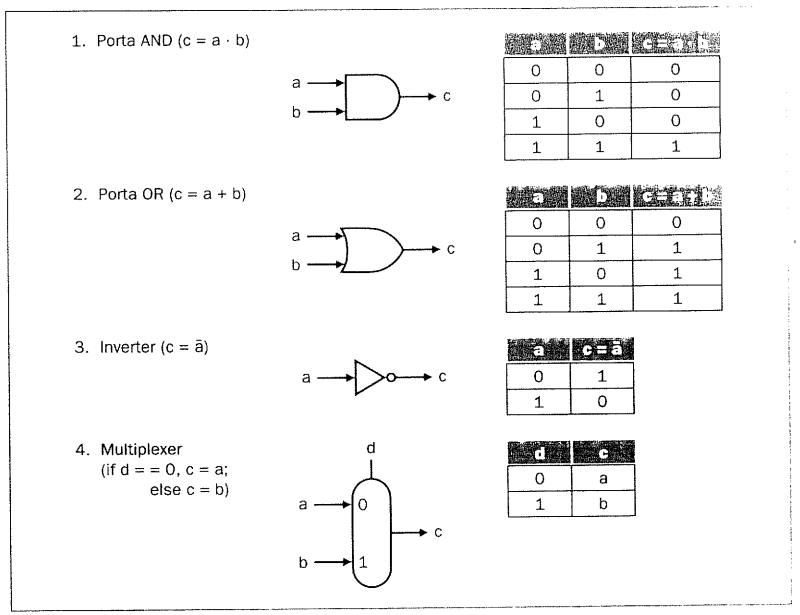


Figura 4.8 I quattro componenti hardware elementari utilizzati per la costruzione di una unità aritmetico logica. Sulla sinistra è riportato il nome dell'operazione ed un'equazione che la descrive. Al centro vi è il simbolo del blocco che verrà usato negli schemi successivi. Sulla destra vi sono le tabelle che descrivono le uscite in termini di ingressi. Utilizzando la stessa notazione della Appendice B, a·b significa "a AND b", a+b significa "a OR b", ed una linea al di sopra (ad esempio ā) significa inversione.

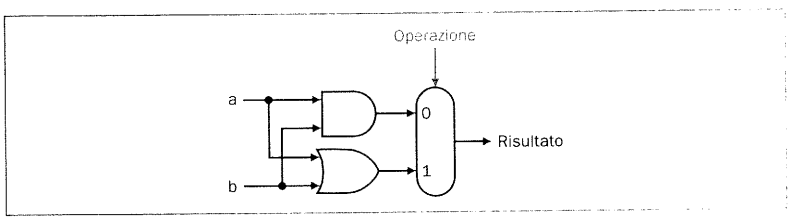


Figura 4.9 L'unità logica a 1 bit che esegue AND e OR.

La ALU a 1 bit

Le operazioni logiche sono le più semplici da implementare, in quanto corrispondono direttamente ai componenti hardware della figura 4.8.

L'unità logica che esegue le operazioni di AND e OR ha l'aspetto riportato nella figura 4.9. Il multiplexer sulla destra seleziona *a AND b* oppure *a OR b*, a seconda che il valore del segnale *Operazione* sia 0 o 1. La linea che controlla il multiplexer è colorata per distinguerla dalle linee che contengono i dati. Si noti che sono stati cambiati i nomi delle linee di controllo e di uscita del multiplexer, per dare loro dei nomi che rispecchiassero le funzioni svolte dalla ALU.

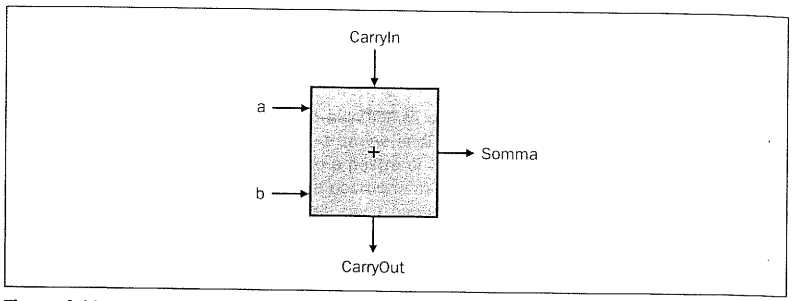


Figura 4.10 Sommatore a 1 bit. Il sommatore mostrato è denominato full adder; è anche chiamato sommatore (3,2) in quanto possiede 3 ingressi e 2 uscite. Un sommatore avente soltanto gli ingressi a e b è chiamato sommatore (2,2) o half adder.

Ingressi			Uscite		Commenti
a	b	CarryIn	CarryOut	Somma	
0	0	0	0	0	$0+0+0=00_{due}$
0	0	1	0	1	$0+0+1=01_{due}$
0	1	0	0	1	$0+1+0=01_{due}$
0	1	1	1	0	$0+1+1=10_{due}$
1	0	0	0	1	$1+0+0=01_{due}$
1	0	1	1	0	$1+0+1=10_{due}$
1	1	0	1	0	$1+1+0=10_{due}$
1	1	1	1	1	$1+1+1=11_{due}$

Figura 4.11 Specifiche di ingresso e di uscita per un sommatore a 1 bit.

La funzione da realizzare subito dopo è la somma. Dalla figura 4.3 si può dedurre quali siano gli ingressi e le uscite di un sommatore a 1 bit. Innanzitutto un sommatore deve avere due ingressi per gli operandi ed una uscita di un bit per la somma. Deve esserci una seconda uscita per trasferire il riporto (*carry*), denominata *CarryOut*. Dal momento che il CarryOut proveniente dal sommatore adiacente deve essere considerato come ingresso, è necessario un terzo ingresso, denominato *CarryIn*. La figura 4.10 mostra gli ingressi e le uscite di un sommatore a 1 bit. Poiché la somma è un'operazione nota, si possono specificare i valori delle uscite di questa "scatola nera" sulla base dei suoi ingressi, come è stato fatto nella figura 4.11.

Da quanto detto nell'Appendice B, è noto che si possono esprimere le funzioni di uscita dei segnali CarryOut e Somma in termini di equazioni logiche, e che queste equazioni possono a loro volta essere implementate tramite i blocchi elementari della figura 4.8. Si consideri ad esempio CarryOut: la figura 4.12 mostra i valori degli ingressi quando CarryOut vale 1.

Si può convertire questa tabella della verità in un'equazione logica, come spiegato nell'Appendice B. Si ricordi che a+b significa "a OR b" e che a·b significa "a AND b".

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

Se a·b·CarryIn è vero, allora uno degli altri tre termini deve essere anch'esso vero, e quindi si può trascurare l'ultimo termine dell'equazione, corrispondente alla quarta riga della tabella. Si può quindi semplificare l'equazione:

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

Ingressi		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

Figura 4.12 Valori degli ingressi quando CarryOut vale 1.

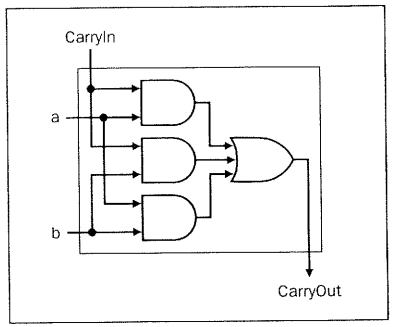


Figura 4.13 Hardware per la generazione del segnale di Carry Out all'interno del sommatore. Il resto della circuiteria del sommatore è costituito dalla logica per la generazione dell'uscita Somma, come dalla formula riportata.

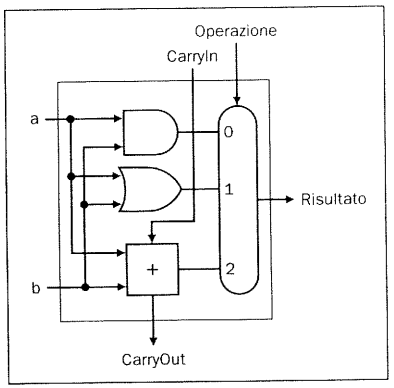


Figura 4.14 Una ALU a 1 bit che esegue le operazioni di AND, OR e somma (si veda la figura 4.13).

La figura 4.13 mostra che l'hardware che genera il CarryOut all'interno della scatola nera del sommatore consiste di tre porte AND ed una porta OR. Le tre porte AND corrispondono esattamente ai tre termini tra parentesi nella formula precedente, e la porta OR esegue la somma dei tre termini.

Il bit Somma vale 1 quando un solo ingresso vale 1, oppure quando tutti e tre gli ingressi valgono 1. Il segnale Somma corrisponde ad un'equazione Booleana complessa (si ricordi che $\bar{a}$ significa NOT a):

$$\text{Somma} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

Il disegno della logica per il bit Somma nella scatola nera del sommatore è lasciato come esercizio (si veda l'Esercizio 4.43).

La figura 4.14 mostra la ALU a 1 bit ottenuta dalla combinazione del sommatore e delle parti precedentemente viste. Talvolta i progettisti vogliono che la ALU esegua alcune ulteriori semplici operazioni, come la generazione dello 0. Il modo più semplice per aggiungere un'operazione è quello di espandere il multiplexer controllato dal segnale Operazione e, per esempio, di collegare lo 0 direttamente al nuovo ingresso del multiplexer espanso.

La ALU a 32 bit

Ora che si è visto il progetto completo di una ALU a 1 bit, è possibile creare una ALU completa a 32 bit collegando insieme un certo numero di "scatole nere"; usando x_i per indicare l'i-esimo bit di x, la figura 4.15 mostra una ALU a 32 bit. Come

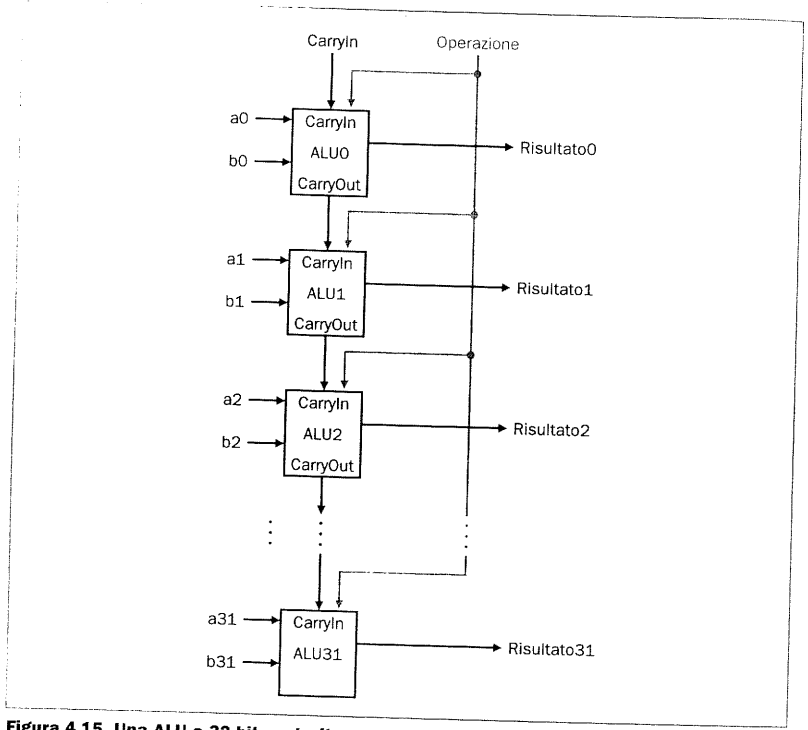


Figura 4.15 Una ALU a 32 bit costruita a partire da 32 ALU a 1 bit. Il CarryOut del bit meno significativo è collegato al CarryIn del bit più significativo. Questa organizzazione si chiama a propagazione di riporto (ripple carry).

un singolo sasso può causare delle onde che si propagano sino alle sponde di un lago tranquillo, così un singolo segnale di carry out proveniente dal bit meno significativo (Risultato0) può propagarsi attraverso l'intero sommatore, causando la generazione di un riporto da parte del bit più significativo (Risultato31). Per questa ragione il sommatore che si ottiene collegando direttamente i riporti dei sommatori a 1 bit si chiama sommatore a propagazione di riporto (*ripple carry adder*). Una maniera più veloce per collegare i sommatori a 1 bit verrà presentata nel seguito di questo paragrafo.

La sottrazione corrisponde alla somma con la versione negata di un operando, ed è in questo modo che i sommatori eseguono la sottrazione. Si ricordi che la scorciatoia per negare un numero in complemento a due consiste nell'invertire ciascun bit (ottenendo ciò che viene chiamato il complemento a 1, come illustrato nell'approfondimento al termine del Paragrafo 4.2) e nel sommare poi il valore 1. Per invertire ciascun bit, è sufficiente aggiungere un multiplexer 2:1 che scelga tra b e $\overline{b}$, come mostrato nella figura 4.16.

Si supponga di collegare 32 di queste ALU a 1 bit, come si è fatto nella figura 4.15. L'aggiunta del multiplexer dà la possibilità di generare il valore di b o del suo inverso, a seconda del valore di InvertiB, ma questo è soltanto il primo passo

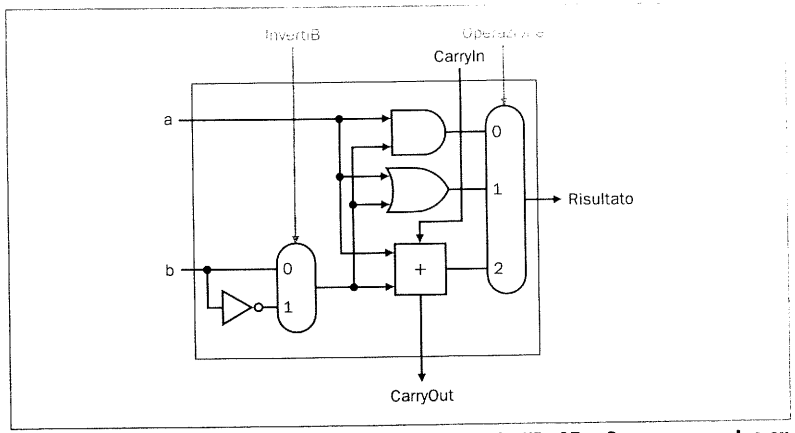


Figura 4.16 Una ALU a 1 bit che esegue le operazioni di AND, OR e Somma su a e b o su a e $\overline{b}$. Scegliendo b (Inverti B=1) e forzando CarryIn a 1 nel bit meno significativo della ALU si ottiene la sottrazione in complemento a due di b da a anziché la somma di b ad a.

nel processo di negazione di un numero in complemento a due. Si noti che il bit meno significativo possiede un segnale di CarryIn, anche se questo non è necessario per la somma. Cosa succede se si forza questo CarryIn a 1 anziché a 0? In questo caso il sommatore eseguirà il calcolo a+b+1. Se si seleziona la versione invertita di b, si ottiene esattamente ciò che si desiderava:

$$a + \overline{b} + 1 = a + (\overline{b} + 1) = a + (-b) = a - b$$

La semplicità del progetto della logica che implementa un sommatore per numeri in complemento a due aiuta a comprendere perché la rappresentazione in complemento a due sia divenuta uno standard universale per l'aritmetica intera su calcolatore.

Una ALU a 32 bit su misura per il MIPS

L'insieme di operazioni considerato, costituito da somma, sottrazione, AND e OR, si trova implementato nella ALU di quasi tutti i calcolatori. Se si dà un'occhiata alla figura 4.7, si vede che le operazioni previste da gran parte delle istruzioni del MIPS possono essere eseguite da una ALU di questo tipo. Ma il progetto della ALU non è completo.

Un'istruzione che ancora richiede un supporto hardware è la "set on less than" (`slt`). Si ricordi che essa genera un 1 se rs < rt, uno 0 altrimenti. Di conseguenza `slt` forzerà tutti i bit, tranne quello meno significativo, al valore 0, mentre il bit meno significativo verrà forzato ad un valore che dipende dal risultato del confronto. Per far sì che la ALU esegua la `slt`, si deve innanzitutto espandere il multiplexer a tre ingressi della figura 4.16, aggiungendo un ingresso per il risultato della `slt`. Il nuovo ingresso prende il nome *Less* e verrà usato esclusivamente per la `slt`.

Il disegno nella parte alta della figura 4.17 mostra la nuova ALU a 1 bit con il multiplexer espanso. Dalla descrizione che è stata fatta di `slt`, si deduce che si deve connettere il valore 0 all'ingresso Less dei 31 moduli corrispondenti ai bit superiori della ALU, dal momento che questi bit sono sempre forzati a 0. Ciò che rima-

ne da capire è come confrontare e forzare al valore corretto il bit meno significativo nel caso delle istruzioni "set on less than".

Cosa avviene se si sottrae b da a? Se la differenza è negativa, allora a<b, in quanto

$$(a - b) < 0 \Rightarrow ((a - b) + b) < (0 + b)$$
$$\Rightarrow a < b$$

Si vuole che il bit meno significativo di un'operazione di "set on less than" valga 1 se a<b; in altre parole deve valere 1 se a–b è negativo e 0 se è positivo. Il risultato desiderato corrisponde esattamente al valore del bit di segno: 1 significa negativo e 0 significa positivo. Seguendo questo ragionamento, è sufficiente collegare al bit meno significativo il bit di segno che proviene dall'uscita del sommatore per ottenere l'operazione "set on less than".

Purtroppo nel caso dell'istruzione `slt` l'uscita Risultato proveniente dal modulo corrispondente al bit più significativo della ALU nella parte alta della figura 4.17 non è l'uscita del sommatore: l'uscita della ALU per l'operazione `slt` è ovviamente il valore dell'ingresso Less.

Bisogna quindi disporre di una nuova ALU a 1 bit per il bit meno significativo, la quale possieda un ulteriore bit di uscita: l'uscita del sommatore. La parte bassa della figura 4.17 mostra il progetto, in cui è stata aggiunta la nuova uscita del sommatore, denominata *Set*, utilizzata soltanto per `slt`. Tutte le volte che è necessaria una ALU speciale per il bit più significativo si aggiungerà la logica per il rilevamento dell'overflow, dal momento che anch'essa è associata a tale bit.

In realtà il test di minore è leggermente più complicato di quanto descritto, a causa dell'overflow; l'Esercizio 4.23 analizza più a fondo questo problema. La figura 4.18 mostra la ALU a 32 bit.

Si noti che tutte le volte che si vuole che la ALU esegua una sottrazione, si forza sia CarryIn che InvertiB a 1. Per la somma e le operazioni logiche ambedue i segnali di controllo sono invece forzati a 0. Si può quindi semplificare il controllo della ALU combinando insieme CarryIn e InvertiB e introducendo una nuova linea di controllo denominata *NegaB*.

Per specializzare ulteriormente la ALU rispetto al set di istruzioni MIPS, si devono supportare le istruzioni di salto condizionato. Queste istruzioni eseguono il salto quando i due registri sono uguali oppure quando sono diversi. Il modo più semplice per controllare l'uguaglianza tramite la ALU consiste nel sottrarre b da a e nel verificare se il risultato è 0, dal momento che

$$(a - b = 0) \Rightarrow a = b$$

Quindi aggiungendo della logica per verificare se il risultato è 0, si può poi eseguire il controllo di uguaglianza.

Il modo più semplice consiste nell'eseguire un OR tra tutte le uscite e nel far passare il segnale attraverso un inverter:

$$\text{Zero} = \overline{(\text{Risultato31} + \text{Risultato30} + \ldots + \text{Risultato2} + \text{Risultato1} + \text{Risultato0})}$$

La figura 4.19 mostra la ALU a 32 bit aggiornata. Si può vedere la combinazione della linea di 1 bit NegaB e delle linee di 2 bit Operazione come una linea di controllo a 3 bit per la ALU, che indicano se eseguire una somma, una sottrazione, un AND, un OR, o un'operazione di "set on less than". La figura 4.20 mostra le linee di controllo della ALU e le corrispondenti operazioni svolte.

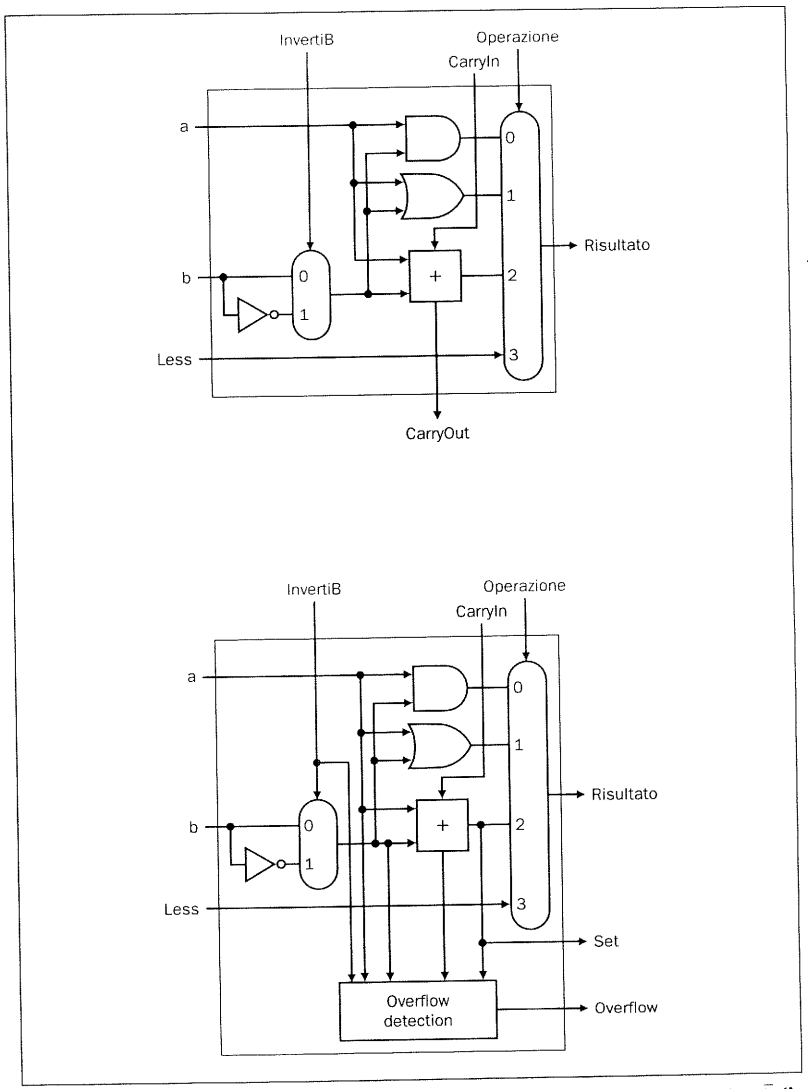


Figura 4.17 Una ALU a 1 bit che esegue le operazioni di AND, OR, somma su a e b o $\bar{b}$ (in alto), e una ALU a 1 bit per il bit più significativo (in basso). Il disegno in alto prevede un ingresso diretto per eseguire l'operazione di "set on less than" (si veda la figura 4.18); nella parte bassa compare un'uscita del sommatore (denominata Set) per supportare il confronto di minore. Si faccia riferimento all'Esercizio 4.42 per vedere come calcolare l'overflow con un numero minore di ingressi.

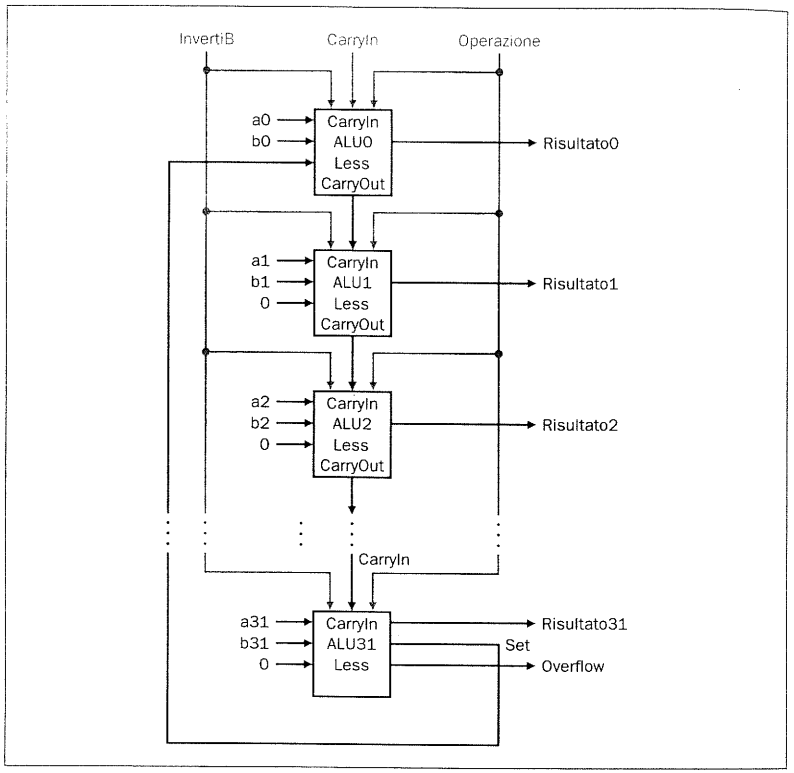


Figura 4.18 Una ALU a 32 bit costruita a partire da 31 copie della ALU a 1 bit della parte alta della figura 4.17 e da una copia della ALU a 1 bit tratta dalla parte bassa della stessa figura. Gli ingressi Less sono collegati a 0 ad esclusione del bit meno significativo, che è collegato all'uscita Set del bit più significativo. Se la ALU esegue a–b e si seleziona l'ingresso 3 del multiplexer della figura 4.17, allora Risultato = 0 … 001 se a < b e Risultato = 0 … 000 in caso contrario.

Infine ora che si è visto cosa c'è dentro una ALU a 32 bit, si userà il simbolo universale per una ALU completa, come si vede nella figura 4.21.

Infine ora che si è visto che cosa vi sia all'interno di una ALU a 32 bit, verrà utilizzato il simbolo universale per la ALU completa, come mostrato in figura 4.21.

Il Carry Lookahead

Il punto successivo è: quanto è veloce la ALU appena progettata nell'eseguire la somma di due operandi su 32 bit? Si possono determinare gli ingressi a e b, ma l'ingresso CarryIn dipende dal funzionamento dell'adiacente sommatore a 1 bit. Se si risale lungo la catena delle dipendenze, il bit più significativo è collegato a quello meno significativo, ed il bit più significativo della somma deve attendere la valutazione *sequenziale* di tutti i 32 sommatori a 1 bit. Tale reazione sequenziale a catena è troppo lenta per poter essere usata in un dispositivo critico dal punto di vista della velocità.

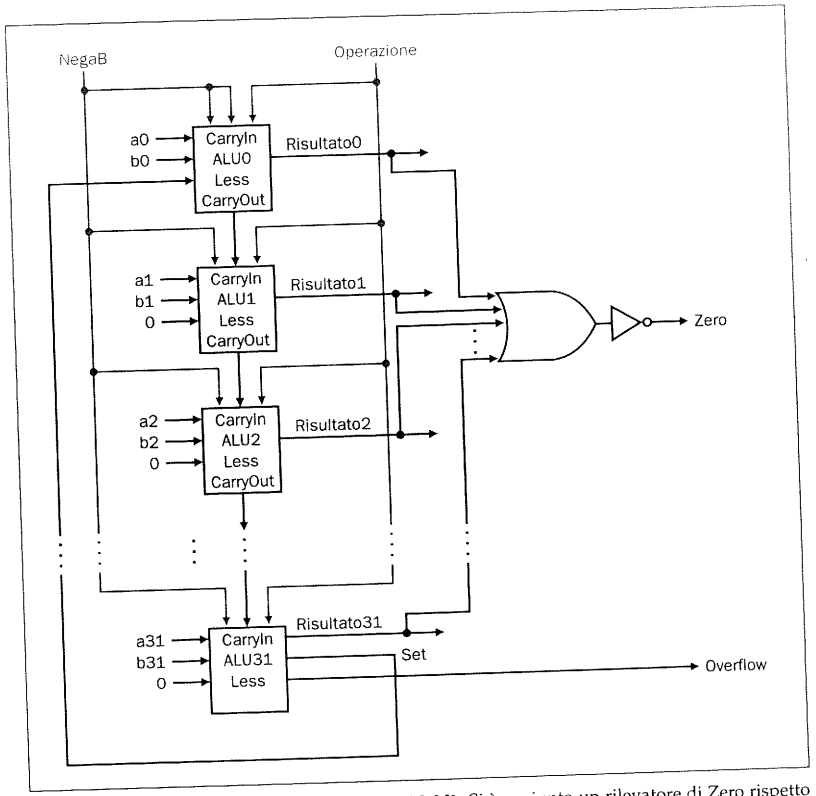


Figura 4.19 La versione finale della ALU a 32 bit. Si è aggiunto un rilevatore di Zero rispetto alla figura 4.18.

Linee di controllo della ALU	Funzione
000	and
001	or
010	somma
110	sottrazione
111	set on less than

Figura 4.20 I valori delle tre linee di controllo della ALU NegaB e Operazione e le corrispondenti operazioni della ALU.

Vi sono diversi metodi che permettono di anticipare il carry così che il caso peggiore di propagazione sia proporzionale ad una funzione del logaritmo in base 2 del numero di bit del sommatore e non al numero di bit. Questi segnali anticipati sono più veloci in quanto passano attraverso un numero minore di porte logiche, ma sarà necessario un numero maggiore di porte per anticipare opportunamente il riporto.

La chiave per capire gli schemi veloci di riporto sta nel ricordare che, a differenza del software, l'hardware esegue in parallelo più operazioni ogni volta che gli ingressi cambiano valore.

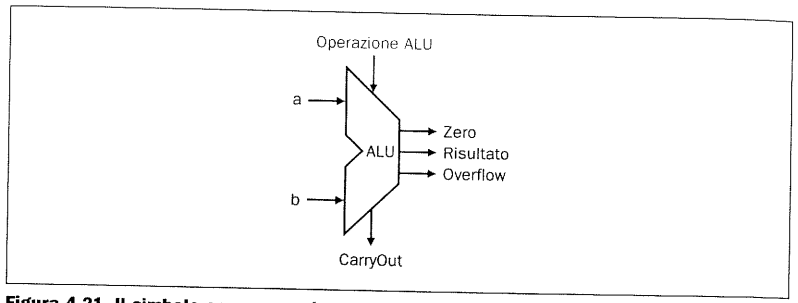


Figura 4.21 Il simbolo comunemente usato per rappresentare una ALU, come riportata in figura 4.19. Il simbolo è anche usato per rappresentare un sommatore e normalmente porta quindi l'etichetta ALU oppure Somma.

Il riporto veloce grazie ad un hardware "infinito"

Nell'Appendice B si afferma che qualunque equazione può essere rappresentata tramite due livelli di logica. Dal momento che i soli ingressi esterni sono costituiti dai due operandi e dal CarryIn entrante nello stadio meno significativo del sommatore, si potrebbero in teoria calcolare i valori di CarryIn per tutti gli altri stadi del sommatore in due soli livelli di logica.

Ad esempio, il CarryIn per il bit 2 del sommatore è esattamente il CarryOut del bit 1, e quindi la formula è

$$\text{CarryIn2} = (b1 \cdot \text{CarryIn1}) + (a1 \cdot \text{CarryIn1}) + (a1 \cdot b1)$$

Analogamente il CarryIn1 è definito come

$$\text{CarryIn1} = (b0 \cdot \text{CarryIn0}) + (a0 \cdot \text{CarryIn0}) + (a0 \cdot b0)$$

Se si utilizza la più concisa e diffusa abbreviazione ci per CarryIni, si possono riscrivere le formule come segue:

$$c2 = (b1 \cdot c1) + (a1 \cdot c1) + (a1 \cdot b1)$$
$$c1 = (b0 \cdot c0) + (a0 \cdot c0) + (a0 \cdot b0)$$

Sostituendo la definizione di c1 nella prima equazione si ottiene

$$c2 = (a1 \cdot a0 \cdot b0) + (a1 \cdot a0 \cdot c0) + (a1 \cdot b0 \cdot c0) + (b1 \cdot a0 \cdot b0) + (b1 \cdot a0 \cdot c0) + (b1 \cdot b0 \cdot c0) + (a1 \cdot b1)$$

Si può immaginare quanto si espanda l'equazione mano a mano che si passa ai bit di ordine più elevato nel sommatore; la complessità cresce esponenzialmente con il numero di bit. Tale complessità si riflette sul costo dell'hardware per il riporto veloce, rendendo questo schema proibitivamente costoso per sommatori di grandi dimensioni.

Il riporto veloce utilizzando il primo livello di astrazione: propagazione e generazione

La maggior parte degli schemi di riporto veloce limitano la complessità delle equazioni al fine di semplificare il circuito logico, ottenendo tuttavia dei significativi miglioramenti in termini di velocità rispetto al *sommatore a propagazione di riporto*. Nel Capitolo 1 si è detto che i sistemi di elaborazione affrontano la complessità utilizzando livelli diversi di astrazione. L'implementazione di un sommatore con carry-lookahead (che significa *anticipazione del riporto*) si basa proprio su diversi livelli di astrazione.

Come primo passo si fattorizzi l'equazione originale:

$$ci + 1 = (bi \cdot ci) + (ai \cdot ci) + (ai \cdot bi) = (ai \cdot bi) + (ai + bi) \cdot ci$$

Se si deve riscrivere l'equazione per c2 utilizzando la stessa formula, ci si accorge che vi sono delle combinazioni che si ripetono:

$$c2 = (a1 \cdot b1) + (a1 + b1) \cdot ((a0 \cdot b0) + (a0 + b0) \cdot c0)$$

Si noti la presenza ripetuta di ($ai \cdot bi$) e ($ai+bi$) nella formula. Questi due importanti fattori sono tradizionalmente chiamati *generazione* (gi) e *propagazione* (pi):

$$gi = ai \cdot bi$$

$$pi = ai + bi$$

Utilizzando tali fattori nella definizione di $ci+1$, si ottiene

$$ci+1 = gi + pi \cdot ci$$

Se si vuole capire da dove prendono il loro nome i due segnali, si supponga che gi valga 1. Allora

$$ci + 1 = gi + pi \cdot ci = 1 + pi \cdot ci = 1$$

Ciò significa che il sommatore *genera* il CarryOut ($ci+1$) indipendentemente dal valore del CarryIn (ci). Si supponga ora che gi valga 0 e pi valga 1. Allora

$$ci + 1 = gi + pi \cdot ci = 0 + 1 \cdot ci = ci$$

Quindi il sommatore *propaga* il CarryIn sul CarryOut. Mettendo insieme le due osservazioni, si deduce che CarryIn$i+1$ vale 1 se gi vale 1 oppure pi è 1 e CarryIni è 1.

Facendo un'analogia, si immagini una fila di pezzi del domino messi in piedi, uno appoggiato all'altro. L'ultimo pezzo può essere fatto cadere spingendo giù un pezzo lontano, purché non ci siano buchi tra i due. Analogamente un riporto in uscita può essere reso vero da un segnale genera in una posizione lontana, purché tutti i segnali propaga esistenti tra i due siano al valore vero.

Utilizzando le definizioni di propagazione e generazione come primo livello di astrazione, si possono esprimere i segnali CarryIn in maniera più conveniente. Per 4 bit si ha

$$c1 = g0 + (p0 \cdot c0)$$

$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot g1) + (p0 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

$$c4 = g3 + (p3 \cdot g2) + (p0 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

Queste equazioni non aggiungono nulla a ciò che si ottiene con il buon senso: il segnale CarryIn vale 1 se qualche sommatore precedente ha generato un riporto e tutti i sommatori intermedi lo propagano. La figura 4.22 usa una metafora idraulica per tentare di spiegare il carry-lookahead.

Anche questa formulazione semplificata conduce comunque a lunghe equazioni e quindi ad una logica considerevolmente complessa anche per un sommatore a 16 bit. Si cercherà dunque di procedere all'adozione di due livelli di astrazione.

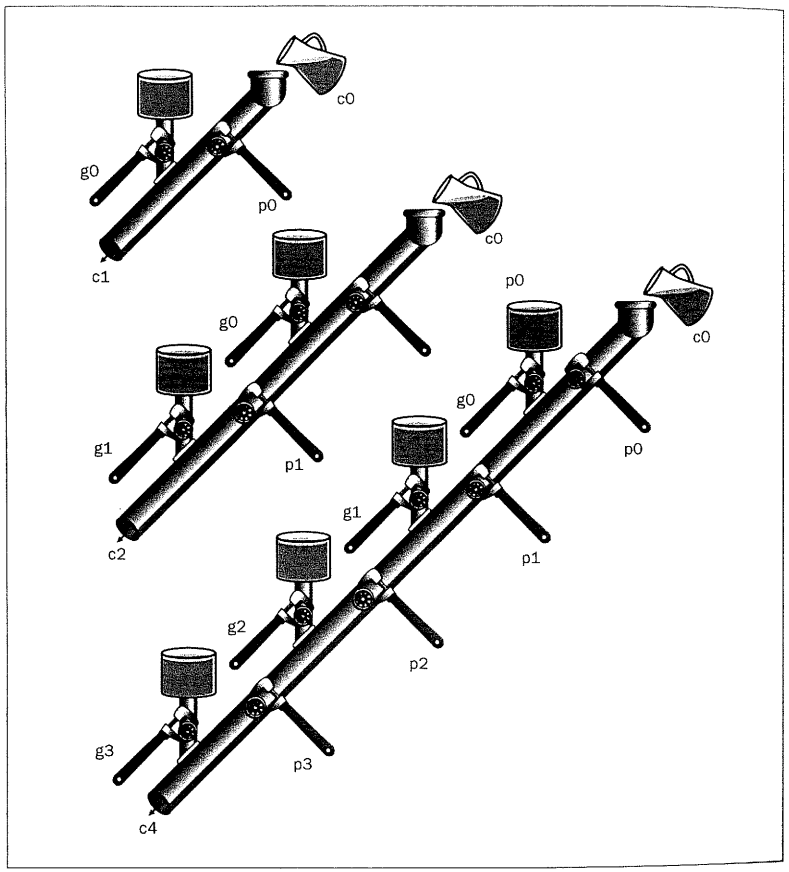


Figura 4.22 Un'analogia idraulica per il carry-lookahead di 1 bit, 2 bit e 4 bit utilizzando acqua, tubi, valvole. I rubinetti permettono di aprire e chiudere le valvole. L'acqua è rappresentata colorata. L'uscita del tubo (c_i+1) sarà piena se il valore del più vicino segnale genera (g_i) è aperto, oppure se il valore dell'i-esimo segnale propaga (p_i) è aperto e vi è dell'acqua a monte, proveniente o da un precedente genera, o da un propaga avente dell'acqua a monte. Il CarryIn (c0) può produrre un riporto in uscita senza l'aiuto di nessun genera, ma con l'aiuto di *tutti* i propaga.

Il riporto veloce utilizzando il secondo livello di astrazione

Innanzitutto si consideri il sommatore a 4 bit con la sua logica di carry-lookahead come un singolo blocco elementare. Se attraverso il carry si collegano dei blocchi simili in cascata per formare un sommatore a 16 bit, la somma sarà più veloce rispetto alla versione originale, con una modesta aggiunta di hardware.

Per andare più veloci è necessario un carry-lookahead di livello superiore. Per eseguire il carry-lookahead con i sommatori di 4 bit è necessario propagare e generare i segnali al livello superiore. Ecco le corrispondenti equazioni per quattro sommatori di 4 bit:

$$P0 = p3 \cdot p2 \cdot p1 \cdot p0$$
$$P1 = p7 \cdot p6 \cdot p5 \cdot p4$$
$$P2 = p11 \cdot p10 \cdot p9 \cdot p8$$
$$P3 = p15 \cdot p14 \cdot p13 \cdot p12$$

In pratica, il "super" segnale di propagazione per l'astrazione da 4 bit (P_i) è vero solo se ciascuno dei bit nel suo gruppo propaga un riporto.

Per quanto riguarda il "super" segnale genera (G_i), si deve soltanto tenere in conto se vi è un riporto in uscita dal bit più significativo del gruppo di 4 bit. Questo ovviamente avviene se genera è vero per il bit più significativo; avviene inoltre se un precedente genera è vero *e* tutti i propaga intermedi, incluso quello del bit più significativo, sono anch'essi veri:

$$G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)$$
$$G1 = g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)$$
$$G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)$$
$$G3 = g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)$$

La figura 4.23 aggiorna l'analogia idraulica in maniera da mostrare P0 e G0.

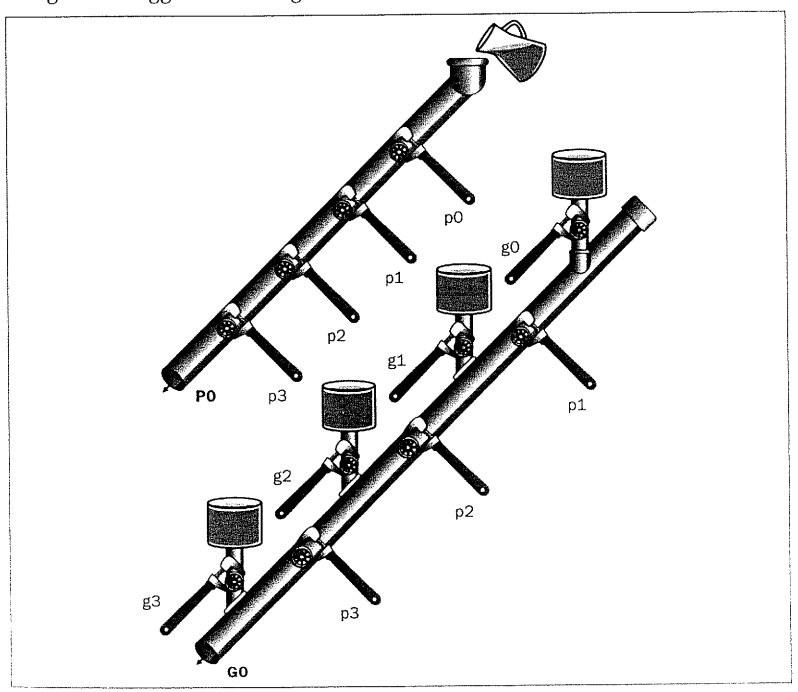


Figura 4.23 Analogia idraulica per il livello successivo di segnali di carry-lookahead P0 e G0. P0 è aperto solo se i quattro propaga (p_i) sono aperti, mentre l'acqua scorre in G0 solo se almeno un genera (g_i) è aperto e tutti i propaga a monte di tale genera sono aperti.

Quindi le equazioni per il riporto in ingresso ad ogni gruppo di 4 bit all'interno del sommatore a 16 bit (C1, C2, C3, C4) al livello superiore di astrazione sono molto simili alle equazioni per il riporto in uscita da ciascun bit del sommatore a 4 bit (c1, c2, c3, c4) scritte in precedenza:

$$C1 = G0 + (P0 \cdot c0)$$

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$

$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

La figura 4.24 mostra i sommatori a 4 bit connessi all'unità di carry-lookahead. Gli Esercizi dal 4.44 al 4.48 indagano sulle differenze di velocità tra questi schemi di riporto, sulle diverse notazioni per i segnali di propagazione e generazione su più bit, e sul progetto di un sommatore a 64 bit.

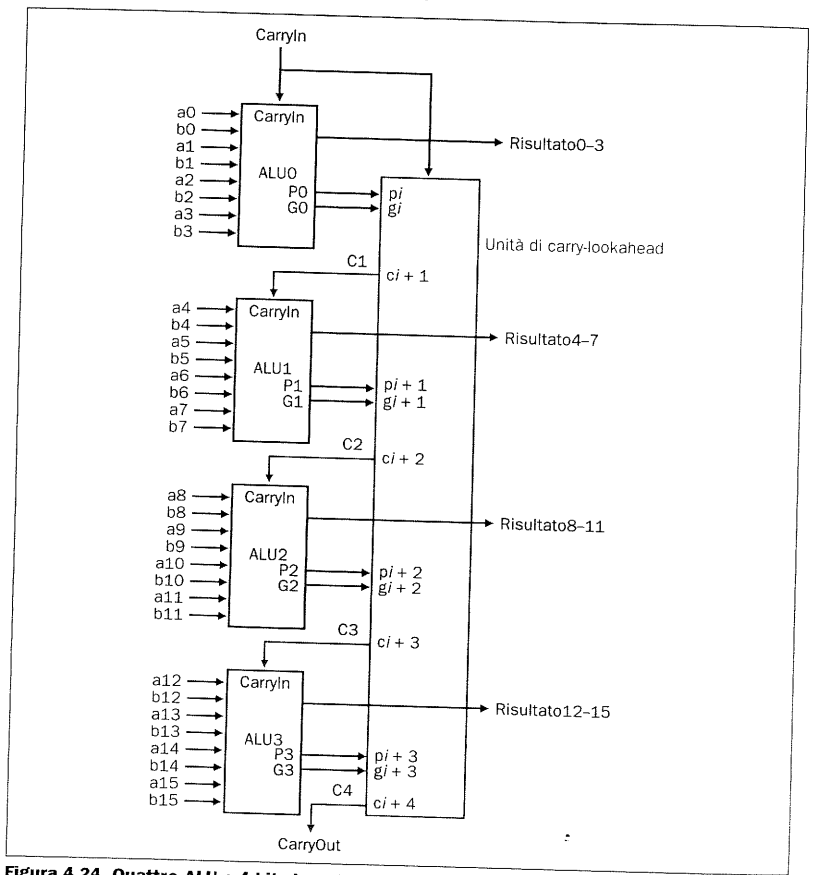


Figura 4.24 Quattro ALU a 4 bit che utilizzano il carry-lookahead per formare un sommatore a 16 bit. Si noti che i riporti provengono dall'unità di carry-lookahead e non dalle ALU a 4 bit.

I due livelli di propagazione e generazione

Esempio Si determinino i valori di gi, pi, Pi e Gi per i seguenti due numeri su 16 bit:

```
a:   0001  1010  0011  0011due
b:   1110  0101  1110  1011due
```

Inoltre che valore ha CarryOut15 (C4)?

Soluzione Allineando i bit è più semplice vedere i valori del segnale genera gi ($ai \cdot bi$) e propaga pi ($ai+bi$):

```
a:   0001  1010  0011  0011
b:   1110  0101  1110  1011
gi:  0000  0000  0010  0011
pi:  1111  1111  1111  1011
```

dove i bit sono numerati da 15 a 0 da sinistra a destra. Successivamente, i "super" propaga (P3, P2, P1, P0) sono semplicemente gli AND dei propaga di livello inferiore:

$$P3 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P2 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0$$

I "super" genera sono più complessi, per cui conviene usare le seguenti equazioni:

$$G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) = 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0$$

$$G1 = g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4) = 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1$$

$$G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8) = 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0$$

$$G3 = g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12) = 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0$$

Infine CarryOut15 è dato da

$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0) = 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) = 0 + 0 + 1 + 0 + 0 = 1$$

Quindi vi è un riporto in uscita quando si sommano i due numeri dati di 16 bit.

La ragione per cui il carry-lookahead rende il calcolo dei riporti più veloce è che tutta la logica inizia a lavorare appena inizia il ciclo di clock, ed il risultato non cambia più una volta che l'uscita di ciascuna porta logica ha smesso di cambiare.

Utilizzando un numero minore di porte logiche per trasmettere il segnale di riporto in ingresso, si ottiene che il tempo in cui l'uscita delle porte logiche varia è minore, e quindi il tempo di risposta del sommatore si riduce.

Per apprezzare l'importanza del carry-lookahead è necessario calcolare le prestazioni relative tra questo ed i sommatori a propagazione di riporto.

Confronto tra la velocità del sommatore a propagazione di riporto e quella del sommatore con carry-lookahead

Esempio Un modo semplice per modellare il tempo nell'analisi dell'hardware consiste nell'assumere che un segnale richieda lo stesso tempo per attraversare una qualunque porta AND o OR. Il tempo richiesto si calcola contando semplicemente il numero di porte logiche lungo il cammino più lungo che attraversa una porzione di logica. Si confronti il numero di ritardi introdotti dalle porte logiche lungo i cammini critici di due sommatori a 16 bit, uno basato su propagazione di riporto e l'altro su carry-lookahead a due livelli.

Soluzione La figura 4.13 mostra che il segnale di riporto in uscita richiede due volte il ritardo di una porta per ciascun bit. Quindi il rapporto tra il ritardo di una porta ed il tempo perché il riporto in ingresso al bit meno significativo raggiunga il riporto in uscita dal bit più significativo è pari a $16 \times 2 = 32$.

Per quanto riguarda il carry-lookahead, il riporto in uscita dal bit più significativo è semplicemente C4, definito nell'esempio. Sono necessari due livelli di logica per specificare C4 in termini di P_i e G_i (l'OR di diversi termini AND). P_i viene specificato in un livello di logica (AND) utilizzando i p_i, e G_i è specificato in due livelli utilizzando p_i e g_i, così che il caso peggiore per questo successivo livello di astrazione corrisponde a due livelli di logica. I termini p_i e g_i corrispondono ciascuno ad un livello di logica, definito in termini di a_i e b_i. Se si assume il ritardo di una porta logica per ciascun livello di logica in tali equazioni, il caso peggiore è pari a $2+2+1 = 5$ volte il ritardo di una porta.

Quindi adottando questo semplice metodo per la stima della velocità dell'hardware, si ottiene che per la somma su 16 bit il sommatore con carry-lookahead è sei volte più veloce.

Riepilogo

Il punto più importante trattato in questo paragrafo è il fatto che la ALU tradizionale può essere costruita utilizzando un multiplexer e poche altre porte logiche, il tutto ripetuto 32 volte. Per renderla più utile nel quadro dell'architettura MIPS, si è espansa la ALU tradizionale aggiungendo i circuiti logici per verificare se il risultato è 0, per rilevare l'overflow e per eseguire le operazioni base richieste dall'istruzione "set on less than".

Il meccanismo del carry-lookahead offre una strada più veloce rispetto alla propagazione dei riporti attraverso 32 sommatori di 1 bit. Questa strada più veloce è resa praticabile da due segnali, genera e propaga. Il primo genera un riporto indipendentemente dall'esistenza di un riporto in entrata, il secondo propaga in avanti

il riporto. Il carry-lookahead fornisce inoltre un nuovo esempio di quanto l'astrazione sia importante nel progetto dei calcolatori, al fine di poter affrontare la loro complessità.

Approfondimento: Ci si è finora occupati di tutte le operazioni aritmetiche e logiche previste dalle istruzioni base del MIPS, ad eccezione di una: la ALU della figura 4.21 non offre supporto per le istruzioni di scalamento. Sarebbe possibile ampliare il multiplexer della ALU in maniera da includere lo scalamento a sinistra di 1 bit o lo scalamento a destra di 1 bit. Tuttavia i progettisti logici hanno creato un circuito, denominato *barrel shifter*, che può eseguire lo scalamento di una entità da 1 a 32 bit esattamente nel tempo necessario per sommare due numeri su 32 bit, così che lo scalamento è normalmente eseguito al di fuori della ALU.

Approfondimento: L'equazione logica per l'uscita Somma del full adder riportata in precedenza può essere espressa in maniera più semplice utilizzando una porta logica più potente di AND e OR. Una porta EXOR (*OR Esclusivo* o *Exclusive* OR) è vera se i due operandi hanno valori diversi, e quindi

$$x \neq y \Rightarrow 1 \text{ e } x == y \Rightarrow 0$$

In alcune tecnologie l'EXOR è più efficiente rispetto a due livelli di porte AND e OR. Utilizzando il simbolo $\oplus$ per rappresentare l'EXOR, ecco la nuova equazione:

$$\text{Somma} = a \oplus b \oplus \text{CarryIn}$$

Inoltre la ALU è stata progettata nel modo tradizionale, utilizzando le porte logiche. I calcolatori sono oggi progettati in termini di transistor CMOS, che corrispondono sostanzialmente a degli interruttori. Le ALU CMOS e i barrel shifter traggono vantaggio da questi interruttori, e sono composti da un numero minore di multiplexer rispetto a quelli degli schemi qui riportati; le idee di base del progetto restano però le stesse.

4.6 La moltiplicazione

La moltiplicazione è una vessazione,
la divisione è altrettanto terribile;
la regola del tre mi confonde,
e la pratica mi rende pazzo.

Anonimo, Manoscritto elisabettiano, 1570

Dopo aver costruito la ALU ed illustrato somma, sottrazione e scalamenti, si è ora pronti per affrontare la più complessa moltiplicazione.

Innanzitutto però conviene rivedere la moltiplicazione dei numeri in rappresentazione decimale, con l'obiettivo di ricordare i passi ed i nomi degli operandi. Per ragioni che diventeranno chiare tra poco, si limiterà l'esempio all'uso delle sole cifre 0 e 1. Si moltiplichi 1000_{dieci} per 1001_{dieci}:

```
Moltiplicando          1000dieci
Moliplicatore     x     1001dieci
                       1000
                      0000
                     0000
                    1000
Prodotto           1001000dieci
```

Il primo operando è chiamato *moltiplicando* ed il secondo *moltiplicatore*. Il risultato finale è chiamato *prodotto*. Come si ricorderà, l'algoritmo imparato alle elementari consiste nel prendere le cifre del moltiplicatore una alla volta da destra a sinistra, nel moltiplicare il moltiplicando per la singola cifra del moltiplicatore, e nello scalare il prodotto intermedio una cifra alla volta verso sinistra rispetto ai precedenti prodotti intermedi. La prima osservazione riguarda il numero di cifre del prodotto, che è considerevolmente più ampio rispetto sia al moltiplicando che al moltiplicatore. Infatti, se si ignorano i bit di segno, la lunghezza del prodotto di un moltiplicando su n bit e di un moltiplicatore su m bit è un numero lungo $n+m$ bit. Sono quindi necessari $n+m$ bit per rappresentare tutti i possibili prodotti. Di conseguenza come per la somma, anche nella moltiplicazione si deve tenere conto della possibilità che si verifichi un overflow, dal momento che spesso si desidera un prodotto su 32 bit come risultato della moltiplicazione di due numeri su 32 bit.

Nell'esempio si sono limitate le cifre decimali allo 0 e all'1. Con due sole possibilità, ciascun passo della moltiplicazione è semplice:

1. Si mette una copia del moltiplicando (1 × moltiplicando) nella posizione opportuna se la cifra del moltiplicatore è 1, oppure
2. Si mette 0 (0 × moltiplicando) nella posizione opportuna se il moltiplicatore è 0.

Mentre per l'esempio decimale appena visto si è forzata la limitazione all'uso dei soli 0 ed 1, la moltiplicazione di numeri binari utilizza sempre 0 ed 1, e quindi presenta sempre queste due sole possibilità.

Ora che si sono riviste le basi della moltiplicazione, il passo successivo di solito consiste nel presentare i circuiti logici altamente ottimizzati che implementano la moltiplicazione. Qui non si seguirà l'approccio tradizionale, nella convinzione che sia possibile guadagnare in comprensibilità se si analizza l'evoluzione dell'hardware e dell'algoritmo per la moltiplicazione attraverso tre passi. Il resto di questo paragrafo presenta successivi raffinamenti dell'hardware e dell'algoritmo, fino ad arrivare alla versione usata in alcuni calcolatori. Per il momento si assuma di moltiplicare soltanto numeri positivi.

Prima versione dell'algoritmo e del circuito logico che implementano la moltiplicazione

Il progetto iniziale riprende l'algoritmo imparato alle elementari; l'hardware corrispondente è mostrato nella figura 4.25. Il disegno della logica è stato fatto in maniera tale che i dati si muovano dall'alto verso il basso, così da ricalcare più da vicino il metodo carta e penna.

Si assuma che il moltiplicatore si trovi nel registro Moltiplicatore di 32 bit e che il registro Prodotto di 64 bit sia inizializzato a 0. Sulla base dell'esempio fatto con il metodo carta e penna, è chiaro che sarà necessario spostare il moltiplicando a sinistra di una cifra ad ogni passo, in maniera che possa essere sommato ai prodotti intermedi. Dopo 32 passi un moltiplicando su 32 bit si sarà spostato di 32 bit verso sinistra. Ci sarà quindi bisogno di un registro Moltiplicando a 64 bit, inizializzato con il moltiplicando su 32 bit nella metà di destra e con tutti 0 nella metà di sinistra. Il registro è poi scalato a sinistra di una posizione ad ogni passo per allineare il moltiplicando con la somma che si accumula nel registro Prodotto di 64 bit.

La figura 4.26 mostra i tre passi fondamentali necessari pèr il calcolo di ogni bit. Il bit meno significativo del moltiplicatore (Moltiplicatore0) determina se il moltiplicando deve essere sommato al registro Prodotto.

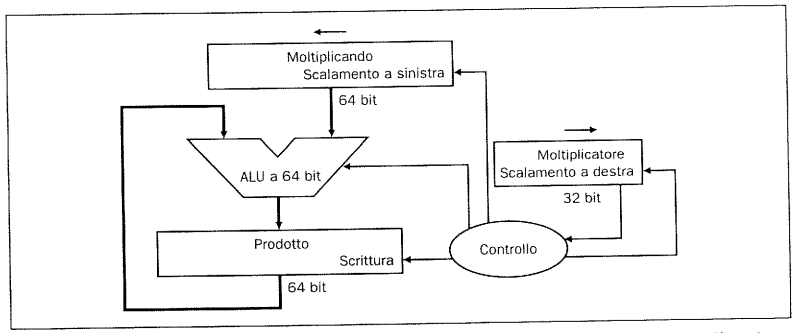


Figura 4.25 Prima versione del circuito logico che realizza la moltiplicazione. Il registro Moltiplicando, la ALU ed il registro Prodotto sono tutti ampi 64 bit e solo il registro Moltiplicatore contiene 32 bit. Il moltiplicando su 32 bit risiede inizialmente nella metà destra del registro Moltiplicando ed è scalato a sinistra di 1 bit ad ogni passo. Il moltiplicatore è scalato nella direzione opposta ad ogni passo. L'algoritmo parte con il prodotto inizializzato a 0. La logica di controllo decide quando scalare i registri Moltiplicando e Moltiplicatore e quando scrivere i nuovi valori nel registro Prodotto.

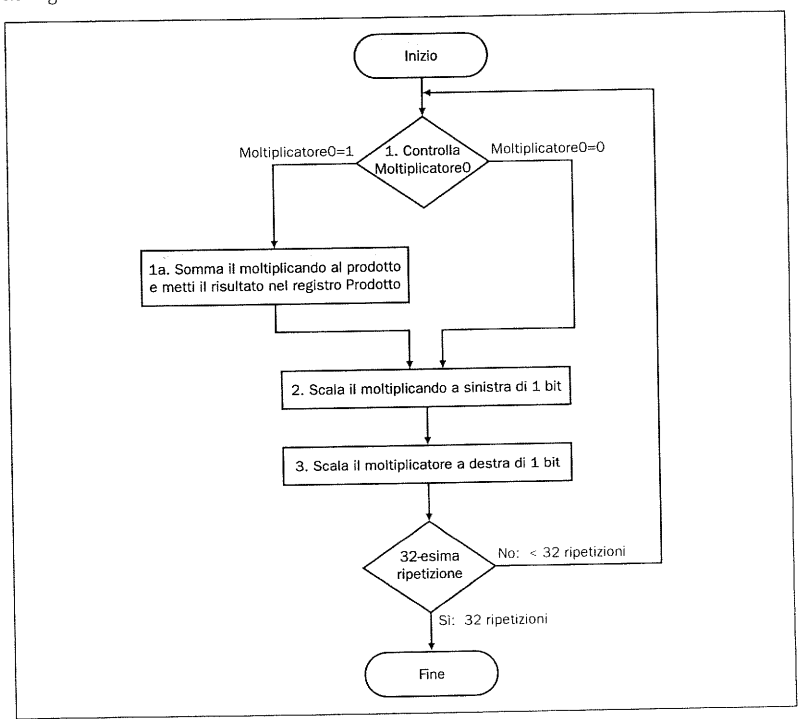


Figura 4.26 Prima versione dell'algoritmo di moltiplicazione, basata sull'hardware della figura 4.25. Se il bit meno significativo del moltiplicatore è 1, si somma il moltiplicando al prodotto. Se no, si procede al passo successivo. Nei due passi successivi si scala il moltiplicando verso sinistra ed il moltiplicatore verso destra. Questi tre passi sono ripetuti 32 volte.

Esempio

Primo algoritmo di moltiplicazione

Utilizzando dei numeri su 4 bit per risparmiare spazio, si moltiplichi $2_{dieci} \times 3_{dieci}$, ossia $0010_{due} \times 0011_{due}$.

Soluzione

La figura 4.27 mostra il valore di ciascun registro per ciascuno dei passi etichettati come indicato dalla figura 4.26, sino al valore finale $0000\ 0110_{due}$, o 6_{dieci}. Il colore contrastante è utilizzato per indicare i valori dei registri che cambiano ad ogni passo, ed il bit evidenziato con un cerchio è quello esaminato per determinare l'operazione eseguita al passo successivo.

Iterazione	Passo	Moltiplicatore	Moltiplicando	Prodotto
0	Valori iniziali	0010	0000 0010	0000 0010
1	1a: 1 ⇒ Prod = Prod + Mcando	0011	0000 0010	0000 0010
	2: Scala a sinistra Moltiplicando	0011	0000 0100	0000 0010
	3: Scala a destra Moltiplicatore	0001	0000 0100	0000 0110
2	1a: 1 ⇒ Prod = Prod + Mcando	0001	0000 1000	0000 0110
	2: Scala a sinistra Moltiplicando	0001	0000 1000	0000 0110
	3: Scala a destra Moltiplicatore	0000	0000 1000	0000 0110
3	1: 0 ⇒ Nessuna operazione	0000	0000 1000	0000 0110
	2: Scala a sinistra Moltiplicando	0000	0001 0000	0000 0110
	3: Scala a destra Moltiplicatore	0000	0001 0000	0000 0110
4	1: 0 ⇒ Nessuna operazione	0000	0001 0000	0000 0110
	2: Scala a sinistra Moltiplicando	0000	0010 0000	0000 0110
	3: Scala a destra Moltiplicatore	0000	0010 0000	0000 0110

Figura 4.27 Esempio di moltiplicazione utilizzando il primo algoritmo (riportato in figura 4.26). Il bit che si esamina per determinare il passo successivo è evidenziato con un cerchio colorato.

Se ogni passo corrispondesse ad un ciclo di clock, allora l'algoritmo richiederebbe quasi 100 colpi di clock per la moltiplicazione. L'importanza relativa delle operazioni aritmetiche come la moltiplicazione varia da programma a programma, ma la somma e la sottrazione sono tra le 5 e le 100 volte più frequenti della moltiplicazione. Conseguentemente, in molte applicazioni la moltiplicazione può richiedere un certo numero di colpi di clock aggiuntivi senza influenzare significativamente le prestazioni. Comunque, la legge di Amdahl (si veda il Capitolo 2) ricorda che anche una frequenza relativamente bassa relativa ad un'operazione lenta può limitare le prestazioni.

Seconda versione dell'algoritmo e del circuito logico che implementano la moltiplicazione

I primi progettisti di calcolatori si resero conto che nel primo algoritmo metà dei bit del moltiplicando rimanevano a 0 e quindi solo metà del registro conteneva bit utili. Una ALU completa a 64 bit sembrava quindi inutile e lenta, dal momento che metà dei bit del sommatore sommavano 0 alla somma parziale.

L'algoritmo originale esegue lo scalamento verso sinistra del moltiplicando inserendo degli 0 nelle nuove posizioni, così che il moltiplicando non può influenzare i

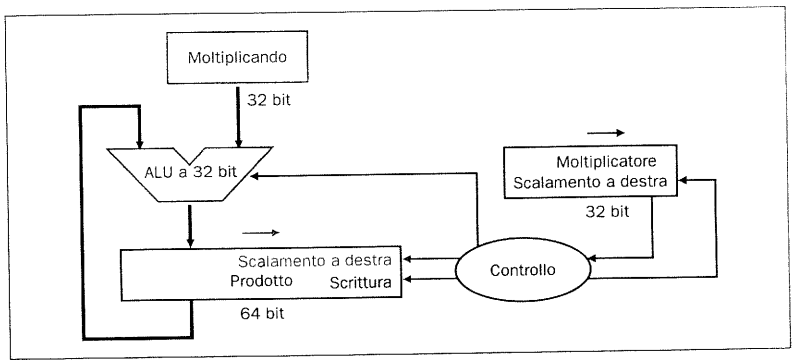


Figura 4.28 Seconda versione dell'hardware per la moltiplicazione. Si confronti con la prima versione della figura 4.25. Il registro Moltiplicando, la ALU ed il registro Moltiplicatore sono tutti ampi 32 bit e solo il registro Prodotto è di 64 bit. Ora il prodotto viene scalato a destra. I cambiamenti sono evidenziati.

bit meno significativi del prodotto dopo che questi sono stati forzati al loro valore. Anziché scalare il moltiplicando a sinistra, si disse, perché non scalare il prodotto a destra? In tal modo il moltiplicando potrebbe restare fisso rispetto al prodotto e dal momento che si stanno sommando 32 bit, il sommatore dev'essere ampio soltanto 32 bit. La figura 4.28 mostra come questi cambiamenti dimezzino l'ampiezza sia della ALU che del moltiplicando.

La figura 4.29 mostra l'algoritmo di moltiplicazione ispirato da queste osservazioni. L'algoritmo parte con il registro Moltiplicando ed il registro Moltiplicatore (ambedue di 32 bit) forzati al loro valore iniziale ed il registro Prodotto di 64 bit forzato a 0. L'algoritmo costruisce una somma su 32 bit, così che solo la metà sinistra del registro Prodotto di 64 bit viene modificata dalla somma.

Esempio

Secondo algoritmo di moltiplicazione

Si moltiplichi $0010_{due} \times 0011_{due}$ utilizzando l'algoritmo della figura 4.29.

Soluzione

La figura 4.30 mostra la versione modificata dell'esempio su numeri di 4 bit; di nuovo il risultato è pari a $0000\ 0110_{due}$.

Versione finale dell'algoritmo di moltiplicazione, con relativo circuito logico

L'ultima osservazione che fecero i primi, attenti progettisti di calcolatori fu che il registro Prodotto sprecava uno spazio che corrispondeva esattamente alla dimensione del moltiplicatore. Mano a mano che lo spazio sprecato nel prodotto scompare, scompaiono anche i bit del moltiplicatore; di conseguenza la terza versione dell'algoritmo di moltiplicazione combina la metà più a destra del prodotto con il moltiplicatore. La figura 4.31 mostra l'hardware. Il bit da controllare è ora il bit meno significativo del registro Prodotto di 64 bit (Prodotto0).

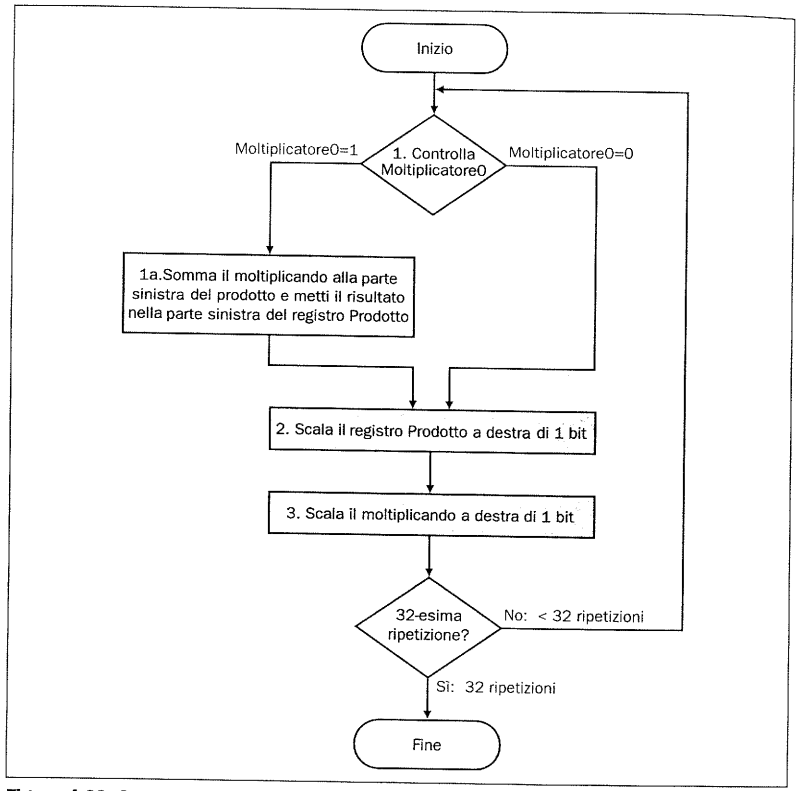


Figura 4.29 Seconda versione dell'algoritmo per la moltiplicazione, basata sul circuito della figura 4.28. In questa versione, anziché scalare il moltiplicando a sinistra, viene fatto scalare a destra il registro Prodotto. I cambiamenti rispetto alla figura 4.26 sono evidenziati.

Iterazione	Passo	Moltiplicatore	Moltiplicando	Prodotto
0	Valori iniziali	0011	0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcando	0011	0010	0010 0000
	2: Scala a destra Prodotto	0011	0010	0001 0000
	3: Scala a destra Moltiplicatore	0001	0010	0001 0000
2	1a: 1 ⇒ Prod = Prod + Mcando	0001	0010	0011 0000
	2: Scala a destra Prodotto	0001	0010	0001 1000
	3: Scala a destra Moltiplicatore	0000	0010	0001 1000
3	1: 0 ⇒ Nessuna operazione	0000	0010	0001 1000
	2: Scala a destra Prodotto	0000	0010	0000 1100
	3: Scala a destra Moltiplicatore	0000	0010	0000 1100
4	1: 0 ⇒ Nessuna operazione	0000	0010	0000 1100
	2: Scala a destra Prodotto	0000	0010	0000 0110
	3: Scala a destra Moltiplicatore	0000	0010	0000 0110

Figura 4.30 Esempio di moltiplicazione utilizzando il secondo algoritmo (riportato in figura 4.29). Il bit che si esamina per determinare il passo successivo è evidenziato con un cerchio colorato.

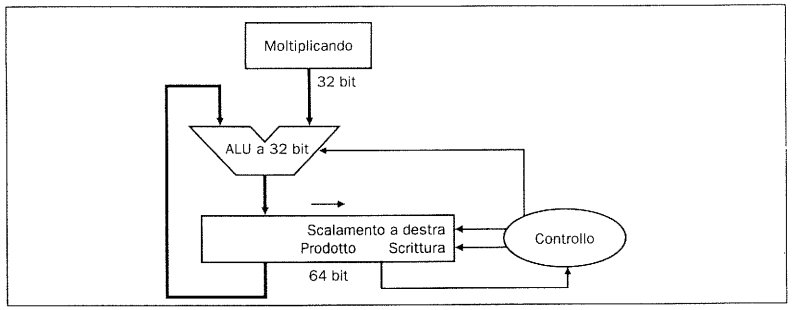


Figura 4.31 Terza versione del circuito logico che realizza la moltiplicazione. Se si confronta con la seconda versione, riportata nella figura 4.28, si nota che il registro Moltiplicatore è scomparso. Il moltiplicatore è invece posto nella metà destra del registro Prodotto.

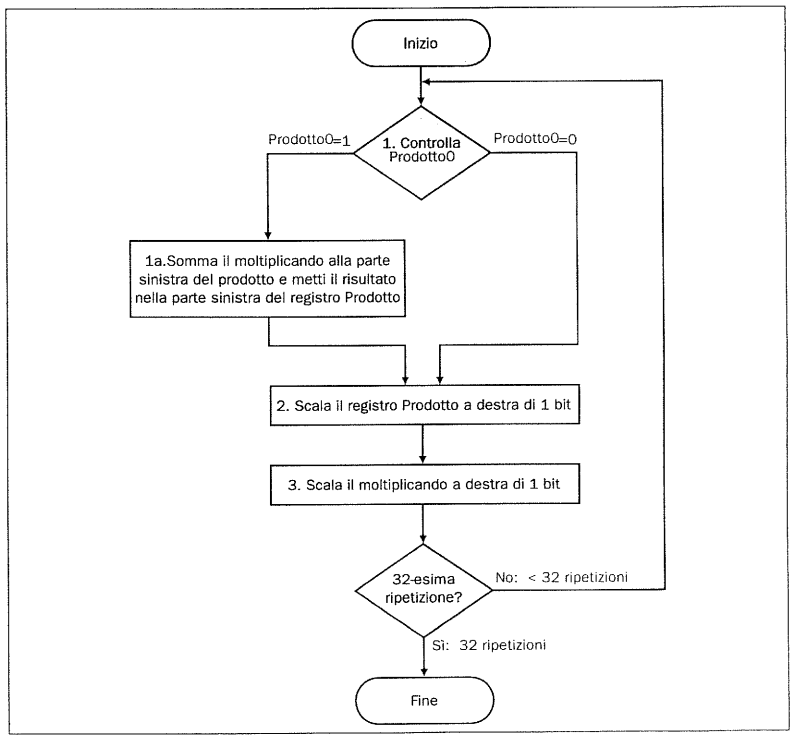


Figura 4.32 Terza versione dell'algoritmo per la moltiplicazione. Sono necessari solo due passi poiché i registri Prodotto e Moltiplicatore sono stati combinati. I cambiamenti rispetto alla figura 4.29 sono evidenziati.

L'algoritmo comincia allocando il moltiplicatore nella metà destra del registro Prodotto e scrivendo 0 nella metà superiore. La figura 4.32 mostra la nuova sequenza di passi.

Terza versione dell'algoritmo di moltiplicazione

Esempio Si moltiplichi $0010_{due} \times 0011_{due}$ utilizzando l'algoritmo della figura 4.32.

Soluzione La figura 4.33 mostra la versione finale dell'esempio su numeri di 4 bit.

Iterazione	Passo	Moltiplicando	Prodotto
0	Valori iniziali	0010	0000 0011
1	1a: 1 ⇒ Prod = Prod + Mcando	0010	0010 0011
	2: Scala a destra Prodotto	0010	0001 0001
2	1a: 1 ⇒ Prod = Prod + Mcando	0010	0011 0001
	2: Scala a destra Prodotto	0010	0001 1000
3	1: 0 ⇒ Nessuna operazione	0010	0001 1000
	2: Scala a destra Prodotto	0010	0000 1100
4	1: 0 ⇒ Nessuna operazione	0010	0000 1100
	2: Scala a destra Prodotto	0010	0000 0110

Figura 4.33 Esempio di moltiplicazione utilizzando il terzo algoritmo (riportato in figura 4.32). Il bit esaminato per determinare il passo successivo è evidenziato con un cerchio colorato.

Moltiplicazione di numeri con segno

Sinora si sono trattati numeri senza segno. Il modo più semplice per capire come gestire dei numeri con segno è quello di convertire prima il moltiplicatore ed il moltiplicando in numeri positivi e poi ricordare i segni originali. L'algoritmo dovrebbe poi lavorare per 31 iterazioni, lasciando i segni fuori dal calcolo. Come viene insegnato nelle scuole elementari, si deve negare il prodotto solo se i segni originali sono discordi.

Ne consegue che l'ultimo algoritmo funzionerà anche per numeri con segno, purché si ricordi che i numeri con cui si ha a che fare hanno un numero infinito di cifre e che si rappresentano solo 32 bit; di conseguenza i passi di scalamento devono estendere il segno del prodotto nel caso dei numeri con segno. Quando l'algoritmo è completo, la parola meno significativa conterrà il prodotto su 32 bit.

L'algoritmo di Booth

Un approccio alla moltiplicazione di numeri con segno più elegante di quello appena proposto è chiamato *algoritmo di Booth*. Questo nasce dall'osservazione che vi sono molti modi di calcolare un prodotto, purché si abbia la possibilità di eseguire somme e sottrazioni. Si supponga di voler moltiplicare 2_{dieci} per 6_{dieci}, ossia 0010_{due} per 0110_{due}:

```
          0010due
X         0110due
+          0000     scalamento (0 nel moltiplicatore)
+         0010      somma      (1 nel moltiplicatore)
+        0010       somma      (1 nel moltiplicatore)
+       0000        scalamento (0 nel moltiplicatore)
        00001100due
```


Booth osservò che una ALU in grado di eseguire somme e sottrazioni poteva ottenere lo stesso risultato in più di una maniera. Per esempio, dal momento che

$$6_{dieci} = -2_{dieci} + 8_{dieci}$$

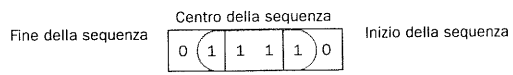
ossia

$$0110_{due} = -0010_{due} + 1000_{due}$$

si può sostituire una stringa di 1 nel moltiplicatore con una operazione iniziale di sottrazione allorquando si incontra il primo 1, e poi successivamente una somma, quando si incontra il primo bit *dopo* l'ultimo 1. Per esempio,

```
          0010due
X         0110due
+          0000     scalamento (0 nel moltiplicatore)
-         0010      sottrazione (primo 1 nel moltiplicatore)
+       0000        scalamento (stringa di 1)
+      0010         somma (il passo precedente
                    ha incontrato l'ultimo 1)
        00001100due
```

Booth inventò questo metodo nell'ambito di uno sforzo per ottenere maggiore velocità, dal momento che negli elaboratori della sua epoca l'operazione di scalamento era più veloce della somma. In effetti, per alcune combinazioni il suo algoritmo era più veloce; per fortuna esso è in grado di lavorare anche con numeri con segno, come verrà dimostrato in seguito. La chiave dell'idea di Booth sta nel classificare le sequenze di 1 in un inizio, un centro ed una fine:



Ovviamente, una stringa di 0 di per sé non necessita di operazioni aritmetiche, per cui può essere ignorata.

Se ci si deve limitare ad osservare le coppie di bit, si possono cercare le corrispondenze rispetto alla situazione del disegno precedente, sulla base del valore di tali bit:

Bit corrente	Bit a destra	Spiegazione	Esempio
1	0	Inizio di una sequenza di 1	00001111000_{due}
1	1	Centro di una sequenza di 1	00001111000_{due}
0	1	Fine di una sequenza di 1	00001111000_{due}
0	0	Centro di una sequenza di 0	00001111000_{due}

L'algoritmo di Booth modifica il primo passo dell'algoritmo nella figura 4.32, che corrispondeva ad osservare un bit del moltiplicatore per poi decidere se sommare il moltiplicando, e lo fa ora corrispondere all'osservazione di due bit del moltiplicatore. Il nuovo passo quindi ha quattro casi, a seconda del valore dei 2 bit. Si assuma che la coppia di bit esaminata consista nel bit corrente e nel bit alla sua destra, ossia quello che era il bit corrente al passo precedente. Il secondo passo continua ad essere lo scalamento a destra del prodotto. Il nuovo algoritmo è quindi il seguente:

1. A seconda del valore del bit corrente e di quello precedente, esegui una delle seguenti operazioni:

Iterazione	Moltiplicando	Algoritmo originale		Algoritmo di Booth	
		Passo	Prodotto	Passo	Prodotto
0	0010	Valori iniziali	0000 0110	Valori iniziali	0000 0110 0
1	0010	1: 0 ⇒ Nessuna operazione	0000 0110	1a: 00 ⇒ Nessuna operazione	0000 0110 0
	0010	2: Scala a destra Prodotto	0000 0011	2: Scala a destra Prodotto	0000 0011 0
2	0010	1a: 1 ⇒ Prod = Prod + Mcando	0010 0011	1c: 10 ⇒ Prod = Prod-Mcando	1110 0011 0
	0010	2: Scala a destra Prodotto	0001 0001	2: Scala a destra Prodotto	1111 0001 1
3	0010	1a: 0 ⇒ Nessuna operazione	0011 0001	1d: 11 ⇒ Nessuna operazione	1111 0001 1
	0010	2: Scala a destra Prodotto	0001 1000	2: Scala a destra Prodotto	1111 1000 1
4	0010	1: 0 ⇒ Nessuna operazione	0001 1000	1b: 01 ⇒ Prod = Prod-Mcando	0001 1000 1
	0010	2: Scala a destra Prodotto	0000 1100	2: Scala a destra Prodotto	0000 1100 0

Figura 4.34 Confronto tra l'algoritmo della figura 4.32 e l'algoritmo di Booth per numeri positivi. Il (i) bit che si esamina (esaminano) per determinare il passo successivo è evidenziato (sono evidenziati) con un cerchio colorato.

00: Centro di una stringa di 0: nessuna operazione aritmetica.

01: Fine di una stringa di 1: somma il moltiplicando alla metà sinistra del prodotto.

10: Inizio di una stringa di 1: sottrai il moltiplicando dalla metà sinistra del prodotto.

11: Centro di una stringa di 1: nessuna operazione aritmetica.

2. Come nell'algoritmo precedente, scala a destra di 1 bit il registro Prodotto.

Si è ora pronti per iniziare il calcolo riportato in figura 4.34. Si parte nel primo passo con uno 0 nel mitico bit alla destra del bit più a destra. La figura 4.34 confronta i due algoritmi (quello di Booth è a destra). Si noti che l'operazione compiuta dall'algoritmo di Booth è ora identificata dai valori dei 2 bit. A partire dal quarto passo i due algoritmi hanno gli stessi valori nel registro Prodotto.

L'altro vincolo che va rispettato è che lo scalamento a destra deve preservare il segno del risultato intermedio, dal momento che si sta lavorando su numeri con segno.

La soluzione consiste nell'estendere il segno quando il prodotto viene fatto scalare a destra. In questo modo il passo 2 della seconda iterazione trasforma 1110 0011 0_{due} in 11110001 1_{due} anziché in 01110001 1_{due}. Questo tipo di scalamento viene chiamato *scalamento aritmetico a destra*, per distinguerlo dallo scalamento logico a destra.

Esempio

L'algoritmo di Booth

Si sperimenti l'algoritmo di Booth con i numeri negativi: $2_{dieci} \times -3_{dieci} = -6_{dieci}$, ossia $0010_{due} \times 1101_{due} = 1111\ 1010_{due}$.

Soluzione

La figura 4.35 mostra i passi.

Iterazione	Passo	Moltiplicando	Prodotto
0	Valori iniziali	0010	0000 1101 0
1	1c: 10 ⇒ Prod = Prod – Mcando	0010	1110 1101 0
	2: Scala a destra Prodotto	0010	1111 0110 1
2	1b: 01 ⇒ Prod = Prod + Mcando	0010	0001 0110 1
	2: Scala a destra Prodotto	0010	0000 1011 0
3	1c: 10 ⇒ Prod = Prod – Mcando	0010	1110 1011 0
	2: Scala a destra Prodotto	0010	1111 0101 1
4	1d: 11 ⇒ Nessuna operazione	0010	1111 0101 1
	2: Scala a destra Prodotto	0010	1111 1010 1

Figura 4.35 Esempio di applicazione dell'algoritmo di Booth con numeri negativi. I bit che si esaminano per determinare il passo successivo sono evidenziati con un cerchio colorato.

L'esempio riportato moltiplica un bit alla volta, ma è possibile generalizzare l'algoritmo di Booth in maniera tale che generi bit multipli e sia quindi più veloce (si veda l'Esercizio 4.53).

Interfaccia Hardware Software

La sostituzione delle operazioni aritmetiche con quelle di scalamento può anche verificarsi nel caso delle moltiplicazioni per valori costanti. Alcuni compilatori sostituiscono le moltiplicazioni per piccole costanti con una serie di scalamenti, somme e sottrazioni. Dal momento che uno scalamento a sinistra di un bit produce un numero due volte più grande in base 2, eseguire uno scalamento a sinistra ha lo stesso effetto di una moltiplicazione per una potenza di 2, e quindi quasi tutti i compilatori sostituiscono con uno scalamento a sinistra la moltiplicazione per una potenza di 2.

Moltiplicazione per 2^i tramite scalamento

Esempio

Si moltiplichi 5_{dieci} per 2_{dieci} usando lo scalamento a sinistra di una posizione.

Soluzione

Dato che

$$101_{due} = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)_{dieci} = 4 + 0 + 1_{dieci} = 5_{dieci}$$

se si scala a sinistra di una posizione, si ottiene

$$1010_{due} = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)_{dieci} = 8 + 0 + 2 + 0_{dieci} = 10_{dieci}$$

e

$$5 \times 2^1_{dieci} = 10_{dieci}$$

Di conseguenza l'istruzione MIPS `sll` può essere usata per eseguire moltiplicazioni per potenze di 2.

Ora che si è dimostrato che l'algoritmo di Booth funziona, si può scoprire *perché* funziona anche con gli interi con segno in complemento a due. Sia a il moltiplicatore e b il moltiplicando; si utilizzerà a_i per fare riferimento all'i-esimo bit di a. Esprimendo l'algoritmo di Booth in termini di valori dei bit del moltiplicatore si ottiene la seguente tabella:

a_i	a_{i-1}	Operazione
0	0	Non fare nulla
0	1	Somma b
1	0	Sottrai b
1	1	Non fare nulla

Anziché esprimere l'algoritmo di Booth in forma tabulare, lo si può rappresentare partendo dall'espressione

$$(a_{i-1} - a_i)$$

dove al valore assunto dall'espressione corrispondono le seguenti operazioni:

0: non fare nulla
+1: somma b
−1: sottrai b

Poiché si sa che lo scalamento a sinistra del moltiplicando rispetto al registro Prodotto è equivalente alla moltiplicazione per una potenza di 2, l'algoritmo di Booth può essere espresso come una somma:

$$\begin{array}{ll} & (a_{-1} - a_0) \times b \times 2^0 \\ + & (a_0 - a_1) \times b \times 2^1 \\ + & (a_1 - a_2) \times b \times 2^2 \\ \ldots & \ldots \\ + & (a_{29} - a_{30}) \times b \times 2^{30} \\ + & (a_{30} - a_{31}) \times b \times 2^{31} \end{array}$$

Si può semplificare questa somma notando che

$$-a_i \times 2^i + a_i \times 2^{i+1} = (-a_i + 2a_i) \times 2^i = (2a_i - a_i) \times 2^i = a_i \times 2^i$$

ricordando che $a_{-1}=0$ e raccogliendo b da ogni termine:

$$b \times ((a_{31} \times -2^{31}) + (a_{30} \times 2^{30}) + (a_{29} \times 2^{29}) + \ldots + (a_1 \times 2^1) + (a_0 \times 2^0))$$

La lunga espressione tra parentesi a destra della prima operazione di moltiplicazione è semplicemente la rappresentazione di a in complemento a due. Quindi la somma può essere ulteriormente semplificata e divenire

$$b \times a$$

In conclusione, l'algoritmo di Booth esegue effettivamente la moltiplicazione in complemento a due di a e b.

La moltiplicazione nel MIPS

Il MIPS possiede una coppia di registri di 32 bit, chiamati *Hi* e *Lo*, che possono contenere il prodotto su 64 bit di una moltiplicazione. Per produrre un risultato corretto, con segno o senza segno, il MIPS prevede due istruzioni: multiply (`mult`) e multiply unsigned (`multu`). Per caricare il prodotto intero su 32 bit, il programmatore può usare l'istruzione *move from lo* (`mflo`). L'assemblatore del MIPS genera una pseudo istruzione per la moltiplicazione che specifica tre registri general-purpose, e che genera le istruzioni `mflo` e `mfhi` necessarie per mettere il prodotto nei registri specificati.

Interfaccia Hardware Software

Ambedue le istruzioni MIPS per la moltiplicazione ignorano l'overflow, e quindi è compito del software fare i controlli per verificare se il prodotto è troppo grande per essere contenuto in 32 bit. Per evitare l'overflow, Hi deve contenere 0 per `multu`, oppure deve contenere la replica del segno di Lo per `mult`. L'istruzione *move from hi* (`mfhi`) può essere utilizzata per trasferire il valore di Hi in un registro general-purpose al fine di eseguire il test di overflow.

Riepilogo

La moltiplicazione è ottenuta tramite un semplice circuito logico per eseguire lo scalamento e la somma; il circuito deriva dal metodo carta e penna imparato alle scuole elementari. Anche i compilatori utilizzano le istruzioni di scalamento per implementare operazioni di moltiplicazione per potenze di due. La moltiplicazione di numeri con segno è più complessa, e l'algoritmo di Booth affronta il problema essenzialmente ricorrendo ad una fattorizzazione intelligente della rappresentazione del moltiplicatore come numero in complemento a due.

Approfondimento: La ragione originaria per cui l'algoritmo di Booth era veloce è che i primi elaboratori potevano eseguire le operazioni di scalamento più in fretta di quanto non potessero eseguire le somme: vi era la speranza che la codifica utilizzata potesse aumentare il numero delle operazioni di scalamento. Tuttavia l'algoritmo è sensibile a particolari combinazioni di bit, e può in alcuni casi aumentare il numero di operazioni di somma o sottrazione. Per esempio configurazioni di bit che alternano 0 e 1, chiamate *1 isolati*, fanno sì che l'hardware esegua una somma o sottrazione ad ogni passo. Se si esamina un numero maggiore di bit per poter identificare gli 1 isolati, si riesce a ridurre il numero di somme necessarie nel caso peggiore. Vantaggi ancora maggiori si possono ottenere eseguendo il calcolo di più bit ad ogni passo, come verrà discusso nell'Esercizio 4.53.

Moltiplicazioni ancora più veloci possono essere ottenute essenzialmente fornendo un sommatore a 32 bit per ogni bit del moltiplicatore: un ingresso è costituito dal moltiplicando messo in AND con un bit del moltiplicatore, l'altro è l'uscita di uno dei precedenti sommatori. Quando si somma una così lunga serie di numeri, è conveniente l'uso di un *carry save adder* (si vedano gli Esercizi dal 4.49 al 4.52).

Approfondimento: La sostituzione della moltiplicazione con uno scalamento, come nell'esempio precedente, è un caso di applicazione di una strategia di ottimizzazione generale utilizzata dai compilatori e nota come *riduzione della forza* (*strength reduction*).

4.7 Divisione

Divide et impera.

Versione latina di "Dividi e domina", antica massima politica
citata da Machiavelli, 1532

L'operazione reciproca della moltiplicazione è la divisione, un'operazione ancora meno frequente e più complessa. In più, essa offre la possibilità di eseguire un'operazione matematicamente non valida, la divisione per 0.

Si inizierà con un esempio di divisione basato su numeri decimali per richiamare i nomi degli operandi e l'algoritmo della divisione imparato alle elementari. Per ragioni simili a quelle addotte nel precedente paragrafo, si limiteranno le cifre decimali a 0 e 1. L'esempio prevede la divisione di 1001010_{dieci} per 1000_{dieci}:

```
                    1001dieci          Quoziente
Divisore 1000dieci | 1001010dieci      Dividendo
                     -1000
                       10
                       101
                       1010
                      -1000
                         10dieci       Resto
```

I due operandi (*dividendo* e *divisore*) ed il risultato (*quoziente*) della divisione sono accompagnati da un secondo risultato chiamato *resto*. Un altro modo per esprimere le relazioni tra i vari componenti è il seguente:

$$\text{Dividendo} = \text{Quoziente} \times \text{Divisore} + \text{Resto}$$

dove il resto è più piccolo del divisore. In alcuni casi poco frequenti i programmi usano l'istruzione di divisione soltanto per ottenere il resto, ignorando il quoziente.

L'algoritmo di base per la divisione imparato alle elementari cerca di verificare quanto è grande il numero che può essere sottratto, costruendo il quoziente un bit alla volta, ad ogni tentativo. L'esempio proposto (accuratamente scelto) usa soltanto i numeri 0 e 1, così che è facile prevedere quante volte il divisore stia nella porzione di dividendo considerata: sarà sempre 0 volte, oppure 1 volta. I numeri binari contengono soltanto 0 e 1 e quindi anche la divisione binaria è limitata a queste due possibilità, risultando così relativamente semplice.

Si assuma che sia il dividendo che il divisore siano positivi e quindi che il quoziente ed il resto siano non-negativi. Gli operandi della divisione e ambedue i risultati sono valori su 32 bit, e per ora verrà ignorato il segno. Anziché descrivere esplicitamente con disegni ed esempi ciascuno dei tre passi di evoluzione dell'algoritmo, come si è fatto per la moltiplicazione, si risparmierà spazio e si fornirà semplicemente un accenno sui due passi intermedi, mentre si descriverà in dettaglio l'algoritmo finale.

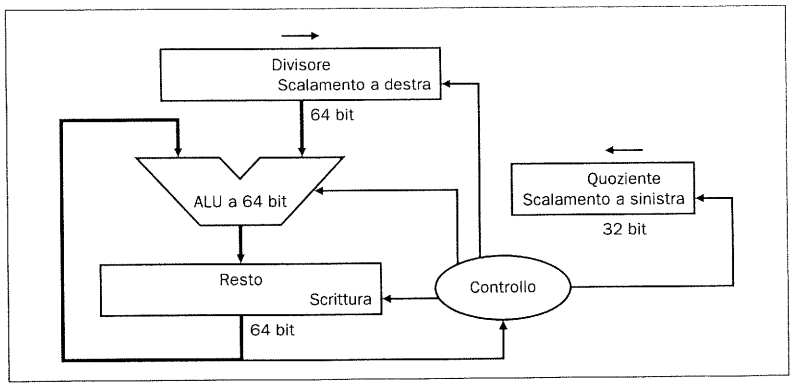


Figura 4.36 Prima versione del circuito logico che implementa la divisione. Il registro Divisore, la ALU, il registro Resto sono tutti lunghi 64 bit, mentre il registro Quoziente è l'unico di 32 bit. Il divisore su 32 bit si trova inzialmente nella metà sinistra del registro Divisore ed è scalato a destra di un bit ad ogni passo. Il resto è inizializzato con il dividendo. La logica di controllo decide quando eseguire lo scalamento dei registri Divisore e Quoziente e quando scrivere il nuovo valore nel registro Resto.

Prima versione dell'algoritmo e della logica di divisione

La figura 4.36 mostra la circuiteria per implementare qualcosa di simile all'algoritmo delle scuole elementari. Si comincia mettendo a 0 il registro Quoziente di 32 bit. Ad ogni passo l'algoritmo deve scalare il divisore a destra di una posizione e quindi si deve iniziare con il divisore messo nella metà sinistra del registro Divisore di 64 bit ed eseguire uno scalamento dello stesso di 1 bit a destra ad ogni passo per allinearlo con il dividendo. Il registro Resto è inizializzato con il valore del dividendo.

La figura 4.37 mostra i tre passi del primo algoritmo di divisione. Diversamente dagli esseri umani, il calcolatore non è abbastanza furbo da capire in anticipo se il divisore è più piccolo del dividendo. Deve prima sottrarre il divisore (passo 1); si ricordi che questo è il modo con cui è eseguito il confronto nell'istruzione "set on less than". Se il risultato è positivo, il divisore è più piccolo o uguale al dividendo, e quindi si genera un 1 nel quoziente (passo 2a). Se il risultato è negativo, il passo successivo consiste nel ripristinare il valore iniziale sommando di nuovo il divisore al resto e generando uno 0 nel quoziente (passo 2b). Il divisore è scalato a destra e si riparte con l'iterazione. Al termine delle iterazioni il resto ed il quoziente si troveranno nei corrispondenti registri.

Primo algoritmo di divisione

Esempio Usando la versione su 4 bit dell'algoritmo per risparmiare spazio, si divida 7_{dieci} per 2_{dieci}, ossia $0000\ 0111_{due}$ per 0010_{due}.

Soluzione La figura 4.38 mostra il valore di ciascun registro per ciascuno dei passi; al termine il quoziente vale 3_{dieci} ed il resto 1_{dieci}. Si noti che il controllo eseguito nel

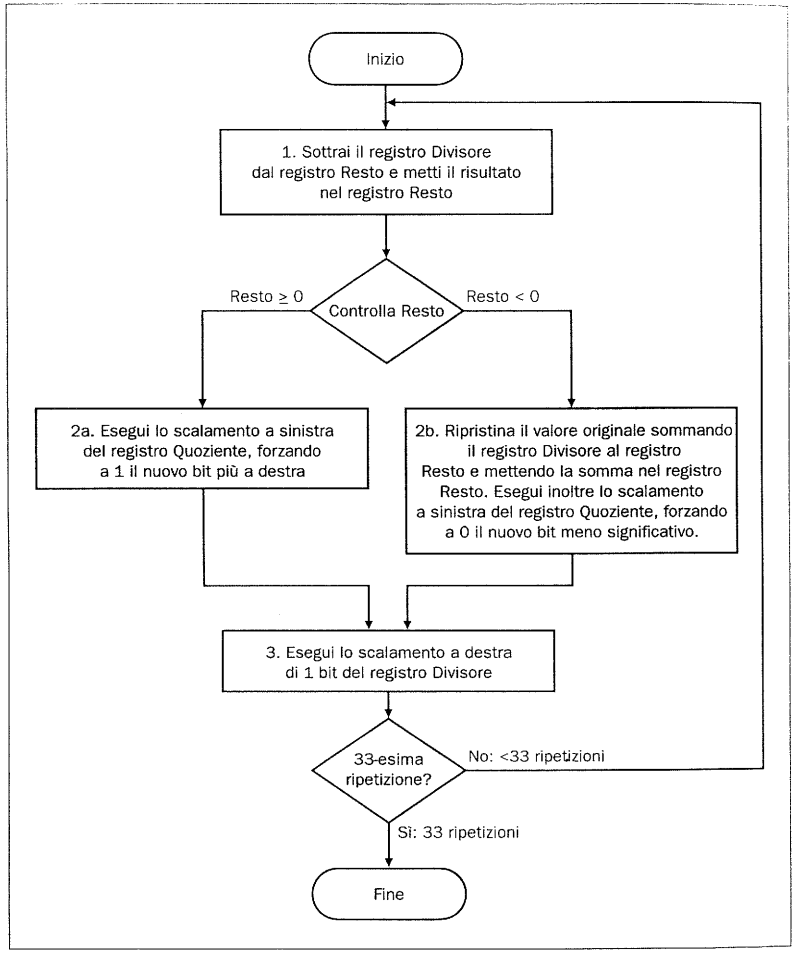


Figura 4.37 Il primo algoritmo di divisione, basato sul circuito logico della figura 4.36. Se il resto è positivo, il divisore è contenuto nel dividendo e il passo 2a genera un 1 nel quoziente. Un resto negativo dopo il passo 1 significa che il divisore non è contenuto nel dividendo, il passo 2b genera uno 0 nel quoziente e somma il divisore al resto, annullando così la sottrazione eseguita nel passo 1. L'operazione finale di scalamento, nel passo 3, allinea opportunamente il divisore rispetto al dividendo per la successiva iterazione. I passi elencati sono ripetuti 33 volte; la ragione dell'apparente passo aggiuntivo diventerà chiara analizzando la successiva versione dell'algoritmo.

passo 2 per verificare se il resto sia positivo o negativo controlla semplicemente se il bit di segno del registro Resto è 0 o 1. La caratteristica sorprendente di questo algoritmo è che richiede $n+1$ passi per produrre il valore corretto di quoziente e resto.

Iterazione	Passo	Quoziente	Divisore	Resto
0	Valori iniziali	0000	0010 0000	0000 0111
1	1: Resto = Resto – Div	0000	0010 0000	1110 0111
	2b: Resto < 0 ⇒ + Div, sll Q, Q 0 = 0	0000	0010 0000	0000 0111
	3: Scala Div a destra	0000	0001 0000	0000 0111
2	1: Resto = Resto – Div	0000	0001 0000	1111 0111
	2b: Resto < 0 ⇒ +Div, sll Q, Q 0 = 0	0000	0001 0000	0000 0111
	3: Scala Div a destra	0000	0000 1000	0000 0111
3	1: Resto = Resto – Div	0000	0000 1000	1111 1111
	2b: Resto < 0 ⇒ +Div, sll Q, Q 0 = 0	0000	0000 1000	0000 0111
	3: Scala Div a destra	0000	0000 0100	0000 0111
4	1: Resto = Resto – Div	0000	0000 0100	0000 0011
	2a: Resto ≥ 0 ⇒ sll Q, Q 0 = 1	0001	0000 0100	0000 0011
	3: Scala Div a destra	0001	0000 0010	0000 0011
5	1: Resto = Resto – Div	0001	0000 0010	0000 0001
	2a: Resto ≥ 0 ⇒ sll Q, Q 0 = 1	0011	0000 0010	0000 0001
	3: Scala Div a destra	0011	0000 0001	0000 0001

Figura 4.38 Esempio di divisione usando il primo algoritmo (riportato in figura 4.37). Il bit che si esamina per determinare il passo successivo è evidenziato con un cerchio colorato.

Seconda versione dell'algoritmo e del circuito logico che implementa la divisione

Ancora una volta, i primi progettisti di calcolatori si accorsero che al più metà del divisore conteneva informazioni utili, e quindi che sia il divisore che la ALU potevano essere divisi a metà. Eseguire lo scalamento a sinistra del resto anziché lo scalamento a destra del divisore produce lo stesso allineamento e raggiunge lo scopo di semplificare l'hardware necessario per la ALU e per il divisore. La figura 4.39 mostra la versione semplificata del circuito logico che implementa la seconda versione dell'algoritmo.

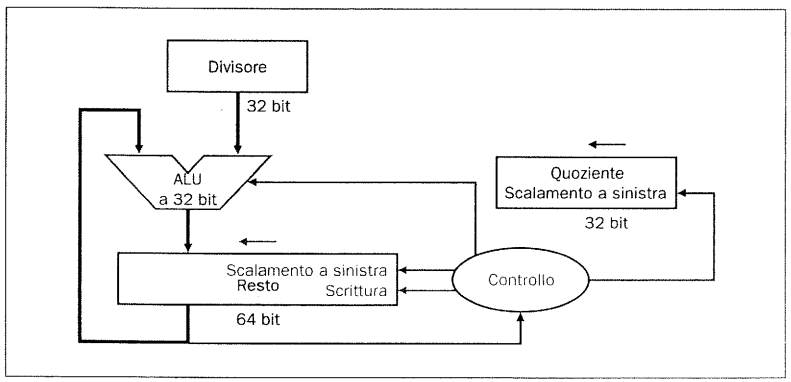


Figura 4.39 Seconda versione del circuito logico che implementa la divisione. Il registro Divisore, la ALU, il registro Quoziente sono tutti lunghi 32 bit, mentre il registro Resto è l'unico rimasto di 64 bit. Se si confronta con la figura 4.36, la ALU ed il registro Divisore sono dimezzati ed il resto è scalato a sinistra. Le modifiche sono evidenziate.

Un'altra modifica deriva dall'osservazione che il primo passo dell'algoritmo corrente non può produrre un 1 nel bit quoziente; se così facesse, allora il quoziente sarebbe troppo grande per poter essere contenuto nel registro. Cambiando l'ordine delle operazioni in maniera da eseguire prima lo scalamento e poi la sottrazione, si può eliminare un'iterazione dell'algoritmo. Quando l'algoritmo termina, il resto si troverà nella metà sinistra del registro Resto.

Versione finale dell'algoritmo e del circuito logico che implementa la divisione

Spinti dallo stesso desiderio di approfondimento e dalle stesse motivazioni che si sono viste per la terza versione dell'algoritmo per la moltiplicazione, i primi progettisti di calcolatori videro che il registro Quoziente poteva essere eliminato eseguendo uno scalamento dei bit del quoziente nel Resto, anziché inserendo degli 0 come nell'algoritmo precedente. La figura 4.40 mostra la terza versione dell'algoritmo.

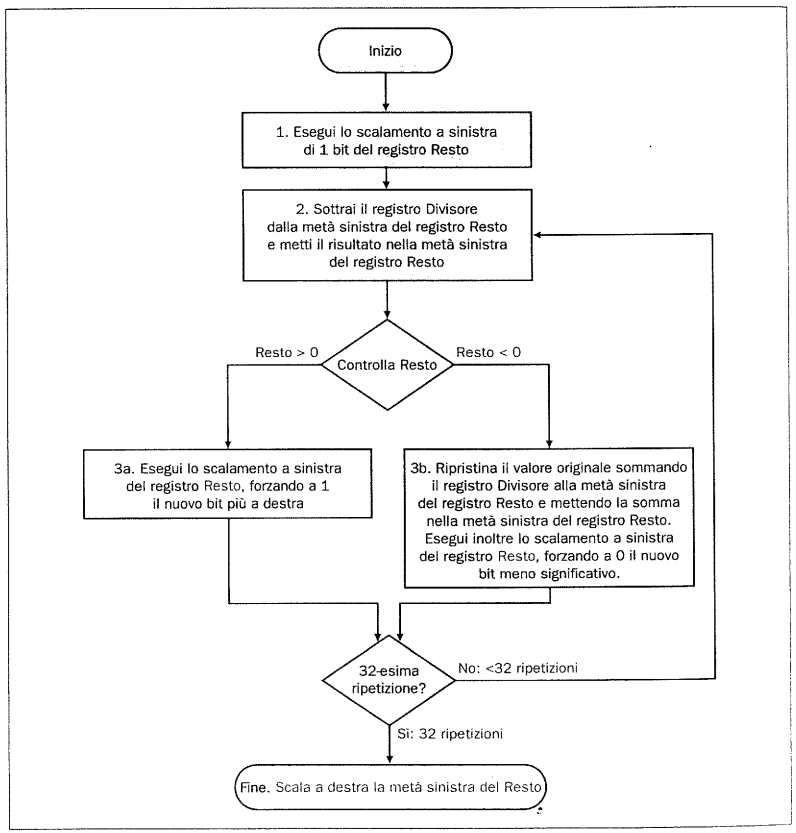


Figura 4.40 Il terzo algoritmo di divisione ha solo due passi. Il registro Resto viene scalato a sinistra.

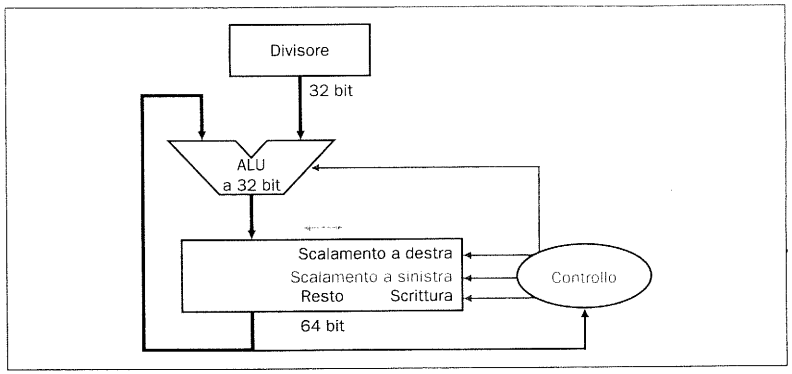


Figura 4.41 Terza versione del circuito logico che implementa la divisione. Questa versione combina il registro Quoziente con la metà destra del registro Resto.

L'algoritmo inizia eseguendo lo scalamento a sinistra del resto, come nel caso precedente. Poi però, il ciclo contiene soltanto due passi, dal momento che lo scalamento del registro Resto ha l'effetto di scalare sia il resto nella metà sinistra che il quoziente nella metà destra (si veda la figura 4.41). La conseguenza della fusione tra i due registri e del nuovo ordine delle operazioni nel ciclo è che il resto viene scalato a sinistra una volta di troppo; è quindi necessario un passo finale di correzione che esegua lo scalamento a ritroso del solo resto nella metà sinistra del registro.

Terzo algoritmo di divisione

Esempio Si usi la terza versione dell'algoritmo per dividere $0000\ 0111_{due}$ per 0010_{due}.

Soluzione La figura 4.42 mostra come venga creato il quoziente nella parte inferiore del registro Resto e come ambedue siano scalati a sinistra con una singola operazione.

Iterazione	Passo	Divisore	Resto
0	Valori iniziali	0010	0000 0111
	Scala a sinistra di un bit Resto	0010	0000 1110
1	2: Resto = Resto – Div	0010	1110 1110
	3b: Resto < 0 ⇒ + Div, sll R, R0 = 0	0010	0001 1100
2	2: Resto = Resto-Div	0010	1111 1100
	3b: Resto < 0 ⇒ + Div, sll R, R0 = 0	0010	0011 1000
3	2: Resto = Resto-Div	0010	0001 1000
	3a: Resto < 0 fi + Div, sll R, R0 = 0	0010	0011 0001
4	2: Resto = Resto – Div	0010	0001 0001
	3a: Resto ≥ 0 fi sll R, R0 = 1	0010	0010 0011
	Scala a destra di 1 bit la metà sinistra di Resto	0010	0001 0011

Figura 4.42 Esempio di divisione usando il terzo algoritmo (riportato in figura 4.40). Il bit che si esamina per determinare il passo successivo è evidenziato con un cerchio colorato.

La divisione con segno

Fino ad ora, parlando della divisione, i numeri con segno sono stati ignorati. La soluzione più semplice consiste nel ricordare i segni del divisore e del dividendo e nel negare il quoziente se i segni sono discordi.

La complicazione è che si deve anche decidere il segno del resto. Si ricordi che deve sempre valere la seguente equazione:

$$\text{Dividendo} = \text{Quoziente} \times \text{Divisore} + \text{Resto}$$

Per capire come decidere il segno del resto, si dia un'occhiata al seguente esempio, in cui si dividono tutte le combinazioni di $\pm 7_{\text{dieci}}$ per $\pm 2_{\text{dieci}}$. Il primo caso è facile:

$$+7 \div +2\text{: Quoziente} = +3, \text{ Resto} = +1$$

Verificando i risultati si ha

$$7 = 3 \times 2 + (+1) = 6 + 1$$

Se si cambia il segno del dividendo, anche quello del quoziente deve cambiare:

$$-7 \div +2\text{: Quoziente} = -3$$

Riscrivendo la formula base per calcolare il resto si ottiene

$$\text{Resto} = (\text{Dividendo} - \text{Quoziente} \times \text{Divisore})$$
$$= -7 - (-3 \times +2) = -7 - (-6) = -1$$

Quindi

$$-7 \div +2\text{: Quoziente} = -3, \text{ Resto} = -1$$

Verificando di nuovo i risultati si ha

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

La ragione per cui il risultato non è un quoziente pari a –4 ed un resto pari a +1, che soddisferebbero comunque la formula, è che il valore assoluto del quoziente cambierebbe a quel punto a seconda del segno del dividendo e del divisore! Chiaramente, se

$$-(x \div y) \neq (-x) \div y$$

la programmazione sarebbe un compito ancora più impegnativo. Questo comportamento anomalo viene evitato seguendo la regola per cui il dividendo ed il resto devono avere lo stesso segno, indipendentemente dal segno del divisore e del quoziente.

Seguendo la stessa regola si possono calcolare le altre combinazioni:

$$+7 \div -2\text{: Quoziente} = -3, \text{ Resto} = +1$$

$$-7 \div -2\text{: Quoziente} = +3, \text{ Resto} = -1$$

In conclusione, un corretto algoritmo di divisione per numeri con segno produce un quoziente negativo se i segni degli operandi sono opposti, e fa sì che il segno del resto, quando questo non è nullo, corrisponda a quello del dividendo.

La divisione nel MIPS

Si sarà forse osservato che lo stesso circuito logico può essere utilizzato sia per la moltiplicazione sia per la divisione. L'unico vincolo è che vi sia un registro di 64 bit in grado di eseguire lo scalamento a sinistra o a destra, oltre ad una ALU di 32 bit che possa eseguire somma e sottrazione. Per esempio, il MIPS usa i registri di 32 bit Hi e Lo sia per la moltiplicazione sia per la divisione. Come ci si può attendere dall'algoritmo presentato, una volta che l'istruzione di divisione è terminata Hi contiene il resto, mentre Lo contiene il quoziente.

Per gestire sia interi con segno che senza segno, il MIPS ha due istruzioni: *divide* (`div`) e *divide unsigned* (`divu`). L'assemblatore MIPS prevede che le istruzioni di divisione specifichino tre registri e genera le istruzioni `mflo` e `mfhi` necessarie per mettere il risultato desiderato in un registro general-purpose.

Interfaccia Hardware Software

Le istruzioni MIPS per la divisione ignorano l'overflow, e quindi è il software che deve determinare se il quoziente è troppo grande. Oltre all'overflow, la divisione può anche ridursi ad un calcolo improprio: la divisione per 0. Alcuni elaboratori distinguono questi due eventi anomali. Il software del MIPS deve invece controllare il divisore per rilevare il verificarsi della divisione per 0, così come avviene per l'overflow.

Riepilogo

I circuiti logici di supporto della moltiplicazione e della divisione permettono al MIPS di fornire una coppia di registri di 32 bit usati sia per la moltiplicazione che per la divisione. La figura 4.43 riassume le aggiunte all'architettura MIPS relative agli ultimi due paragrafi.

Approfondimento: La ragione per cui è necessaria un'iterazione aggiuntiva nel primo algoritmo ed uno scalamento preliminare nel secondo e nel terzo, coinvolge il posizionamento del dividendo nel registro Resto. Ci si aspetta un quoziente su 32 bit ed un divisore su 32 bit, ma in realtà ciascuno di questi è un intero su 31 bit più un bit di segno. Il prodotto sarebbe composto da 31+31 = 62 bit, più un unico bit di segno; la circuiteria può quindi supportare solo un dividendo su 63 bit. Poiché i registri hanno normalmente una dimensione che è una potenza di 2, questo significa che si deve mettere il dividendo su 63 bit nella posizione opportuna all'interno del registro Resto di 64 bit. Se si mettono i 63 bit a destra, bisogna che l'algoritmo lavori per un passo in più per calcolare l'ultimo bit. Una soluzione migliore consiste nell'eseguire uno scalamento preliminare, risparmiando così un passo dell'algoritmo.

Vi è un algoritmo ancora più veloce, che non esegue subito la somma del dividendo a ritroso se il resto è negativo. L'algoritmo si limita a *sommare* il dividendo al resto scalato nel passo successivo, dal momento che $(r+d) \times 2-d = r \times 2+d$. Questo algoritmo di divisione, denominato *senza ripristino* (*nonrestoring*), richiede un solo ciclo di clock per ogni passo, ed è analizzato più in dettaglio nell'Esercizio 4.54; l'algoritmo presentato qui è invece chiamato *con ripristino* (*restoring*).

Operandi MIPS

Nome	Esempio	Commenti
32 registri	$s0-$s7, $t0-$t9, $gp, $fp, $zero, $sp, $ra, $at, Hi, Lo	Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. Il registro $zero ha sempre valore 0. Il registro $at è riservato all'assembler per la gestione di costanti di grandi dimensioni. Hi e Lo contengono il risultato di divisione e moltiplicazione.
2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte e quindi le parole successive differiscono di 4 unità. La memoria mantiene le strutture dati, come i vettori, i registri riversati (spilled) e i registri salvati durante le chiamate a procedura.

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	add $s1, $s2, $s3	$s1 = $s2 + $s3	Tre operandi; dati nei registri
	sottrazione	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Tre operandi; dati nei registri
	somma immediata	addi $s1, $s2, 100	$s1 = $s2 + 100	Usata per sommare costanti
	somma senza segno	addu $s1, $s2, $s3	$s1 = $s2 + $s3	Tre operandi; nessun test sull'overflow
	sottrazione senza segno	subu $s1, $s2, $s3	$s1 = $s2 - $s3	Tre operandi; nessun test sull'overflow
	somma immediata senza segno	addiu $s1, $s2, 100	$s1 = $s2 + 100	Somma una costante; nessun test sull'overflow
	sposta dal registro del coprocessore	mfc0 $s1, $epc	$s1 = $epc	Copia l'Exception PC ed altri registri speciali
	moltiplicazione	mult $s2, $s3	Hi, Lo = $s2 × $s3	Prodotto con segno su 64 bit in Hi e Lo
	moltiplicazione senza segno	multu $s2, $s3	Hi, Lo = $s2 × $s3	Prodotto senza segno su 64 bit in Hi e Lo
	divisione	div $s2, $s3	Hi, Lo = $s2 / $s3	Lo = quoziente, Hi = resto
	divisione senza segno	divu $s2, $s3	Hi, Lo = $s2 / $s3	Quoziente e resto senza segno
	move from Hi	mfhi $s1	$s1 = Hi	Esegue una copia di Hi
	move from Lo	mflo $s1	$s1 = Lo	Esegue una copia di Lo
Logica	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	Tre operandi registri; AND bit a bit
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	Tre operandi registri; OR bit a bit
	and immediato	andi $s1, $s2, 100	$s1 = $s2 & 100	AND bit a bit tra registro e costante
	or immediato	ori $s1, $s2, 100	$s1 = $s2 ! 100	OR bit a bit tra registro e costante
	scalamento logico a sinistra	sll $s1, $s2, 10	$s1 = $s2 << 10	Scalamento a sinistra di una costante
	scalamento logico a destra	srl $s1, $s2, 10	$s1 = $s2 >> 10	Scalamento a destra di una costante
Trasferimento dati	load word	lw $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Word dalla memoria al registro
	store word	sw $s1, 100($s2)	Memoria[$s2 + 100] = $s1	Word dal registro alla memoria
	load byte unsigned	lbu $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Byte dalla memoria al registro
	store byte	sb $s1, 100($s2)	Memoria[$s2 + 100] = $s1	Byte dal registro alla memoria
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Carica una costante nei 16 bit più significativi
Salto condizionato	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Controllo dell'uguaglianza ed eventuale salto; salto PC-relative
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Controllo della disuguaglianza ed eventuale salto; salto PC-relative

(segue)

Figura 4.43 L'architettura del MIPS introdotta sinora. Le porzioni di questa tabella aggiunte rispetto alla figura 4.7 sono evidenziate. Il linguaggio dell'elaboratore MIPS è riassunto nelle ultime pagine del libro.

Categoria	Istruzione	Esempio	Significato	Commenti
Salto condizionato	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0;	Confronto del tipo "minore di"; completamento a due
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0;	Confronto con una costante; completamento a due
	set less than unsigned	sltu $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0;	Confronto del tipo "minore di"; numeri senza segno
	set on less than immediate unsigned	sltiu $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0;	Confronto con una costante; numeri senza segno
Salto incondizionato	jump	j 2500	go to 10000	Salto ad un indirizzo specificato
	jump register	jr $ra	go to $ra	Usato nei costrutti switch e nell'uscita dalle procedure
	jump and link	jal 2500	$ra = PC + 4; go to 10000	Per le chiamate a procedura

Figura 4.43 (continua).

4.8 I numeri in virgola mobile

La velocità non ti porta da nessuna parte se stai andando dalla parte sbagliata.

Proverbio americano

Oltre agli interi con segno e senza segno, i linguaggi di programmazione supportano i numeri aventi una parte frazionaria (ossia con alcune cifre dopo la virgola), quelli cioè che sono chiamati *reali* in linguaggio matematico. Ecco alcuni esempi di numeri reali:

$3{,}14159265\ldots_{\text{dieci}}$ (p)

$2{,}71828\ldots_{\text{dieci}}$ (*e*)

$0{,}000000001_{\text{dieci}}$ o $1{,}0_{\text{dieci}} \times 10^{-9}$ (i secondi in un nanosecondo)

$3155760000_{\text{dieci}}$ o $3{,}15576_{\text{dieci}} \times 10^{9}$ (i secondi in un secolo tipico)

Si noti che nell'ultimo caso il numero non rappresenta una piccola entità, ma è più grande di ciò che è possibile rappresentare tramite un intero con segno su 32 bit. Per gli ultimi due numeri una notazione alternativa è chiamata *notazione scientifica*, che ha una singola cifra a sinistra della virgola. Un numero in notazione scientifica che non abbia uno 0 in posizione iniziale è detto *normalizzato*, ed è questo il modo normale per rappresentare i numeri in notazione scientifica. Per esempio $1{,}0_{\text{dieci}} \times 10^{-9}$ è in notazione scientifica, ma $0{,}1_{\text{dieci}} \times 10^{-8}$ e $10{,}0_{\text{dieci}} \times 10^{-10}$ non lo sono.

Così come si possono rappresentare numeri decimali in notazione scientifica, allo stesso modo si possono rappresentare numeri binari:

$$1{,}0_{\text{due}} \times 2^{-1}$$

Per poter mantenere un numero binario in forma normalizzata, si deve disporre di una base che si possa incrementare o decrementare esattamente dello stesso numero di bit di cui il numero può essere scalato, per giungere ad avere una cifra non nulla a sinistra della virgola. Solo una base pari a 2 soddisfa questo vincolo. Poiché la base non è 10, è anche necessario un nuovo nome per la virgola; *virgola binaria* va bene allo scopo.

L'aritmetica dei calcolatori che supporta tali numeri è detta *in virgola mobile* poiché rappresenta numeri in cui la virgola binaria non è fissa, come è per gli interi. Il linguaggio di programmazione C usa il termine *float* (*mobile*) per tali numeri. Come nella notazione scientifica, i numeri sono rappresentati con una singola cifra non nulla a sinistra della virgola binaria. In binario la forma prevista è

$$1{,}xxxxxxxxxx_{due} \times 2^{yyyy}$$

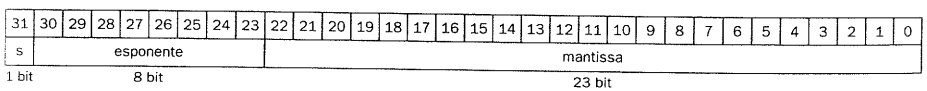
Benché i calcolatori rappresentino l'esponente in base 2, così come il resto del numero, al fine di semplificare la notazione l'esponente verrà rappresentato in decimale.

Una notazione scientifica standard in forma normalizzata per i reali offre tre vantaggi: semplifica lo scambio di dati che includano i numeri in virgola mobile; semplifica gli algoritmi aritmetici per la virgola mobile, in quanto si sa che i numeri sono sempre rappresentati in una certa forma; accresce l'accuratezza dei numeri memorizzabili in una parola, in quanto a destra della virgola binaria gli zeri iniziali non necessari sono sostituiti da cifre reali.

La rappresentazione in virgola mobile

Chi definisce una rappresentazione in virgola mobile deve trovare un compromesso tra la dimensione della mantissa e la dimensione dell'esponente, dal momento che, essendo la dimensione della parola fissa, se si aggiunge un bit all'uno lo si deve togliere dall'altro. Questo compromesso si riflette sull'accuratezza della rappresentazione e sulla dimensione dell'intervallo dei numeri rappresentabili: se si aumenta la dimensione della mantissa si migliora la sua accuratezza, mentre aumentando la dimensione dell'esponente si aumenta l'intervallo di numeri rappresentabili. Come sottolineato dalle linee guida di progetto introdotte nel Capitolo 3, un buon progetto richiede dei buoni compromessi.

I numeri in virgola mobile sono normalmente dei multipli della dimensione della parola. La rappresentazione di un numero in virgola mobile sul MIPS è mostrata di seguito, dove *s* è il segno del numero in virgola mobile (1 significa negativo), *esponente* è il valore del campo esponente di 8 bit (incluso il segno dell'esponente), e *mantissa* è il numero su 23 bit che rappresenta il numero frazionario. Questa rappresentazione è chiamata *modulo e segno*, dal momento che il segno costituisce un bit separato dal resto del numero.



In generale, i numeri in virgola mobile sono nella forma

$$(-1)^s \times F \times 2^E$$

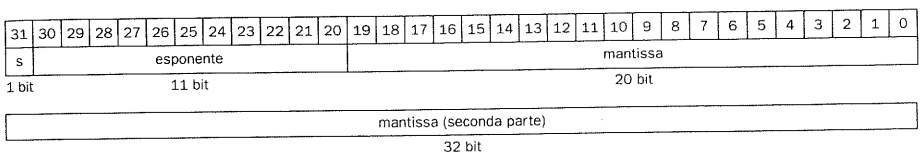
F tiene conto del valore nel campo mantissa ed E di quello nel campo esponente; l'esatta relazione tra questi campi verrà chiarita tra poco.

Le grandezze scelte per l'esponente e la mantissa danno all'aritmetica del MIPS un intervallo di rappresentabilità straordinario. Frazioni piccole come $2{,}0_{dieci} \times 10^{-38}$ e numeri grandi come $2{,}0_{dieci} \times 10^{-38}$ possono essere rappresentati nel calcolatore. Esistono tuttavia numeri molto grandi (anche se diversi da infinito) che non possono essere rappresentati. Di conseguenza interruzioni generate da situazioni di overflow possono verificarsi nell'aritmetica in virgola mobile così come nell'aritmetica intera. Si noti che *overflow* qui significa che l'esponente è troppo grande per poter essere rappresentato nel campo esponente.

La virgola mobile offre anche un nuovo tipo di evento eccezionale. I programmatori in generale vogliono sapere quando hanno calcolato un numero troppo grande per essere rappresentato; allo stesso modo, a volte desiderano sapere se la frazione non nulla che è stata calcolata è divenuta così piccola da non poter essere rappresentata; in ambedue i casi il risultato può essere un programma che fornisce dei risultati scorretti. Questa situazione si verifica quando un esponente negativo è troppo grande per poter stare nel campo esponente. Per distinguerla dall'overflow, questa situazione viene chiamata *underflow*.

Un modo per ridurre le possibilità di underflow o di overflow è quello di usare una notazione dotata di un esponente più grande. In C questa notazione corrisponde al tipo *double* e le operazioni sui double sono chiamate aritmetica in virgola mobile *in doppia precisione*; virgola mobile *in singola precisione* è il nome del formato precedente.

La rappresentazione di un numero in virgola mobile in doppia precisione occupa due parole del MIPS, come mostrato sotto, dove *s* è sempre il segno del numero, *esponente* è il valore del campo esponente su 11 bit, e *mantissa* è il numero su 52 bit corrispondente alla frazione.



La doppia precisione del MIPS permette di rappresentare numeri piccoli quasi come $2{,}0_{dieci} \times 10^{-308}$ e grandi quasi come $2{,}0_{dieci} \times 10^{308}$. Benché la doppia precisione incrementi l'intervallo dell'esponente, il suo principale vantaggio sta nella maggiore accuratezza, a causa della mantissa più grande.

Questi formati hanno una validità più generale del MIPS. Essi fanno parte dello standard *IEEE 754 per la virgola mobile*, che si trova utilizzato praticamente in ogni calcolatore a partire dal 1980. Tale standard ha significativamente aumentato sia la facilità di trasferimento dei programmi in virgola mobile, sia la qualità dell'aritmetica dei calcolatori.

Per far stare ancora più bit all'interno della mantissa, lo IEEE 754 rende implicito il bit a 1 iniziale dei numeri binari normalizzati. Quindi la mantissa è in realtà lunga 24 bit in singola precisione (l'1 sottinteso e la frazione su 23 bit) e 53 bit in doppia precisione (1+52). Poiché lo 0 non ha 1 iniziali, gli si è assegnato il valore riservato corrispondente ad un esponente nullo, così che il circuito non gli inserisce un 1 iniziale.

Quindi $00...00_{due}$ rappresenta il valore 0; la rappresentazione degli altri numeri usa il formato precedente, cui si è aggiunta la particolarità dell'1 sottinteso:

$$(-1)^S \times (1 + \text{Mantissa}) \times 2^E$$

dove i bit della mantissa rappresentano la frazione tra 0 e 1, ed E specifica il valore nel campo esponente, che verrà descritto in dettaglio tra poco. Se si numerano i bit della mantissa da *sinistra a destra* s1, s2, s3, ..., allora il valore è dato da

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + ...) \times 2^E$$

I progettisti dello IEEE 754 vollero inoltre una rappresentazione per la quale potesse essere facilmente implementato il confronto tra interi, in particolare pensando all'ordinamento. Questo desiderio si riflette sulla posizione del segno, che si trova nel bit più significativo, permettendo quindi di eseguire facilmente un test di maggiore, minore, o uguale a 0.

Il posizionamento dell'esponente prima della mantissa semplifica inoltre l'ordinamento dei numeri in virgola mobile utilizzando le istruzioni di confronto tra interi, in quanto i numeri con esponente maggiore sono più grandi dei numeri con esponente minore, purché ambedue gli esponenti abbiano lo stesso segno. In realtà l'operazione è un po' più complicata rispetto ad un semplice ordinamento di interi, poiché questa notazione è essenzialmente del tipo modulo e segno, anziché complemento a due.

Gli esponenti negativi rappresentano un problema per l'ordinamento. Se si usa il complemento a due o una qualunque altra notazione in cui gli esponenti negativi hanno un 1 nel bit più significativo del campo esponente, un esponente negativo apparirà come un numero grande. Per esempio $1{,}0_{due} \times s^{-1}$ sarebbe rappresentato come:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Si ricordi che l'1 iniziale è implicito nella mantissa. Il valore $1{,}0_{due} \times s^{+1}$ apparirà come il numero binario più piccolo:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

La notazione migliore deve quindi rappresentare l'esponente più negativo come $00...00_{due}$ e quello più positivo come $11...11_{due}$. Questa convenzione è chiamata *notazione polarizzata* e la polarizzazione è il numero sottratto alla rappresentazione normale senza segno per determinare il valore reale.

Lo IEEE 754 usa una polarizzazione pari a 127 per la singola precisione, così che –1 è rappresentato dalla combinazione di bit di valore $-1+127_{dieci}$, ossia $126_{dieci} = 0111\ 1110_{due}$, e +1 è rappresentato da 1+127, ossia $128_{dieci} = 1000\ 0000_{due}$. La presenza di un esponente polarizzato implica che il valore rappresentato da un numero in virgola mobile nella realtà è dato da

$$(-1)^S \times (1+\text{Mantissa}) \times 2^{(\text{Esponente}-\text{Polarizzazione})}$$

La polarizzazione dell'esponente per la doppia precisione è pari a 1023.

Quindi il formato IEEE 754 può essere processato dalle istruzioni di confronto tra interi per accelerare l'ordinamento dei numeri in virgola mobile. Si vedrà ora la rappresentazione.

La rappresentazione in virgola mobile

Esempio Si costruisca la rappresentazione binaria IEEE 754 del numero $-0{,}75_{dieci}$ in singola e doppia precisione.

Soluzione Il numero $-0{,}75_{dieci}$ è anche esprimibile come

$$-3/4_{dieci} \text{ o } -3/2^2{}_{dieci}$$

In notazione scientifica il valore è:

$$-0{,}11_{due} \times 2^0$$

ed in notazione scientifica normalizzata:

$$-1{,}1_{due} \times 2^{-1}$$

In generale, la rappresentazione per un numero in singola precisione è data da

$$(-1)^S \times (1+\text{Mantissa}) \times 2^{(\text{Esponente}-127)}$$

e quindi, quando si somma la polarizzazione 127 all'esponente di $-1{,}1_{due} \times 2^{-1}$, il risultato è:

$$(-1)^1 \times (1 + 0{,}1000\ 0000\ 0000\ 0000\ 0000\ 000_{due}) \times 2^{(126-127)}$$

La rappresentazione binaria in singola precisione di $-0{,}75_{dieci}$ è quindi:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bit 23 bit

La rappresentazione in doppia precisione è

$$(-1)^1 \times (1+0{,}1000\ 0000\ 0000\ 000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{due}) \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bit 20 bit

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bit

Si provi ora a procedere nella direzione opposta.

Conversione da rappresentazione binaria a decimale in virgola mobile

Esempio Quale numero decimale è rappresentato dalla seguente parola?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Soluzione Il bit di segno è 1, il campo esponente contiene 129, ed il campo mantissa contiene $1 \times 2^{-2} = 1/4$, ossia 0,25. Usando l'equazione di base

$$\begin{aligned}(-1)^S \times (1+\text{Mantissa}) \times 2^{(\text{Esponente-Polarizzazione})} &= (-1)^1 \times (1+0{,}25) \times 2^{(129-127)} \\ &= -1 \times 1{,}25 \times 2^2 = \\ &= -1{,}25 \times 4 = \\ &= -5{,}0\end{aligned}$$

Nei prossimi paragrafi si descriveranno gli algoritmi per la somma e la moltiplicazione in virgola mobile. Nella loro parte centrale essi usano le corrispondenti operazioni intere sulle mantisse, ma è necessaria una parte di calcolo aggiuntiva per gestire gli esponenti e normalizzare il risultato. Si fornirà innanzitutto una derivazione intuitiva dell'algoritmo in decimale, e poi nelle figure si fornirà una versione binaria più dettagliata.

Approfondimento: Nel tentativo di aumentare l'intervallo di rappresentabilità senza togliere bit alla mantissa, alcuni calcolatori prima dello standard IEEE 754 utilizzavano una base diversa da 2. Per esempio, i mainframe IBM 360 e 370 utilizzavano la base 16. Poiché il cambiamento di uno nell'esponente IBM implica lo scalamento di 4 bit della mantissa, i numeri "normalizzati" con base 16 possono avere sino a 3 bit a 0 iniziali! Quindi l'uso di cifre esadecimali implica che fino a 3 bit devono essere scartati dalla mantissa, il che crea dei notevoli problemi nell'accuratezza dell'aritmetica in virgola mobile, come è descritto nel Paragrafo 4.12.

La somma in virgola mobile

Per illustrare i problemi che si incontrano con la somma in virgola mobile, si consideri la somma a mano di due numeri in notazione scientifica: $9{,}999_{\text{dieci}} \times 10^1 + 1{,}610_{\text{dieci}} \times 10^{-1}$. Si assuma di poter memorizzare soltanto quattro cifre decimali per la mantissa e due cifre decimali per l'esponente.

Passo 1. Per sommare correttamente i due numeri, si deve allineare la virgola del numero che ha l'esponente più piccolo. Quindi si deve trovare una forma per il numero minore, ossia $1{,}610_{\text{dieci}} \times 10^{-1}$, che corrisponda in termini di esponente con il numero di esponente più grande. Si può ottenere tale risultato osservando che esistono diverse rappresentazioni di un numero in virgola mobile non normalizzato in notazione scientifica:

$$1{,}610_{\text{dieci}} \times 10^{-1} = 0{,}1610_{\text{dieci}} \times 10^0 = 0{,}01610_{\text{dieci}} \times 10^1$$

Il numero a destra è la versione desiderata, poiché il suo esponente coincide con quello del numero più grande, $9{,}999_{\text{dieci}} \times 10^1$. Quindi il primo passo esegue lo scalamento a destra della mantissa del numero più piccolo, fino a che il suo esponente non coincide con quello del numero più grande. Si possono però rappresentare solo quattro cifre decimali e quindi dopo lo scalamento il numero effettivo è

$$0{,}016_{\text{dieci}} \times 10^1$$

Passo 2. A questo punto si esegue la somma delle mantisse:

$$\begin{array}{r} 9{,}999_{\text{dieci}} \\ +\ \ 0{,}016_{\text{dieci}} \\ \hline 10{,}015_{\text{dieci}} \end{array}$$

La somma è quindi $10{,}015_{\text{dieci}} \times 10^1$.

Passo 3. La somma ottenuta non è in notazione scientifica normalizzata, per cui è necessario correggerla. Esistono di nuovo diverse rappresentazioni del numero, e la forma normalizzata è

$$10{,}015_{\text{dieci}} \times 10^1 = 1{,}0015_{\text{dieci}} \times 10^2$$

Quindi a valle della somma può essere necessario eseguire un suo scalamento per portarla alla forma normalizzata, con un opportuno aggiustamento dell'esponente. L'esempio mostra uno scalamento a destra, ma se un numero fosse stato positivo e l'altro negativo, allora sarebbe stato possibile che la somma avesse diversi 0 iniziali, e quindi richiedesse un certo numero di operazioni di scalamento a sinistra. Ogni volta che l'esponente è incrementato o decrementato, si deve controllare se si è verificato un overflow o un underflow, cioè ci si deve accertare che l'esponente continui ad essere rappresentabile all'interno del suo campo.

Passo 4. Poiché si è assunto che la mantissa possa essere lunga soltanto quattro cifre (escludendo il segno), è necessario arrotondare il numero. Nell'algoritmo imparato alle scuole elementari si usa una regola per cui si tronca il numero se la cifra a destra del punto desiderato è compresa tra 0 e 4, e si somma 1 alla cifra se il numero a destra è compreso tra 5 e 9. Il numero

$$1{,}0015_{\text{dieci}} \times 10^2$$

è arrotondato a quattro cifre per la mantissa nella forma

$$1{,}002_{\text{dieci}} \times 10^2$$

dal momento che la quarta cifra a destra della virgola era compresa tra 5 e 9. Si noti che esiste una caso sfortunato per l'arrotondamento, consistente nel dover sommare un 1 ad una stringa di 9: la somma può non essere più normalizzata ed occorre eseguire di nuovo il passo 3.

La figura 4.44 mostra l'algoritmo per la somma binaria in virgola mobile che ricalca l'esempio decimale. I passi 1 e 2 sono simili all'esempio appena discusso: si aggiusta la mantissa del numero con esponente minore e si sommano le due mantisse. Il passo 3 normalizza il risultato eseguendo un controllo su overflow ed underflow; il test per l'overflow e l'underflow nel passo 3 dipende dalla precisione degli operandi. Si ricordi che la combinazione di bit tutti a 0 nell'esponente è riservata ed utilizzata per la rappresentazione in virgola mobile dello zero. Inoltre la combinazione di tutti 1 nell'esponente è riservata per indicare valori e situazioni al di fuori dell'insieme dei normali numeri in virgola mobile (si veda l'approfondimento al fondo del Paragrafo 4.8). Ne consegue che per la singola precisione l'esponente più grande è 127 e quello più piccolo –126. I limiti per la doppia precisione sono 1023 e –1022.

Per semplicità, nel passo 4 si adotta il troncamento, che è una delle quattro opzioni di arrotondamento previste dallo standard IEEE 754. L'accuratezza del calcolo in virgola mobile dipende per una buona parte dall'accuratezza dell'arrotondamento e quindi, benché sia facile da realizzare, il troncamento porta lontani dall'accuratezza.

La somma decimale in virgola mobile

Esempio Si provi a sommare i numeri $0{,}5_{dieci}$ e $-0{,}4375_{dieci}$ in binario usando l'algoritmo della figura 4.44.

Soluzione Innanzitutto si dia un'occhiata alla versione binaria dei due numeri in notazione scientifica normalizzata, assumendo di adottare una precisione di 4 bit:

$$\begin{aligned} 0{,}5_{dieci} &= 1/2_{dieci} &&= 1/2^1{}_{dieci} &&= \\ &= 0{,}1_{due} &&= 0{,}1_{due} \times 2^0 &&= 1{,}000_{due} \times 2^{-1} \\ -0{,}4375_{dieci} &= -7/16_{dieci} &&= -7/2^4{}_{dieci} &&= \\ &= -0{,}0111_{due} &&= -0{,}0111_{due} \times 2^0 &&= -1{,}110_{due} \times 2^{-2} \end{aligned}$$

L'algoritmo da seguire è il seguente:

Passo 1: La mantissa del numero con l'esponente minore ($-1{,}11_{due} \times 2^{-2}$) è scalata a destra fino a che il suo esponente non coincide con quello del numero maggiore:

$$-1{,}110_{due} \times 2^{-2} = -0{,}111_{due} \times 2^{-1}$$

Passo 2: Si sommano le mantisse:

$$1{,}0_{due} \times 2^{-1} + (-0{,}111_{due} \times 2^{-1}) = 0{,}001_{due} \times 2^{-1}$$

Passo 3: Si normalizza la somma, controllando se si è verificato overflow o underflow:

$$0{,}001_{due} \times 2^{-1} = 0{,}010_{due} \times 2^{-2} = 0{,}100_{due} \times 2^{-3} = \\ = 1{,}000_{due} \times 2^{-4}$$

Poiché $127 \geq -4 \geq -126$, non c'è né underflow né overflow. L'esponente polarizzato sarebbe $-4+127$, ossia 123, compreso tra 1 e 254, che sono l'esponente più piccolo e quello più grande tra quelli non riservati.

Passo 4: Si arrotonda la somma:

$$1{,}000_{due} \times 2^{-4}$$

La somma è contenuta esattamente in 4 bit, e quindi non ci sono modifiche dovute all'arrotondamento.

La somma è quindi

$$\begin{aligned} 1{,}000_{due} \times 2^{-4} &= 0{,}0001000_{due} &&= 0{,}0001_{due} &&= \\ &= 1/2^4{}_{dieci} &&= 1/16_{dieci} &&= 0{,}0625_{dieci} \end{aligned}$$

Tale valore coincide con ciò che ci si sarebbe aspettati dalla somma di $0{,}5_{dieci}$ e $-0{,}4375_{dieci}$.

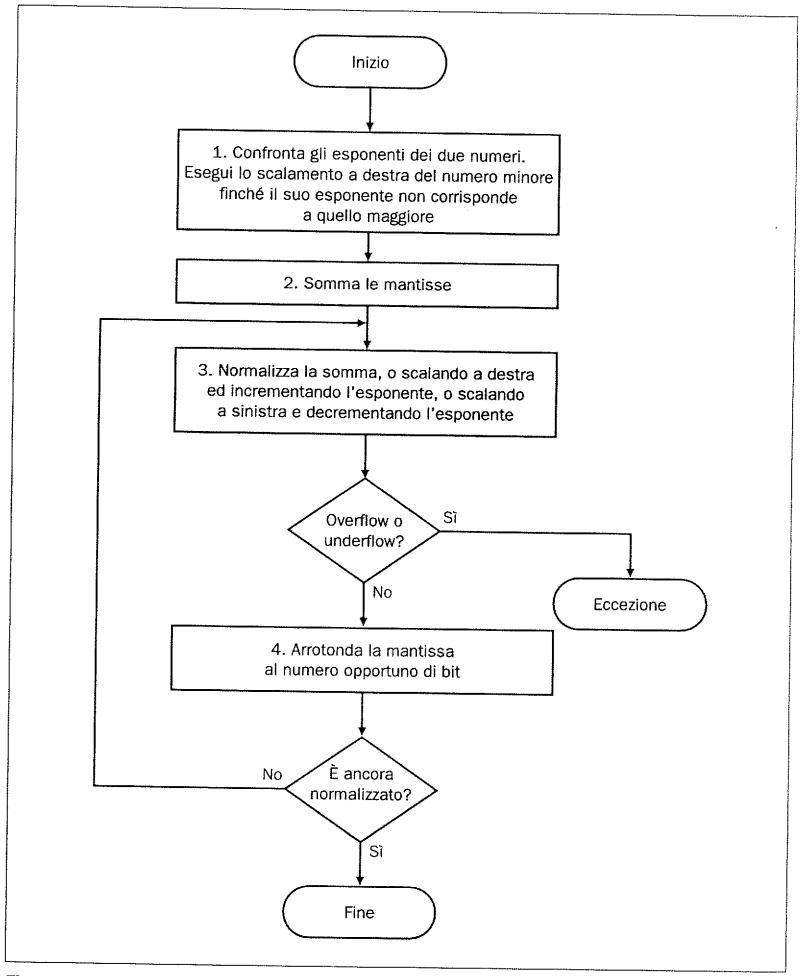


Figura 4.44 Somma in virgola mobile. La sequenza normale corrisponde ad eseguire una volta i passi 3 e 4, ma se l'arrotondamento fa sì che la somma diventi denormalizzata, si deve ripetere il passo 3.

Molti elaboratori possiedono dell'hardware dedicato per l'esecuzione più rapida possibile delle operazioni in virgola mobile. La figura 4.45 delinea l'organizzazione di base della circuiteria per la somma in virgola mobile.

La moltiplicazione in virgola mobile

Ora che è stata illustrata la somma in virgola mobile, si affronterà la moltiplicazione. Si assuma di moltiplicare a mano numeri decimali in notazione scientifica

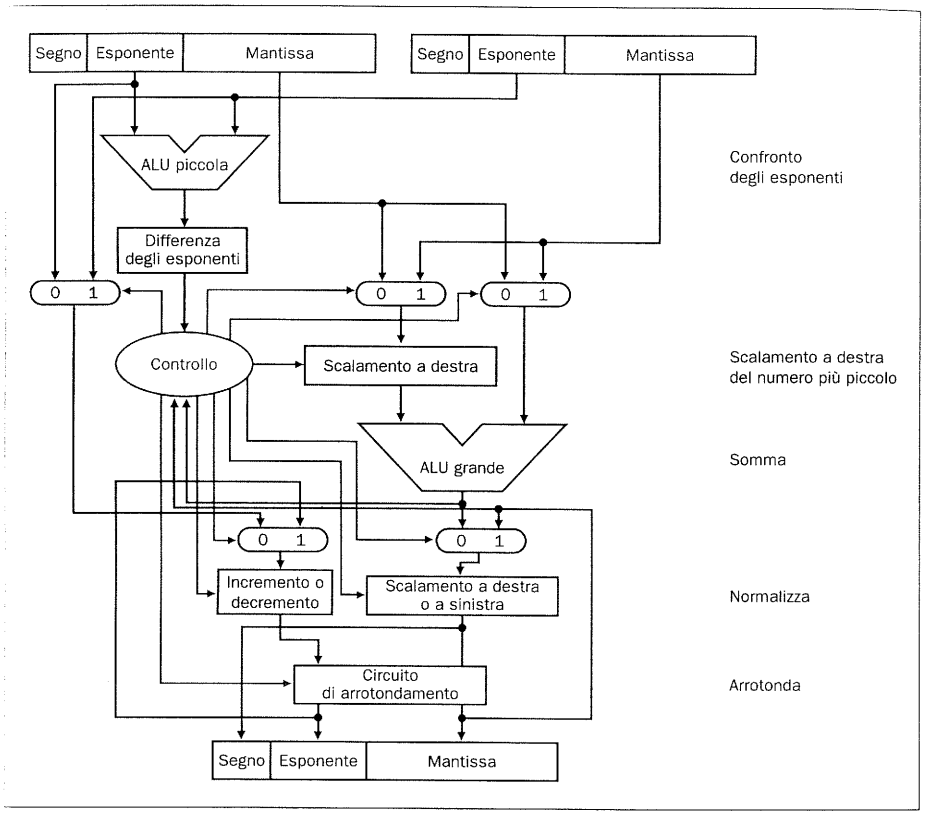


Figura 4.45 Diagramma a blocchi di un'unità aritmetica dedicata alla somma in virgola mobile. I passi della figura 4.44 corrispondono ai vari blocchi, dall'alto verso il basso. Innanzitutto, l'esponente di un operando è sottratto dall'altro utilizzando la ALU piccola per determinare qual è il più grande e di quanto. La differenza controlla i tre multiplexer; da sinistra a destra questi selezionano l'esponente maggiore, la mantissa del numero minore e la mantissa del numero maggiore. La mantissa minore è scalata a destra e poi le mantisse sono sommate insieme utilizzando la ALU grande. Il passo di normalizzazione esegue poi lo scalamento della somma a sinistra o a destra ed incrementa o decrementa l'esponente. L'arrotondamento crea poi il risultato finale, che può richiedere una nuova normalizzazione.

($1{,}110_{dieci} \times 10^{10}$ per $9{,}200_{dieci} \times 10^{-5}$) e di poter memorizzare soltanto quattro cifre della mantissa e due cifre dell'esponente.

Passo 1. Diversamente da quanto si fa per la somma, si calcola l'esponente del prodotto semplicemente sommando gli esponenti degli operandi:

Nuovo esponente = 10 + (–5) = 5

Si ripeta ora l'operazione con gli esponenti polarizzati, in maniera da essere sicuri di ottenere lo stesso risultato: 10+127=137 e -5+127=122, e quindi

Nuovo esponente = 137 + 122 = 259

Il risultato è troppo grande per il campo esponente di 8 bit, e quindi qualcosa non funziona! Il problema nasce dalla polarizzazione, dal momento che insieme con gli esponenti si stanno sommando le polarizzazioni:

Nuovo esponente = (10+127) + (–5+127) = (5+2×127) = 259

Quando si sommano numeri polarizzati, per ottenere il risultato corretto della somma si deve sottrarre la polarizzazione dalla somma:

Nuovo esponente = 137 + 122 – 127 = 259 – 127 = 132 = (5+127)

e 5 è effettivamente l'esponente che si è calcolato inizialmente.

Passo 2. Nel passo successivo si esegue la moltiplicazione delle mantisse:

$$\begin{array}{r} 1{,}110_{dieci} \\ \times\ 9{,}200_{dieci} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{dieci} \end{array}$$

Vi sono tre cifre a destra della virgola per ciascun operando, per cui la virgola è posta sei cifre a destra nella mantissa del prodotto:

$10{,}212000_{dieci}$

Se si assume di poter prendere soltanto tre cifre a destra della virgola, il prodotto è $10{,}2121 \times 10^5$.

Passo 3. Il prodotto non è normalizzato, per cui va corretto. Di nuovo, vi sono molte rappresentazioni per uno stesso numero e si deve scegliere la forma normalizzata:

$10{,}212_{dieci} \times 10^5 = 1{,}0212_{dieci} \times 10^6$

Quindi dopo la moltiplicazione il prodotto può richiedre uno scalamento a destra di una posizione per essere messo nella forma normalizzata, sommando 1 all'esponente. A questo punto si può controllare se si è verificato un overflow o un underflow. L'underflow può verificarsi se ambedue gli operandi sono piccoli, e quindi se ambedue hanno degli esponenti negativi con alto valore assoluto.

Passo 4. Si è fatta l'assunzione che la mantissa sia lunga soltanto 4 bit (ad esclusione del segno), per cui si deve arrotondare il numero. Il numero

$1{,}0212_{dieci} \times 10^6$

è arrotondato a quattro cifre per la mantissa

$1{,}021_{dieci} \times 10^6$

Passo 5. Il segno del prodotto dipende dal segno degli operandi di partenza. Se questi hanno ambedue lo stesso segno, il segno è positivo, altrimenti è negativo. Quindi il prodotto è

$+1{,}0212_{dieci} \times 10^6$

Il segno della somma nell'algoritmo per l'addizione era determinato sommando le mantisse, mentre nella moltiplicazione il segno del prodotto è determinato sulla base del segno degli operandi.

Di nuovo, come mostrato dalla figura 4.46, la moltiplicazione di numeri binari in virgola mobile è abbastanza simile ai passaggi che si sono appena descritti. Si inizia

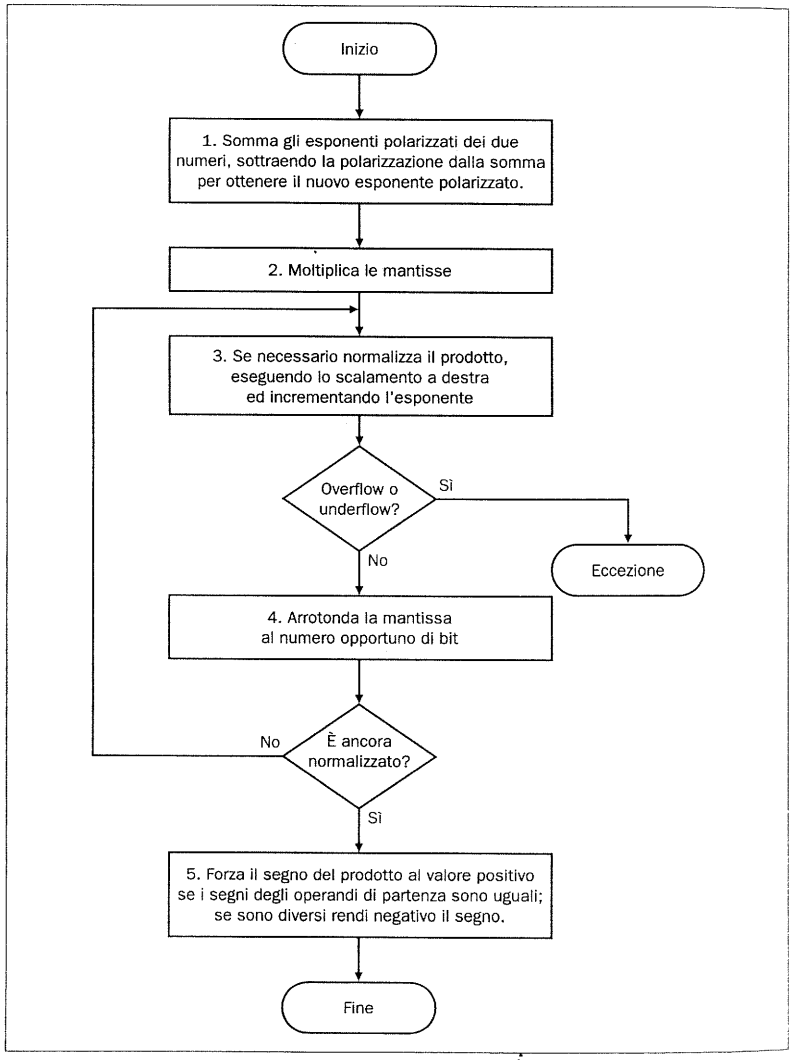


Figura 4.46 Moltiplicazione in virgola mobile. La sequenza normale corrisponde ad eseguire i passi 3 e 4 una volta, ma se l'arrotondamento fa sì che la moltiplicazione diventi denormalizzata, si deve ripetere il passo 3.

con il calcolo del nuovo esponente del prodotto sommando gli esponenti polarizzati, badando a sottrarre una volta la polarizzazione in maniera da ottenere il risultato corretto. Poi si esegue la moltiplicazione delle mantisse, seguita da un eventuale passo di normalizzazione. La dimensione dell'esponente viene controllata per rilevare eventuali situazioni di overflow o underflow, ed il prodotto viene poi arrotondato. Se l'arrotondamento porta ad una ulteriore normalizzazione, si deve di nuovo verificare la dimensione dell'esponente. In conclusione si forza il bit di segno a 1 se i segni degli operandi erano diversi (prodotto negativo), o a 0 se erano uguali (prodotto positivo).

La moltiplicazione decimale in virgola mobile

Esempio Si provi a moltiplicare i numeri $0{,}5_{dieci}$ e $-0{,}4375_{dieci}$ utilizzando i passi della figura 4.46.

Soluzione In binario il compito corrisponde a calcolare $1{,}000_{due} \times 2^{-1}$ per $-1{,}110_{due} \times 2^{-2}$.

Passo 1. Si sommano gli esponenti senza polarizzazione:

$$-1 + (-2) = -3$$

o, utilizzando la rappresentazione polarizzata:

$$(-1+127) + (-2+127) - 127 = (-1-2) + (127+127-127)$$

$$= -3 + 127 = 124$$

Passo 2. Si moltiplicano le mantisse:

$$\begin{array}{r} 1{,}000_{due} \\ \times\ 1{,}110_{due} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{due} \end{array}$$

Il prodotto è $1{,}110000_{due} \times 2^{-3}$, ma bisogna ridurlo a 4 bit, per cui si ottiene $1{,}110_{due} \times 2^{-3}$.

Passo 3. Ora si verifica il prodotto per essere sicuri che sia normalizzato, e poi si controlla l'esponente per rilevare eventuali condizioni di overflow o underflow. Il prodotto è già normalizzato e, poiché $127 \geq -3 \geq -126$, non vi è overflow o underflow. Utilizzando la rappresentazione normalizzata si ha $254 \geq 124 \geq 1$, per cui l'esponente non crea problemi.

Passo 4. L'arrotondamento del prodotto non produce modifiche:

$$1{,}110_{due} \times 2^{-3}$$

Passo 5. Poiché i segni degli operandi iniziali differiscono, il segno del prodotto è negativo. Quindi il prodotto è

$$-1{,}110_{due} \times 2^{-3}$$

Convertendo in decimale per controllare il risultato si ha:

$$-1{,}110_{due} \times 2^{-3} = -0{,}001110_{due} = -0{,}00111_{due}$$
$$= -7/2^5_{dieci} = -7/32_{dieci} = -0{,}21875_{dieci}$$

Il prodotto di $0{,}5_{dieci}$ e $-0{,}4375_{dieci}$ è effettivamente $-0{,}21875_{dieci}$.

Le istruzioni in virgola mobile nel MIPS

Il MIPS supporta i formati IEEE 754 per la singola e la doppia precisione con le seguenti istruzioni:

- Somma in virgola mobile: *addition_single* (`add.s`) e *addition_double* (`add.d`)
- Sottrazione in virgola mobile: *subtraction_single* (`sub.s`) e *subtraction_double* (`sub.d`)
- Moltiplicazione in virgola mobile: *multiplication_single* (`mul.s`) e *multiplication_double* (`mul.d`)
- Divisione in virgola mobile: *division_single* (`div.s`) e *division_double* (`div.d`)
- Confronto in virgola mobile: *comparison_single* (`c.x.s`) e *comparison_double* (`c.x.d`), dove `x` può essere *equal* (`eq`), *not equal* (`neq`), *less than* (`lt`), *less than or equal* (`le`), *greater than* (`gt`), oppure *greater than or equal* (`ge`)
- Salto in virgola mobile: *branch_true* (`bclt`), *branch_false* (`bclf`).

Il confronto in virgola mobile forza a vero o falso un bit, a seconda della condizione di confronto, ed un salto in virgola mobile decide poi se saltare o meno, a seconda della condizione.

I progettisti del MIPS decisero di aggiungere dei registri in virgola mobile, chiamati `$f0`, `$f1`, `$f2`, ... , utilizzati per la singola o la doppia precisione. Di conseguenza essi inclusero operazioni separate di load e store per i registri in virgola mobile: `lwc1` e `swc1`. I registri base per il trasferimento dati in virgola mobile rimangono i registri interi. Il codice MIPS per caricare due numeri in singola precisione dalla memoria, sommarli, e poi memorizzare la somma potrebbe essere qualcosa come

```
lwc1   $f4,x($sp)   # carica il numero v.m. su 32 bit in F4
lwc1   $f6,y($sp)   # carica il numero v.m. su 32 bit in F6
add.s  $f2,$f4,$f6  # F2=F4+F6 in singola precisione
swc1   $f2,z($sp)   # memorizza il numero v.m. su 32 bit da F2
```

Un registro in doppia precisione in realtà corrisponde ad una coppia pari e dispari di registri per la singola precisione, che prende il nome dal registro pari.

Interfaccia Hardware Software

Un punto che deve essere affrontato dai progettisti di calcolatori quando prendono decisioni riguardanti l'aritmetica in virgola mobile è se usare gli stessi registri delle istruzioni intere o se aggiungerne un insieme apposito per la virgola mobile. Poiché i programmi normalmente eseguono le operazioni intere e quelle in virgola mobile su dati diversi, la separazione dei registri incrementa solo leggermente il numero di istruzioni necessarie per l'esecuzione di un programma. L'impatto maggiore è legato alla creazione di un insieme separato di istruzioni di trasferimento dati per spostare i dati tra i registri in virgola mobile e la memoria.

I benefici che derivano dall'avere dei registri separati per la virgola mobile sono la disponibilità di un numero doppio di registri senza dover usare un numero maggiore di bit nel formato delle istruzioni, la disponibilità di una banda doppia in termini di registri, in quanto vi sono insiemi separati di registri per le operazioni intere e per quelle in virgola mobile, e la possibilità di specializzare i registri per la virgola mobile; alcuni elaboratori, ad esempio, convertono gli operandi di tutte le dimensioni esistenti nei registri in un unico formato interno.

La figura 4.47 riassume la parte di architettura MIPS relativa alla virgola mobile introdotta nel Capitolo 4, con le aggiunte evidenziate. Come già si era fatto nella figura 3.18 nel Capitolo 3, la codifica di tali istruzioni è mostrata nella figura 4.48.

Esempio

La compilazione di un programma C in virgola mobile nel codice assembler MIPS

Si converta una data temperatura da gradi Fahrenheit a gradi Celsius:

```
float f2c (float fahr)
    {
        return ((5.0/9.0) * (fahr - 32.0));
    }
```

Si assuma che l'argomento in virgola mobile `fahr` sia passato nel registro `$f12` e che il risultato debba andare in `$f0`. A differenza di quanto avviene per i registri interi, il registro in virgola mobile 0 può contenere un numero. Qual è il corrispondente codice assembler MIPS?

Soluzione

Si assuma che il compilatore metta le tre costanti in virgola mobile in memoria, in maniera facilmente raggiungibile tramite il global pointer register `$gp`. Le prime due istruzioni caricano le costanti 5,0 e 9,0 nei registri in virgola mobile:

```
f2c:
    lwc1   $f16,const5($gp)  # $f16=5,0 (5,0 in memoria)
    lwlc   $f18,const9($gp)  # $f18=9,0 (9,0 in memoria)
```

Le due costanti sono poi divise per ottenere la frazione 5,0/9,0:

```
    div.s $f16, $f16, $f18, # $f16 = 5,0 / 9,0
```

Operandi MIPS per virgola mobile

Nome	Esempio	Commenti
32 registri in virgola mobile	$f0, $f1, $f2, ..., $f31	I registri MIPS per la virgola mobile sono usati in coppia per i numeri in doppia precisione.
2^{30} parole di memoria	Memoria[0], Memoria[4],..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte e quindi le parole successive differiscono di 4 unità. La memoria mantiene le strutture dati, come i vettori, i registri riversati (spilled) e i registri salvati durante le chiamate a procedura.

Linguaggio assembler MIPS per virgola mobile

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	FP add single	add.s $f2, $f4, $f6	$f2=$f4+$f6	Somma in virgola mobile (singola precisione)
	FP subtract single	sub.s $f2, $f4, $f6	$f2=$f4-$f6	Sottrazione in virgola mobile (singola precisione)
	FP multiply single	mul.s $f2, $f4, $f6	$f2=$f4x$f6	Moltiplicazione in virgola mobile (singola precisione)
	FP divide single	div.s $f2, $f4, $f6	$f2=$f4/$f6	Divisione in virgola mobile (singola precisione)
	FP add double	add.d $f2, $f4, $f6	$f2=$f4+$f6	Somma in virgola mobile (doppia precisione)
	FP subtract double	sub.d $f2, $f4, $f6	$f2=$f4-$f6	Sottrazione in virgola mobile (doppia precisione)
	FP multiply double	mul.d $f2, $f4, $f6	$f2=$f4x$f6	Moltiplicazione in virgola mobile (doppia precisione)
	FP divide double	div.d $f2, $f4, $f6	$f2=$f4/$f6	Divisione in virgola mobile (doppia precisione)
Trasferimento dati	load word copr.1	lwc1 $f1, 100($s2)	$f1=Memoria[$s2+100]	Trasferimento dati su 32 bit ad un registro in virgola mobile
	store word copr.1	swc1 $f1, 100($s2)	Memoria[$s2+100]=$f1	Trasferimento dati su 32 bit in memoria
Salti condizionati	Branch on FP true	bc1t 25	se (cond==1) vai a PC+4+100	Salto relativo al PC su condizione vera in virgola mobile
	Branch on Fp false	bc1f 25	se (cond==0) vai a PC+4+100	Salto relativo al PC su condizione falsa in virgola mobile
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2, $f4	se ($f2<$f4) cond=1, altrimenti cond=0	Confronto in virgola mobile singola precisione
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2, $f4	se ($f2<$f4) cond=1, altrimenti cond=0	Confronto in virgola mobile doppia precisione

Linguaggio macchina MIPS per virgola mobile

Nome	Formato	Esempio						Commenti
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2		100		lwc1 $f2,100($s2
swc1	I	57	20	2		100		swc1 $f2,100($s2
bc1t	I	17	8	1		25		bc1t 25
bc1f	I	17	8	0		25		bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Dimensione del campo		6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Tutte le istruzioni MIPS sono lunghe 32 bit

Figura 4.47 L'architettura del MIPS riguardante la virgola mobile introdotta sinora. Per maggiori dettagli si veda l'Appendice A, Paragrafo A.10.

op(31:26):

31–29 \ 28–26	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	Rfmt	Bltz/gez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	andi	ori	xori	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	lh	lwl	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FlPt), (rt(16:16) = 0 => c = f, rt(16:16) = 1 => c = t), rs(25:21):

25–24 \ 32-21	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc1		cfc1		mtc1		ctc1	
1(01)	bc1.*c*							
2(10)	*f* = single	*f* = double						
3(11)								

op(31:26) = 010001 (FlPt), (*f* above: 01000 => *f* = s, 01001 => *f* = d), funct(5:0):

5-3 \ 2-0	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	add.*f*	sub.*f*	mul.*f*	div.*f*		abs.*f*	mov.*f*	neg.*f*
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.*f*	cvt.d.*f*			cvt.w.*f*			
5(101)								
6(110)	c.f.*f*	c.un.*f*	c.eq.*f*	c.ueq.*f*	c.olt.*f*	c.ult.*f*	c.ole.*f*	c.ule.*f*
7(111)	c.sf.*f*	c.ngle.*f*	c.seq.*f*	c.ngl.*f*	c.lt.*f*	c.nge.*f*	c.le.*f*	c.ngt.*f*

Figura 4.48 La codifica delle istruzioni in virgola mobile del MIPS. La notazione fornisce il valore di un campo per riga e per colonna. Per esempio nella parte alta della figura lw corrisponde alla riga numero 4 (100_{due} per i bit 31-29 dell'istruzione) e alla colonna numero 3 (011_{due} per i bit 28-26 dell'istruzione), così che il corrispondente valore del campo op (bit 31-26) è 100011_{due}. La sottolineatura indica che il campo è usato altrove. Per esempio, FlPt nella riga 2 e colonna 1 (op=010001_{due}) è definito nella parte bassa della figura. Quindi sub.f nella riga 0 e colonna 1 della parte in basso significa che il campo funct (bit 5-0 dell'istruzione) vale 000001_{due} e che il campo op (bit 31-26) vale 010001_{due}. Si noti che il campo rs di 5 bit, specificato nella parte centrale della figura, determina se l'operazione è in singola precisione (f=s, per cui rs=10000) o doppia precisione (f=d, per cui rs=100001). Analogamente il bit 16 dell'istruzione determina se l'istruzione bcl.c esegue il controllo sul valore vero (bit 16 = 1 fi bcl.t) o falso (bit 16 = 0 fi bcl.f). La codifica delle istruzioni Rfmt e TLB può essere trovata nella figura 3.18. Le istruzioni in colore sono descritte nei Capitoli 3 e 4, mentre l'Appendice A copre tutte le istruzioni.

Molti compilatori dividerebbero 5,0 per 9,0 al momento della compilazione e salverebbero in memoria la sola costante 5,0/9,0, evitando così la divisione del numero al tempo di esecuzione.
In seguito si carica la costante 32,0 e la si sottrae da fahr ($f12):

```
lwc1  $f18, const32($gp)      # $f18 = 32.0
sub.s $f18, $f12, $f18  # $f18 = fahr - 32.0
```

Infine si moltiplicano i due risultati intermedi, mettendo il prodotto in $f0, che contiene così il risultato di ritorno, e poi si esce dalla procedura:

```
mul.s $f0, $f16, $f18    # $f0 = (5/9)*(fahr-32.0)
jr    $ra                # return
```

Si vedranno ora le operazioni in virgola mobile sulle matrici, che si possono incontrare frequentemente nei programmi scientifici.

La compilazione in assembler MIPS di una procedura C in virgola mobile che utilizza matrici bi-dimensionali

Esempio

La maggior parte dei calcoli in virgola mobile sono eseguiti in doppia precisione. Verrà ora considerata la moltiplicazione di matrici X = Y * Z. Si assuma che X, Y e Z siano tutte matrici quadrate con 32 elementi per dimensione.

```
  void mm (double x[32][32], double y[32][32], double
z[32][32])
  {
        int i, j, k;

        for (i=0; i!=32; i=i+1)
           for (j=0; j!=32; j=j+1)
              for (k=0; k!=32; k=k+1)
                 x[i][j] = x[i][j] + y[i][k]*z[k][j];
  }
```

Gli indirizzi di partenza dei vettori sono dei parametri, e quindi si trovano in $a0, $a1 e $a2. Si assuma che le variabili intere si trovino rispettivamente in $s0, $s1 e $s2. Qual è il codice assembler MIPS corrispondente alla procedura?

Soluzione

Si noti che x[i][j] viene usato nel ciclo più interno. Poiché l'indice del ciclo è k, non influenza x[i][j], e quindi si può evitare di caricare e scaricare x[i][j] ad ogni iterazione. Piuttosto, il compilatore carica x[i][j] in un registro al di fuori del ciclo, accumula la somma dei prodotti di y[i][k] e z[k][j] nello stesso registro, e poi memorizza la somma in x[i][j] al termine del ciclo più interno.

Si può rendere il codice più semplice utilizzando le pseudoistruzioni del linguaggio assembler li (che carica una costante in un registro), e l.d e s.d (che l'assemblatore trasforma in una coppia di istruzioni di trasferimento dati, lwc1 o swc1, verso una coppia di registri in virgola mobile).

Il corpo della procedura inizia salvando il valore 32 (per il controllo di fine ciclo) in un registro temporaneo, e inizializzando poi le tre variabili dei cicli *for*:

```
mm:...
     li $t1, 32 # $t1=32 (dimensione della riga/fine del ciclo)
     li $s0, 0  # i=0; inizializzazione primo ciclo
L1:  li $s1, 0  # j=0; punto di inizio secondo ciclo
L2:  li $s2, 0  # k=0; punto di inizio terzo ciclo
```


Per calcolare l'indirizzo di x[i][j] bisogna sapere come è immagazzinato in memoria un vettore bi-dimensionale 32×32. Come ci si potrebbe aspettare, la sua disposizione è la stessa che avrebbero 32 vettori mono-dimensionali, ciascuno composto da 32 elementi. Quindi la prima fase consiste nel saltare i primi *i* "vettori mono-dimensionali", o righe, per arrivare a quella desiderata. Per fare questo si moltiplica l'indice nella prima dimensione per l'ampiezza della riga, ossia 32. Poiché 32 è una potenza di 2, si può usare uno scalamento:

```
sll   $t2, $s0, 5      # $t2=i*2^5 (ampiezza della riga di x)
```

Poi si somma il secondo indice per selezionare l'elemento j-esimo della riga desiderata:

```
addu  $t2,$t2,$s1      # $t2=i*ampiezza(riga)+j
```

Per trasformare la somma in un indice a byte, lo si moltiplica per la dimensione in byte degli elementi della matrice. Poiché in doppia precisione ogni elemento è costituito di 8 byte, si può sostituire con uno scalamento a sinistra di 3 posizioni:

```
sll   $t2,$t2,3        # $t2=spiazzamento in byte di [i][j]
```

Poi si somma il valore ottenuto all'indirizzo di partenza di x, ottenendo l'indirizzo di x[i][j], e si carica quindi il numero in doppia precisione x[i][j] in $f4:

```
addu  $t2,$a0,$t2      # $t2=indirizzo in byte di x[i][j]
l.d   $f4,0($t2)       # $f4=8 byte di x[i][j]
```

Le cinque istruzioni successive sono virtualmente identiche alle ultime cinque: calcolano l'indirizzo di z[k][j] e caricano il numero in doppia precisione.

```
L3: sll   $t0,$s2,5    # $t0=k*2^5 (ampiezza della riga di z)
    addu  $t0,$t0,$s1  # $t0=k*ampiezza(riga)+j
    sll   $t0,$t0,3    # $t0=spiazzamento in byte di [k][j]
    addu  $t0,$a2,$t0  # $t0=indirizzo in byte di z[k][j]
    l.d   $f16,0($t0)  # $f16=8 byte di z[k][j]
```

Analogamente le cinque istruzioni che seguono sono simili alle ultime cinque: calcolano l'indirizzo di y[i][k] e caricano il numero in doppia precisione.

```
    sll   $t2,$s0,5    # $t0=k*2^5 (ampiezza della riga di y)
    addu  $t0,$t0,$s2  # $t0=i*ampiezza(riga)+k
    sll   $t0,$t0,3    # $t0=spiazzamento in byte di [i][k]
    addu  $t0,$a1,$t0  # $t0=indirizzo in byte di y[i][k]
    l.d   $f18,0($t0)  # $f18=8 byte di y[i][k]
```

Ora che tutti i dati sono stati caricati, si può finalmente eseguire qualche operazione in virgola mobile! Si moltiplicano gli elementi di y e z posizionati nei registri $f18 e $f16, e si accumula la somma in $f4.

```
mul.d  $f16,$f18,$f16  # $f16=y[i][k]*z[k][j]
add.d  $f4,$f4,$f16    # $f4=x[i][j]+y[i][k]*z[k][j]
```

Il blocco finale incrementa l'indice k e rimane nel ciclo sino a che l'indice non raggiunge il valore 32. Se è 32 e si è quindi giunti alla fine del ciclo più interno, bisogna memorizzare la somma accumulata in $f4 in x[i][j].

```
    addiu $s1,$s1,1    # k=k+1
    bne   $s2,$t1,L3   # se (k!=32) vai a L3
    s.d   $f4,0($t2)   # x[i][j]=$f4
```

Analogamente le quattro istruzioni che seguono incrementano la variabile indice dei cicli intermedio ed esterno, eseguendo una nuova iterazione se l'indice non vale 32, ed uscendo se l'indice vale 32.

```
addiu $s1,$s1,1     # j=j+1
bne   $s1,$t1,L2    # se (j!=32) vai a L2
addiu $s0,$s0,1     # i=i+1
bne   $s0,$t1,L1    # se (i!=32) vai a L1
...
```

Approfondimento: La disposizione del vettore discussa nell'esempio, chiamata *per righe*, è usata dal C e da molti altri linguaggi di programmazione. Al contrario il Fortran usa la disposizione *per colonne*, nella quale il vettore è memorizzato colonna per colonna.

Solo 16 dei 32 registri in virgola mobile del MIPS possono essere usati per le operazioni in singola precisione: `$f0`, `$f2`, `$f4`,..., `$f30`. Per i calcoli in doppia precisione si utilizzano questi registri a coppie. I registri in virgola mobile dispari sono utilizzati soltanto per caricare e scaricare la parte destra dei numeri in virgola mobile. Una versione successiva del set di istruzioni del MIPS, denominata MIPS II, aggiunse al set di istruzioni supportato dall'hardware le istruzioni `l.d` e `s.d`. Una versione ancora successiva, MIPS IV, aggiunse l'indirizzamento con indice per le operazioni di trasferimento dati in virgola mobile, eliminando la necessità della quarta istruzione nelle sequenze di caricamento composte da cinque istruzioni come quella vista prima.

Un'altra ragione per avere registri separati per numeri interi ed in virgola mobile è che i microprocessori negli anni '80 non potevano usare abbastanza transistor per poter mettere l'unità in virgola mobile sullo stesso chip dell'unità per l'aritmetica intera. Quindi le unità in virgola mobile, compresi i registri, erano disponibili su richiesta su un secondo chip. Questi chip opzionali di accelerazione sono chiamati *coprocessori*, e la loro esistenza spiega l'acronimo per l'istruzione in virgola mobile di caricamento nel MIPS: `lwc1` significa load word to coprocessor 1 (carica una parola dal coprocessore 1), ossia dall'unità in virgola mobile. Il coprocessore 0 ha a che fare con la memoria virtuale, descritta nel Capitolo 7. A partire dai primi anni '90, i microprocessori hanno integrato la virgola mobile (e quasi tutto il resto) sul chip, e quindi il termine "coprocessore" si unisce ad "accumulatore" e "memoria centrale" quale termine antiquato che rende fuori moda chi lo pronuncia.

Approfondimento: Benché vi siano molti modi per modificare l'hardware dedicato alla moltiplicazione in virgola mobile in maniera da renderla più veloce, la divisione in virgola mobile è considerevolmente più facile da rendere più veloce ed accurata. La lentezza delle divisioni nei primi calcolatori condusse all'eliminazione della divisione da molti algoritmi, ma i calcolatori paralleli hanno portato ad una riscoperta degli algoritmi che utilizzano intensamente la divisione, dal momento che su tali architetture questa funziona meglio. Può quindi essere utile avere delle divisioni più veloci.

Una tecnica per realizzare un moltiplicatore veloce è l'*iterazione di Newton*, dove la divisione è ricondotta a trovare uno zero di una funzione per il calcolo del reciproco 1/x, che è poi moltiplicato per l'altro operando. Le tecniche di iterazione non possono essere arrotondate adeguatamente senza calcolare un numero elevato di bit aggiuntivi. Esiste un chip della TI che risolve il problema calcolando un reciproco estremamente preciso, mentre l'IBM ricorre ad una moltiplicazione-somma in hardware per risolvere lo stesso problema (si veda il Paragrafo 4.9).

La tecnica di divisione *SRT* mira invece ad individuare per tentativi, ad ogni passo, alcuni bit del quoziente, utilizzando una tabella di lookup basata sui bit più significativi del dividendo e del resto, ed affidandosi ai passi successivi per correggere i tentativi sbagliati. Un chip della Cyrix usa questa tecnica per generare 16 bit ad ogni passo!

L'accuratezza dell'aritmetica

Diversamente dagli interi, per i quali è possibile rappresentare con esattezza ogni numero compreso tra il valore più piccolo e quello più grande, i numeri in virgola mobile sono normalmente delle approssimazioni di numeri che in realtà non possono essere rappresentati. La ragione di questo sta nel fatto che ad esempio esiste un'infinita varietà di numeri tra 0 ed 1 ma non più di 2^{53} di questi possono essere rappresentati esattamente in doppia precisione; il meglio che si possa fare è trovare la rappresentazione più vicina al numero effettivo. Per questo lo IEEE 754 offre diverse modalità di arrotondamento che permettono al programmatore di scegliere l'approssimazione desiderata.

L'arrotondamento sembra un'operazione abbastanza semplice, ma per eseguire un arrotondamento accurato sono necessari dei circuiti che calcolino i bit aggiuntivi. Negli esempi precedenti, si è rimasti piuttosto vaghi sul numero di bit che una rappresentazione intermedia può occupare, me è chiaro che se ogni risultato intermedio dovesse essere troncato al numero esatto di bit previsto non ci sarebbe alcuna possibilità di arrotondare in modo intelligente. Per questo lo IEEE 754 mantiene sempre 2 bit aggiuntivi a destra durante i calcoli intermedi, chiamati rispettivamente *guardia* e *arrotondamento*. Si vedrà ora un esempio decimale per illustrare l'importanza di questi bit aggiuntivi.

Arrotondamento con cifre di guardia

Esempio Si sommi $2{,}56_{dieci} \times 10^0$ a $2{,}34 \times 10^2$, assumendo di disporre di tre cifre decimali significative. Si arrotondi al numero decimale più vicino con tre cifre decimali significative, prima con cifre di guardia e di arrotondamento e poi senza.

Soluzione Innanzitutto si deve scalare il numero più piccolo a destra per allineare gli esponenti, così che $2{,}56_{dieci} \times 10^0$ diventa $0{,}0256_{dieci} \times 10^2$. Poiché vi sono le cifre di guardia e di arrotondamento, si possono rappresentare le due cifre meno significative allorché si allineano gli esponenti. La cifra di guardia vale 5 e quella di arrotondamento vale 6. La somma è

$$\begin{array}{r} 2{,}3400_{dieci} \\ +\ 0{,}0256_{dieci} \\ \hline 2{,}3656_{dieci} \end{array}$$

La somma è quindi $2{,}3656_{dieci} \times 10^2$. Poiché vi sono due cifre da arrotondare, si vuole che i valori da 0 a 49 siano arrotondati verso il basso, e quelli da 51 a 99 siano arrotondati verso l'alto, così che 50 funge da separatore. Arrotondando la somma alle tre cifre significative si ottiene $2{,}37_{dieci} \times 10^2$.

Ripetendo i calcoli *senza* le cifre di guardia e di arrotondamento si perdono due cifre nell'elaborazione. La nuova somma è quindi

$$\begin{array}{r} 2{,}34_{dieci} \\ +\ 0{,}02_{dieci} \\ \hline 2{,}36_{dieci} \end{array}$$

La soluzione finale è quindi $2{,}36_{dieci} \times 10^2$, diverso di un'unità nell'ultima cifra rispetto alla somma ottenuta prima.

Poiché il caso peggiore per l'arrotondamento si ha quando il numero effettivo è a metà strada tra le due rappresentazioni in virgola mobile, l'accuratezza nella virgola mobile si misura normalmente in termini di numero di bit di errore nei bit meno significativi della mantissa; tale misura è nota come numero di *unità nell'ultima posizione* (*units in the last place*, o *ulp*). Se un numero fosse diverso da quello effettivo per i 2 bit meno significativi, allora si direbbe che è "fuori di 2 ulp". A meno che non sia scatenata una eccezione di overflow, underflow o operazione non consentita, lo standard IEEE 754 garantisce che il calcolatore utilizzi sempre il numero che si trova entro mezza ulp.

Approfondimento: Benché l'esempio di prima richiedesse soltanto un bit aggiuntivo, la moltiplicazione può richiederne due. Un prodotto binario può avere un bit iniziale a 0, per cui il passo di normalizzazione può produrre uno scalamento a sinistra del prodotto di una posizione. Questo fa sì che il bit di guardia sia scalato nel bit meno significativo del prodotto, lasciando al solo bit di arrotondamento il compito di assistere nell'arrondamento accurato del prodotto.

Lo scopo di bit aggiuntivi di arrotondamento è quello di permettere all'elaboratore di ottenere lo stesso risultato che si sarebbe ottenuto se i risultati intermedi fossero stati calcolati con precisione infinita e poi arrotondati. Per questo lo standard prevede un terzo bit in aggiunta a quelli di guardia ed arrotondamento; esso viene forzato ogni volta che vi sono dei bit non-nulli a destra del bit di arrotondamento. Questo bit (detto *sticky bit*) permette al calcolatore di apprezzare la differenza tra $0{,}50...00_{dieci}$ e $0{,}50...01_{dieci}$ allorché si esegue l'arrotondamento. Lo sticky bit può essere messo a 1 ad esempio durante la somma, quando il numero più piccolo viene scalato a destra.

Riepilogo

Il Quadro di Insieme che segue rafforza il concetto di programma immagazzinato in memoria introdotto nel Capitolo 3; il significato dell'informazione non può essere determinato semplicemente dall'analisi dei bit, dal momento che gli stessi bit possono rappresentare una varietà di oggetti. Questo paragrafo mostra che l'aritmetica dei calcolatori è finita e può quindi non coincidere con l'aritmetica naturale. Per esempio, la rappresentazione standard in virgola mobile secondo lo standard IEEE 754

$$(-1)^S \times (1 + \text{Mantissa}) \times 2^{(\text{Esponente - Polarizzazione})}$$

è quasi sempre un'approssimazione del numero reale. I sistemi di elaborazione devono curarsi di minimizzare la distanza tra l'aritmetica del calcolatore e l'aritmetica del mondo reale, ed i programmatori a volte devono essere consci delle implicazioni derivanti da tale approssimazione.

Quadro d'insieme

Le combinazioni di bit non hanno un significato intrinseco. Essi possono rappresentare numeri interi con segno o senza segno, numeri in virgola mobile, istruzioni, e così via. Ciò che è rappresentato dipende dall'istruzione che opera sui bit all'interno della parola.

La differenza principale tra i numeri dei calcolatori ed i numeri del mondo reale sta nel fatto che i numeri dei calcolatori hanno una dimensione limitata, e quindi una precisione limitata; è possibile calcolare dei numeri che sono troppo grandi o troppo piccoli per essere rappresentati in una parola. I programmatori devono ricordarsi di questi limiti e scrivere i loro programmi di conseguenza.

Interfaccia Hardware Software

Nell'ultimo capitolo sono state presentati le classi di memorizzazione del linguaggio C (si veda il paragrafo Interfaccia Hardware-Software nel Paragrafo 3.6). La tabella seguente mostra alcuni dei tipi di dato del C insieme con le istruzioni di trasferimento dati del MIPS e quelle che operano sui tipi di dato considerati nei Capitoli 3 e 4.

Tipo C	Trasferimento dati	Operazioni
int	lw, sw, lui	addu, addiu, subu, mult, div, and, andi, or, ori, slt, slti
unsigned int	lw, sw, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, sltu, sltiu
char	lb, sb, lui	lw, sw, lui addu, addiu, subu, multu, divu, and, andi, or, ori, sltu, sltiu
campo di bit	lw, sw, lui	and, andi, or, ori, sll, srl
float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	lwc1, swc1	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

Approfondimento: Lo standard IEEE 754 per la virgola mobile è pieno di piccoli accorgimenti che aiutano il programmatore nel suo sforzo di mantenere la massima accuratezza. Alcuni di essi verranno presentati ora, ma per saperne di più si deve dare un'occhiata ai riferimenti alla fine del Paragrafo 4.12.

Vi sono quattro modalità di arrotondamento: arrotondamento sempre al valore superiore (verso $+\infty$), arrotondamento sempre al valore inferiore (verso $-\infty$), troncamento, e arrotondamento al valore pari più vicino. L'ultima modalità definisce cosa fare se il numero si trova esattamente a metà. In alcuni casi la somma di 500 lire viene arrotondata per eccesso alle 1000 lire superiori, a beneficio di chi riscuote la somma. Una soluzione più equa consisterebbe nell'arrotondare tale valore per eccesso in metà dei casi e per difetto nell'altra metà. Nel caso del valore a metà lo standard IEEE 754 afferma che se il bit meno significativo esistente è dispari, si deve sommare uno; se è pari, si deve troncare. Questo metodo crea sempre uno 0 nel bit meno significativo, da cui l'origine del nome. Questa modalità è quella più comunemente usata.

Altre caratteristiche dello standard IEEE 754 sono i simboli speciali per rappresentare casi strani. Per esempio nel caso di una divisione per zero il software, anziché scatenare un'interruzione, può forzare il risultato ad una combinazione di bit che rappresenta il valore $+\infty$ o $-\infty$; l'esponente di valore massimo è riservato per questi simboli speciali. Quando il programmatore stampa i risultati, il programma stampa il simbolo infinito. Per coloro che sono esperti di matematica, si ricordi che lo scopo del concetto di infinito è di fornire una chiusura topologica dell'insieme dei reali.

Lo standard IEEE 754 ha anche un simbolo per il risultato di una operazione non consentita, come 0/0 o la sottrazione di infinito da infinito. Questo simbolo è *NaN*, che sta per *NotANumber* (Non-Numero). Lo scopo di NaN è quello di permettere ai programmatori, quando opportuno, di ritardare alcuni controlli e decisioni ad un momento successivo nel programma. Per consentire dei confronti che includano NaN, lo standard prevede le opzioni *in ordine* e *non in ordine* per i confronti. Per questo il set di istruzioni completo del MIPS ha vari tipi di confronto che permettono di gestire i NaN.

Infine nel tentativo di spremere ogni bit di precisione da ciascuna operazione in virgola mobile, lo standard permette ad alcuni numeri di essere rappresentati in forma non normalizzata. Piuttosto che accettare un *buco* tra lo 0 ed il più piccolo numero normalizzato, lo standard IEEE ammette dei numeri *denormalizzati* (anche detti *denorm* o *subnormali*). Essi hanno lo stesso esponente dello zero, ma una mantissa non-nulla. Questo meccanismo permette ad un numero di degradare in significato sino a che non diventa 0, con un fenomeno denominato *underflow graduale*. Per esempio, il numero normalizzato più piccolo in singola precisione è

$$1{,}0000\ 0000\ 0000\ 0000\ 0000\ 000_{due} \times 2^{-126}$$

ma il numero denormalizzato più piccolo in singola precisione è

$$0{,}0000\ 0000\ 0000\ 0000\ 0000\ 001_{due} \times 2^{-126}, \text{ o } 1{,}0_{due} \times 2^{-149}$$

Per la doppia precisione, l'intervallo della denormalizzazione va da $1{,}0 \times 2^{-1022}$ a $1{,}0 \times 2^{-1074}$.

La possibilità di dover eventualmente gestire un operando denormalizzato ha procurato molti mal di testa ai progettisti di circuiti per la virgola mobile, sempre alla ricerca di modi per costruire unità più veloci. Per questo, molti calcolatori causano un'eccezione se un operando è denormalizzato, lasciando al software il compito di completare l'operazione. Benché le implementazioni software siano perfettamente accettabili, le loro minori prestazioni hanno diminuito la diffusione dei numeri denormalizzati nel software in virgola mobile portabile. D'altro canto, se i programmatori non tengono conto dei numeri denormalizzati, i loro programmi possono comportarsi in maniera inaccettabile.

Di seguito vengono riportate le codifiche dei numeri in virgola mobile previste dallo standard IEEE 754; il segno viene determinato dal bit di segno:

Singola precisione		Doppia precisione		Oggetto rappresentato
Esponente	Mantissa	Esponente	Mantissa	
0	0	0	0	0
0	non-zero	0	non-zero	±numero denormalizzato
1-254	qualsiasi	1-2046	qualsiasi	±numero in virgola mobile
255	0	2047	0	±infinito
255	non-zero	2047	non-zero	NaN (Not a Number)

4.9 Un caso reale: la virgola mobile nel PowerPC e nell'80x86

Sia il PowerPC che l'80x86 possiedono delle normali istruzioni per la moltiplicazione e la divisione che lavorano interamente su registri, diversamente dall'uso di Hi e Lo nel MIPS. In realtà le ultime versioni del MIPS hanno aggiunto delle istruzioni simili.

Le principali differenze possono essere individuate nelle istruzioni in virgola mobile. Il PowerPC è simile al MIPS ad eccezione di una nuova istruzione e di un numero doppio di registri: il PowerPC dispone di 32 registri in virgola mobile per la singola precisione, e di 32 registri per la doppia precisione. L'architettura in virgola mobile dell'80x86, d'altro canto, è completamente diversa da quella di qualsiasi altro calcolatore al mondo.

L'istruzione di moltiplicazione-e-somma del PowerPC

La moltiplicazione di matrici presentata nel Paragrafo 4.8 era basata su un'operazione di moltiplicazione ed un'operazione di somma, e questa situazione è tipica di molte operazioni su matrici e vettori. Per questo il PowerPC ha un'istruzione di moltiplicazione-e-somma: una sola istruzione legge tre operandi, ne moltiplica due, somma il terzo al prodotto, e scrive la somma nell'operando risultato. Quindi le due istruzioni in virgola mobile del MIPS nell'esempio della moltiplicazione di matrici sarebbero state sostituite da una sola nel PowerPC. Tale istruzione può far aumentare le prestazioni di picco in virgola mobile.

L'istruzione di moltiplicazione-e-somma esegue inoltre le due operazioni e *poi* arrotonda, diversamente da quanto farebbero delle istruzioni separate di moltiplicazione e somma, che arrotonderebbero dopo ciascuna operazione. Per aumentare l'accuratezza le istruzioni calcolano i bit aggiuntivi relativi ai risultati intermedi. Oltre a consentire potenzialmente una maggiore velocità, la maggiore accuratezza della moltiplicazione-e-somma può anche essere utile per il calcolo della divisione e della radice quadrata, nonché nelle librerie software che eseguono calcoli con precisione superiore ai 64 bit. Infatti l'hardware del PowerPC usa l'hardware della moltiplicazione-e-somma per calcolare la divisione, e l'accuratezza della divisione è stato il motivo fondamentale per eliminare la distinzione tra le due operazioni.

L'architettura in virgola mobile dell'80x86

Il coprocessore in virgola mobile Intel 8087 fu annunciato nel 1980; la sua architettura estese l'8086 con l'aggiunta di circa 60 istruzioni in virgola mobile.

La Intel mise a disposizione un'architettura a stack con le sue istruzioni in virgola mobile: le istruzioni load eseguono l'inserimento dei numeri sulla cima dello stack, le operazioni prendono gli operandi nei due elementi in cima allo stack, e le istruzioni di store estraggono elementi dalla cima dello stack. La Intel aggiunse a questa architettura a stack alcune istruzioni e modi di indirizzamento che permettono all'architettura stessa di fruire di alcuni dei vantaggi di un modello a registri e memoria. Oltre a prendere gli operandi dai due elementi sulla cima dello stack, uno degli operandi può trovarsi in memoria o in uno dei sette registri esistenti sul chip e che si trovano sotto la cima dello stack. Quindi si dispone di un set di istruzioni completo di tipo stack, a cui si aggiunge un set parziale di istruzioni di tipo registro-memoria.

Questa soluzione ibrida corrisponde comunque ancora ad una versione particolare del modello registro-memoria, in quanto le istruzioni load spostano sempre i dati verso la cima dello stack mentre incrementano il puntatore alla cima dello stack, e le istruzioni di store possono soltanto trasferire in memoria ciò che è sulla cima dello stack. La Intel usa la notazione *ST* per indicare la cima dello stack e *ST(i)* per indicare l'*i*-esimo registro sotto la cima dello stack.

Un'altra caratteristica innovativa di questa architettura consiste nel fatto che gli operandi nello stack dei registri possono essere più grandi di quando sono immagazzinati in memoria, e tutte le operazioni sono eseguite con questa precisione interna superiore. Diversamente da MIPS e PowerPC, per i quali vi è un massimo di 64 bit, gli operandi in virgola mobile dell'80x86 all'interno dello stack hanno un'ampiezza di 80 bit. I numeri sono automaticamente convertiti al formato interno su 80 bit all'atto dell'esecuzione dell'istruzione load, e ri-convertiti alla dimensione opportuna all'atto dell'esecuzione della store. Questa precisione doppia ed estesa (*double extended precision*) non è supportata dai linguaggi di programmazione, benché sia stata utilizzata dai programmatori di software matematico.

I dati in memoria possono corrispondere a numeri in virgola mobile su 32 bit (singola precisione) o 64 bit (doppia precisione). La versione registro-memoria delle istruzioni converte dunque gli operandi in memoria al formato Intel su 80 bit prima di eseguire una qualsiasi operazione. Inoltre le istruzioni di trasferimento dati convertono automaticamente gli interi su 16 o 32 bit in numeri in virgola mobile, e viceversa, all'atto di eseguire le istruzioni di load intera e store intera.

Le operazioni in virgola mobile dell'80x86 possono essere suddivise in quattro classi principali:

1. Istruzioni di spostamento dei dati, comprese load, load di costanti e store.
2. Istruzioni aritmetiche, comprese somma, sottrazione, moltiplicazione, divisione, radice quadrata e valore assoluto.
3. Confronti, comprese le istruzioni per l'invio del risultato al processore che tratta gli interi, così che questo possa eseguire il salto.
4. Istruzioni trascendenti, compreso seno, coseno, logaritmo ed esponenziale.

La figura 4.49 mostra alcune delle 60 operazioni in virgola mobile. Si sono utilizzate le parentesi graffe { } per riportare variazioni opzionali delle operazioni base: {I} significa che esiste una versione intera dell'istruzione, {P} significa che la versione esegue una pop dallo stack di un operando al termine dell'operazione, ed {R} indica un'inversione dell'ordine degli operandi nell'operazione.

Non tutte le combinazioni rese possibili dalla notazione sono disponibili. Quindi

```
F{I}SUB{R}{P}
```

rappresenta le seguenti istruzioni dell'80x86:

```
FSUB, FISUB, FSUBR, FISUBR, FSUBP, FSUBRP
```

Per le istruzioni di sottrazione tra interi non esiste la versione con pop (FISUBP) o con pop ed inversione (FISUBRP).

Si noti che si possono ottenere altre combinazioni ancora se si includono i modi degli operandi previsti per queste operazioni. La figura 4.50 mostra le tante opzioni per la somma in virgola mobile, anche ignorando le versioni della stessa istruzione per numeri interi e con pop.

Trasferimento dati	Aritmetiche	Confronti	Trascendentali
F{I}LD mem/ST(i)	F{I}ADD{P} mem/ST(i)	F{I}COM {R}{P}	FPATAN
F{I}ST{P} mem/ST(i)	F{I}SUB{R}{P} mem/ST(i)	F{I}UCOM {R}{P}	F2XM1
FLDPI	F{I}MUL{P} mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F{I}DIV{R}{P} mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

Figura 4.49 Le istruzioni in virgola mobile dell'80x86. La prima colonna mostra le istruzioni di trasferimento dati, che spostano dati verso la memoria o verso uno dei registri sotto alla cima dello stack. Le ultime tre operazioni nella prima colonna eseguono la push nello stack di costanti quali π, 1,0 e 0,0. La seconda colonna contiene le operazioni aritmetiche descritte precedentemente. Si noti che le ultime tre operano solo sulla cima dello stack. La terza colonna è relativa alle istruzioni di confronto. Poiché non vi sono istruzioni speciali di salto in virgola mobile, il risultato del confronto deve essere trasferito alla CPU intera attraverso l'istruzione FSTSW, scrivendolo o nel registro AX o in memoria, seguita da un'istruzione SAHF per forzare i codici di condizione. Il confronto in virgola mobile può poi essere controllato utilizzando le istruzioni di salto intere. La colonna finale fornisce le operazioni in virgola mobile di livello superiore.

Istruzione	Operandi	Note
FADD		Ambedue gli operandi nello stack; il risultato sostituisce la cima dello stack.
FADD	ST(i)	Un operando sorgente è il registro i-esimo sotto la cima dello stack; il risultato sostituisce la cima dello stack.
FADD	ST(i), ST	Un operando sorgente è la cima dello stack; il risultato sostituisce la cima dello stack.
FADD	mem32	Un operando sorgente è una locazione di memoria di 32 bit; il risultato sostituisce la cima dello stack.
FADD	mem64	Un operando sorgente è una locazione di memoria di 64 bit; il risultato sostituisce la cima dello stack.

Figura 4.50 I vari tipi di operandi per la somma in virgola mobile nell'80x86.

Le istruzioni in virgola mobile sono codificate utilizzando il codice ESC dell'8086 e lo specificatore di indirizzo postbyte (si veda la figura 3.35). Le operazioni in memoria riservano 2 bit per decidere se l'operando è un numero in virgola mobile su 32 o 64 bit, oppure un intero su 16 o 32 bit. Questi stessi due bit sono usati nelle versioni che non accedono alla memoria per decidere se lo stack debba essere sottoposto ad una pop dopo l'operazione e se la cima dello stack o un registro al di sotto debba ricevere il risultato.

Le prestazioni in virgola mobile della famiglia 80x86 sono rimaste tradizionalmente molto inferiori a quelle degli altri calcolatori. È difficile dire se si tratta semplicemente di una mancanza di attenzione da parte dei progettisti Intel, un disinteresse da parte dei clienti di PC, se il problema risieda nell'architettura, o se più probabilmente si tratti di una combinazione di tutti questi fattori. Si può anche ricordare come molte nuove architetture siano state annunciate a partire dal 1980, ma che nessuna abbia avuto seguito nei piani Intel.

4.10 Errori e trabocchetti

La matematica può quindi essere definita come quel tema a proposito del quale non sappiamo mai di che cosa stiamo parlando, né se ciò che stiamo dicendo sia vero.

Bertrand Russel, *Recent Words on the Principles of Mathematics*, 1901

Gli errori ed i trabocchetti a proposito dell'aritmetica hanno generalmente origine dalla differenza tra la limitatezza della precisione dell'aritmetica dei calcolatori, e la non limitatezza della precisione dell'aritmetica naturale.

Errore: l'addizione in virgola mobile è associativa; in altre parole, $x+(y+z)=(x+y)+z$.

Dato il grande intervallo di numeri che possono essere rappresentati in virgola mobile, nascono dei problemi quando si sommano due grossi numeri di segno opposto con un numero piccolo. Per esempio, si supponga che $x=-1{,}5_{dieci} \times 10^{38}$, $y=1{,}5_{dieci} \times 10^{38}$ e $z=1{,}0$, e che tutti questi siano numeri in singola precisione. Allora

$$\begin{aligned} x+(y+z) &= -1{,}5_{dieci} \times 10^{38} + (1{,}5_{dieci} \times 10^{38} + 1{,}0) = \\ &= -1{,}5_{dieci} \times 10^{38} + (1{,}5_{dieci} \times 10^{38}) = 0{,}0 \\ (x+y)+z &= (-1{,}5_{dieci} \times 10^{38} + 1{,}5_{dieci} \times 10^{38}) + 1{,}0 = \\ &= (0{,}0_{dieci}) + 1{,}0 = \\ &= 1{,}0 \end{aligned}$$

Dal momento che i numeri in virgola mobile hanno una precisione limitata a danno luogo ad una approssimazione dei valori reali, $1{,}5_{dieci} \times 10^{38}$ è talmente più grande di $1{,}0_{dieci}$ che $1{,}5_{dieci} \times 10^{38} + 1{,}0$ è ancora uguale a $1{,}5_{dieci} \times 10^{38}$. Per questa ragione la somma di x, y e z vale 0,0 oppure 1,0, a seconda dell'ordine con cui vengono eseguite le somme in virgola mobile, e quindi la somma in virgola mobile *non* è associativa.

Errore: come un'istruzione di scalamento a sinistra può sostituire una moltiplicazione intera per una potenza di 2, così uno scalamento a destra coincide con una divisione per una potenza di 2.

Si ricordi che un numero binario x, per il quale x*i* indica l'*i*-esimo bit, rappresenta il numero

$$...+(x3\times2^3)+(x2\times2^2)+(x1\times2^1)+(x0\times2^0)$$

Lo scalamento a destra dei bit di x di *n* posizioni sembra essere la stessa cosa della divisione per 2^n. E questo *è* vero per gli interi senza segno. Il problema nasce con gli interi con segno. Per esempio, si supponga di voler dividere -5_{dieci} per 4_{dieci}; il quoziente dovrebbe essere -1_{dieci}. La rappresentazione in complemento a due di -5_{dieci} è

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{due}$$

Sulla base dell'errore che si sta esaminando, eseguendo lo scalamento a destra di due posizioni si dovrebbe dividere per 4_{dieci} (2^2):

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{due}$$

Dal momento che vi è uno 0 nel bit di segno, il risultato è chiaramente errato. Il valore creato dallo scalamento a destra è in realtà $1.073.741.822_{dieci}$ anziché -1_{dieci}.

Una soluzione sarebbe rappresentata dal disporre di un'operazione di scalamento a destra aritmetico che estenda il bit di segno anziché inserire nella parola degli 0 da sinistra. Un'operazione di scalamento aritmetico a destra di 2 bit eseguita su -5_{dieci} produce

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{due}$$

Il risultato vale -2_{dieci} anziché -1_{dieci}; vicino al desiderato, ma non ancora esatto.

Il PowerPC tuttavia possiede di una istruzione veloce di scalamento (*shift right algebraic*) che in combinazione con una somma speciale (somma con riporto) dà lo stesso risultato della divisione per una potenza di 2.

Trabocchetto: l'istruzione del MIPS add immediate unsigned `addiu`*, malgrado il suo nome, estende il segno del proprio campo immediato su 16 bit.*

Malgrado il suo nome, `addiu` è utilizzata per sommare costanti a interi con segno allorquando non si ha interesse per un eventuale overflow. Il MIPS non ha un'istruzione di sottrazione con immediato ed i numeri negativi richiedono l'estensione del segno, così che gli architetti del MIPS decisero di estendere il segno del campo immediato.

Errore: solo i matematici teorici si preoccupano dell'accuratezza dei calcoli in virgola mobile.

I titoli dei giornali del Novembre 1994 dimostrano che questa affermazione è sbagliata (si veda la figura 4.51). Qui è riportata la storia dietro i titoli.

Il Pentium usa un algoritmo standard di divisione in virgola mobile che ad ogni passo genera diversi bit del quoziente, utilizzando i bit più significativi del divisore

Figura 4.51 Un campione di articoli di giornali e riviste del Novembre 1994, in cui compaiono il *New York Times*, il *San Jose Mercury News*, il *San Francisco Chronicle* e *Infoworld*. Il baco nella divisione in virgola mobile del Pentium entrò anche nella "Top 10 List" del *David Letterman Late Show* in televisione. La Intel spese circa 300 milioni di dollari per rimpiazzare i chip bacati.

e del dividendo per indovinare i due bit successivi del quoziente. Il valore proposto deriva da una tabella di lookup che contiene -2, -1, 0, +1, o +2. Il valore proposto viene moltiplicato per il divisore e sottratto dal resto per generare un nuovo resto. Come nella divisione senza ripristino (si veda l'Esercizio 4.54), se un valore valutato nel passo precedente produce un resto troppo grande, il resto parziale viene risistemato in un passo successivo.

Evidentemente, vi erano cinque elementi della tabella derivata dall'80486 ai quali la Intel pensò che non si verificassero mai degli accessi, e la PLA del Pentium venne quindi ottimizzata in modo da restituire 0 anziché 2 in queste situazioni. La Intel sbagliava: mentre i primi 11 bit erano sempre corretti, si presentavano occasionalmente degli errori nei bit dal 12 al 52, ossia nelle cifre decimali dalla quarta alla quindicesima.

Viene ora riportata la sequenza temporale degli eventi che caratterizzarono la commedia morale del baco del Pentium:

- *Luglio 1994*: la Intel scopre il baco nel Pentium. Il costo effettivo per correggere il baco era di alcune centinaia di migliaia di dollari. Seguendo le normali procedure per la correzione dei bachi, sarebbero stati necessari vari mesi per fare le modifiche, ri-verificare il progetto, e mettere in produzione il chip corretto. La Intel decise di mettere in produzione i chip buoni nel Gennaio 1995, valutando che sarebbero stati prodotti da 3 a 5 milioni di Pentium con il baco.
- *Settembre 1994*: un professore di matematica del Lynchburg College in Virginia, Thomas Nicely, scopre il baco. Dopo aver chiamato il supporto tecnico Intel e non aver ricevuto nessuna risposta ufficiale, egli rivela su Internet la sua scoperta. L'iniziativa suscita immediatamente un grande clamore ed alcuni osservano come anche piccoli errori divengono grandi quando

moltiplicati per grandi numeri: la percentuale di persone con una malattia rara moltiplicata per la popolazione europea può, per esempio, condurre ad una stima errata del numero di persone malate.

- *7 Novembre 1994*: la rivista *Electronic Engineering Times* mette in copertina la vicenda, che viene rapidamente ripresa da altri giornali.
- *22 Novembre 1994*: la Intel rilascia un comunicato stampa in cui definisce il problema una "anomalia". Il Pentium "può produrre degli errori sulla nona cifra... Anche nel caso della maggior parte degli ingegneri e degli analisti finanziari, viene richiesta un'accuratezza che arriva solo alla quarta o alla quinta cifra decimale. Gli utenti di fogli elettronici e i programmi di videoscrittura non devono preoccuparsi. ... Ci sono forse una dozzina di persone che saranno interessate dal problema. Finora, abbiamo saputo di uno solo... [Solo] i matematici teorici devono preoccuparsi degli elaboratori Pentium acquistati prima dell'estate." Ciò che irritò molti fu il fatto che i clienti vennero invitati a descrivere la loro applicazione alla Intel, e poi la Intel avrebbe deciso se la loro applicazione meritasse un nuovo Pentium senza il baco della divisione, o meno.
- *5 Dicembre 1994*: la Intel sostiene che il malfunzionamento si verifica soltanto una volta ogni 27.000 anni per il tipico utente di fogli elettronici. La Intel assume che un utente esegua 1000 divisioni al giorno **e** moltiplica la probabilità di errore nell'ipotesi che i numeri in virgola mobile utilizzati siano casuali, ottenendo un rapporto di 1 ogni 9 miliardi, ottenendo 9 milioni di giorni, o 27.000 anni. La situazione inizia a calmarsi, malgrado la Intel rifiuti di spiegare perché il tipico utente dovrebbe far uso di numeri in virgola mobile generati casualmente.
- *12 Dicembre 1994*: la Divisione Ricerca della IBM contesta i calcoli della Intel sulla frequenza degli errori (è possibile accedere a questo articolo visitando il sito *www.mkp.com/books_catalog/cod/links.htm*). La IBM sostiene che un comune programma per la gestione di fogli elettronici, che esegua calcoli per 15 minuti al giorno, potrebbe produrre errori dovuti al Pentium con la frequenza di uno ogni 24 giorni. IBM fa l'ipotesi di 5000 divisioni al secondo, e 15 minuti di lavoro al giorno, ottenendo 4,2 milioni di divisioni al giorno, e non ipotizza una distribuzione casuale dei numeri, calcolando invece le probabilità come una su 100 milioni. Come risultato la IBM interrompe immediatamente la distribuzione di tutti i personal computer IBM basati sul Pentium. La situazione diventa nuovamente rovente per la Intel.
- *21 Dicembre 1994*: la Intel rilascia il seguente comunicato, firmato dal presidente della Intel, dall'amministratore delegato, dal direttore operativo **e** dal presidente del Consiglio di Amministrazione: "Noi della Intel desideriamo scusarci sinceramente per il modo in cui abbiamo recentemente reso pubblico il malfunzionamento del processore Pentium. Il simbolo Intel Inside significa che il vostro calcolatore possiede un microprocessore che non è secondo a nessuno in termini di qualità e prestazioni. Migliaia di dipendenti della Intel lavorano con impegno per far sì che questo sia vero. Ma nessun microprocessore è mai perfetto. Ciò che la Intel continua a credere è che un problema tecnicamente di second'ordine ha iniziato ad avere una propria vicenda autonoma. Benché la Intel sostenga fermamente la qualità dell'attuale versione del processore Pentium, riconosciamo che molti utenti sono preoccupati. Vogliamo fugare queste preoccupazioni. La Intel sostituirà l'attuale versione del processore Pentium con la versione aggiornata, in cui il difetto della divisione in virgola mobile è corretto, a qualunque proprietario lo richieda, senza alcun costo, per tutta la durata della vita del suo calcolatore." Gli analisti stimano che questa misura sia costata all'Intel 300 milioni di dollari.

Questa vicenda evidenzia alcuni punti che devono essere ponderati da ciascuno. Quanto più economico sarebbe stato riparare il baco nel Luglio 1994? Qual è stato il costo per riparare al danno di immagine della Intel? E quali sono le responsabilità di una società nel rendere pubblici i difetti di un prodotto così ampiamente usato, e su cui si fa tanto affidamento come un microprocessore?

Nell'Aprile del 1997 un altro baco relativo all'aritmetica in virgola mobile fu individuato nei processori Pentium Pro e Pentium II. Quando le istruzioni di store da virgola mobile ad intero (`fist`, `fistp`) incontrano un numero in virgola mobile negativo che è troppo grande per stare in una parola di 16 o 32 bit dopo essere stato convertito in intero, esse forzano il bit sbagliato nella parola di stato FPO (precision exception anziché invalid operation exception). Ad onore della Intel, bisogna dire che questa volta essa riconobbe pubblicamente il baco, ed offrì una procedura software di correzione per gestire il problema; una reazione piuttosto diversa da quella avuta nel 1994.

4.11 Considerazioni conclusive

L'aritmetica dei calcolatori si distingue dall'aritmetica con carta e penna per i vincoli dettati dalla precisione limitata. Questo limite può produrre delle operazioni non permesse allorquando si calcolano dei numeri più grandi o più piccoli dei limiti prefissati. Alcune anomalie, chiamate "overflow" o "underflow", possono produrre delle eccezioni o interrupt, che corrispondono a eventi eccezionali simili a chiamate a procedure non previste. Il Capitolo 5 discute le eccezioni in maggior dettaglio.

L'aritmetica in virgola mobile presenta in più il problema di essere un'approssimazione dei numeri reali, ed occorre molta attenzione per garantire che il numero scelto sul calcolatore sia la rappresentazione più prossima al numero effettivo. Il problema della mancanza di precisione e della limitatezza della rappresentazione sono oggetto di studio nel campo dell'analisi numerica.

Nel corso degli anni l'aritmetica dei calcolatori è stata oggetto di una diffusa opera di standardizzazione che ha grandemente migliorato la portabilità dei programmi. Nella stragrande maggioranza dei calcolatori oggi in commercio è possibile trovare l'aritmetica binaria in complemento a due e l'aritmetica binaria in virgola mobile dello standard IEEE 754. Ad esempio ogni calcolatore da tavolo venduto da quando questo libro è stato stampato per la prima volta segue tali convenzioni.

Un effetto collaterale del modello di calcolatore a programma memorizzato è il fatto che le combinazioni di bit non hanno un significato intrinseco. La stessa combinazione di bit può rappresentare un intero con segno, un intero senza segno, un numero in virgola mobile, un'istruzione, e così via. È l'istruzione che opera sulla parola a determinarne il significato.

Insieme con la spiegazione dell'aritmetica dei calcolatori fatta in questo capitolo è stata descritta un'altra parte del set di istruzioni del MIPS. Una possibile fonte di confusione consiste nel legame tra le istruzioni coperte in questi capitoli, le istru-

zioni eseguite dai chip MIPS, e le istruzioni accettate dagli assemblatori MIPS. Le prossime figure cercano di rendere chiari questi legami.

La figura 4.52 elenca le istruzioni MIPS illustrate nei Capitoli 3 e 4. Il gruppo di istruzioni sulla sinistra della figura comprende quelle che sono state definite *istruzioni base*: le istruzioni della parte destra sono state invece chiamate *istruzioni aritmetiche*. A sinistra nella figura 4.53 vi sono le istruzioni eseguite dal processore MIPS che non si trovano nella figura 4.52; l'insieme completo delle istruzioni è denominato *MIPS I*. A destra nella figura 4.53 vi sono le istruzioni riconosciute dall'assemblatore che non fanno però parte di MIPS I; questo insieme di istruzioni è stato chiamato *Pseudo MIPS*.

Istruzioni base MIPS	Nome	Formato	Istruzioni aritmetiche MIPS	Nome	Formato
Somma	add	R	Moltiplicazione	mult	R
Somma immediata	addi	I	Moltiplicazione senza segno	multu	R
Somma senza segno	addu	R	Divisione	div	R
Somma immediata senza segno	addiu	I	Divisione senza segno	divu	R
Sottrazione	sub	R	Spostamento da Hi	mfhi	R
Sottrazione senza segno	subu	R	Spostamento da Lo	mflo	R
And	and	R	Sposta dal controllo di sistema (EPC)	mfc0	R
And immediato	andi	I	Somma in virgola mobile in singola precisione	add.s	R
Or	or	R	Somma in virgola mobile in doppia precisione	add.d	R
Or immediata	ori	I	Sottrazione in virgola mobile in singola precisione	sub.s	R
Scalamento logico a sinistra	sll	R	Sottrazione in virgola mobile in doppia precisione	sub.d	R
Scalamento logico a destra	srl	R	Moltiplicazione in virgola mobile in singola precisione	mul.s	R
Caricamento della parte superiore immediato	lui	I	Moltiplicazione in virgola mobile in doppia precisione	mul.d	R
Caricamento di parola	lw	I	Divisione in virgola mobile in singola precisione	div.s	R
Memorizzazione di parola	sw	I	Divisione in virgola mobile in doppia precisione	div.d	R
Caricamento di byte senza segno	lbu	I	Caricamento di parola in registro in virgola mobile in singola precisione	lwc1	I
Memorizzazione di byte	sb	I	Memorizzazione di parola in registro in virgola mobile in singola precisione	swc1	I
Salto se uguale	beq	I	Salta se vero in virgola mobile	bc1t	I
Salto se non uguale	bne	I	Salta se falso in virgola mobile	bc1f	I
Salto	j	I	Confronto tra numeri in virgola mobile in singola precisione	c.x.s	R
Salto e collegamento	jal	I	(x=eq,neq,lt,le,gt,ge)		
Salto tramite registro	jr	R	Confronto tra numeri in virgola mobile in doppia precisione	c.x.d	R
Forza se minore di	slt	R	(x=eq,neq,lt,le,gt,ge)		
Forza se minore di immediato	slti	I			
Forza se minore di senza segno	sltu	R			
Forza se minore di immediato senza segno	sltiu	I			

Figura 4.52 Il set di istruzioni MIPS introdotto sinora. Questo libro si concentra sulle istruzioni nella colonna di sinistra.

Rimanenti istruzioni base MIPS	Nome	Formato	Pseudo MIPS	Nome	Formato
Or esclusivo (rs ⊕ rt)	xor	R	Spostamento	move	rd,rs
Or esclusivo immediato	xori	I	Valore assoluto	abs	rd,rs
Nor (¬(rs∨rt))	nor	R	Not (¬rs)	not	rd,rs
Scalamento aritmetico a destra	sra	R	Negazione (con segno o senza segno)	neg*s*	rd,rs
Scalamento logico a sinistra di variabile	sllv	R	Rotazione a sinistra	rol	rd,rs,rt
Scalamento logico a destra di variabile	srlv	R	Rotazione a destra	ror	rd,rs,rt
Scalamento aritmetico a destra di variabile	srav	R	Moltiplicazione senza controllo di overflow (con segno o senza segno)	mul*s*	rd,rs,rt
Scalamento aritmetico a destra di variabile	srav	R	Moltiplicazione con controllo di overflow (con segno o senza segno)	mulo*s*	rd,rs,rt
Spostamento in Hi	mthi	R	Divisione con controllo di overflow	div	rd,rs,rt
Spostamento in Lo	mtlo	R	Divisione senza controllo di overflow	divu	rd,rs,rt
Caricamento di mezza parola	lh	I	Resto (con segno o senza segno)	rem*s*	rd,rs,rt
Caricamento di mezza parola senza segno	lhu	I	Caricamento con immediato	li	rd,imm
Immagazzinamento mezza parola	sh	I	Caricamento di indirizzo	la	rd,addr
Caricamento parola di sinistra (non allineato)	lwl	I	Caricamento doppio	ld	rd,addr
Caricamento parola di destra (non allineato)	lwr	I	Immagazzinamento doppio	sd	rd,addr
Immagazzinamento della parola di sinistra (non allineato)	swl	I	Caricamento di parola non allineato	ulw	rd,addr
Immagazzinamento della parola di destra (non allineato)	swr	I	Immagazzinamento di parola non allineato	usw	rd,addr
Salto se minore di zero	bltz	I	Caricamento di mezza parola non allineato (con segno o senza segno)	ulh*s*	rd,addr
Salto se minore o uguale a zero	blez	I	Immagazzinamento di mezza parola non allineato	ush	rd,addr
Salto se maggiore di zero	bgtz	I	Salto	b	Etichetta
Salto se minore o uguale a zero	bgez	I	Salto se uguale a zero	beqz	rs,L
Salto se ≥ a zero e collegamento	bgezal	I	Salto se ≥ (con segno o senza segno)	bge*s*	rs,rt,L
Salto se < di zero con collegamento	bgezal	I	Salto se > (con segno o senza segno)	bgt*s*	rs,rt,L
Salto con collegamento tramite registro	jalr	R	Salto se ≤ (con segno o senza segno)	ble*s*	rs,rt,L
Ritorno da eccezione	rfe	R	Salto se < (con segno o senza segno)	blt*s*	rs,rt,L
Chiamata di sistema	syscall	R	Set equal	seq	rd,rs,rt
Interruzione (causa un'eccezione)	break	R	set not equal	sne	rd,rs,rt
Spostamento da VM a intero	mfc1	R	Set greater or equal (con segno o senza segno)	sge*s*	rd,rs,rt
Spostamento a VM da intero	mtc1	R	Set greater (con segno o senza segno)	sgt*s*	rd,rs,rt
Spostamento VM (s o d)	mov*f*	R	Set less or equal (con segno o senza segno)	sle*s*	rd,rs,rt
Valore assoluto VM (s o d)	abs*f*	R	Set less than (con segno o senza segno)	sle*s*	rd,rs,rt
Negazione VM (s o d)	neg*f*	R	Caricamento in virgola mobile (s o d)	l*f*	rd,addr
Conversione VM (w,s o d)	cvt*ff*	R	Immagazzinamento da VM (s o d)	s*f*	rd,addr
Confronto VM senza segno (s o d)	c.xn*f*	R			

Figura 4.53 I restanti set di istruzioni del MIPS I e dello "Pseudo MIPS". L'Appendice A descrive tutte queste istruzioni. Il simbolo *f* indica le versioni in singola (s) e doppia (d) precisione dell'istruzione in virgola mobile, mentre s indica le versioni con segno e senza segno (u).

La figura 4.54 riporta la frequenza delle istruzioni del MIPS per due programmi, gcc e spice. Sono elencate tutte le istruzioni responsabili di almeno lo 0,5% del totale delle istruzioni eseguite. La seguente tabella riassume le informazioni relative:

Sottoinsieme di istruzioni	gcc	spice
Istruzioni base	95%	45%
Istruzioni aritmetiche	0%	49%
Istruzioni rimanenti del MIPS I	5%	6%

Per il resto del libro ci si concentrerà sulle istruzioni base del MIPS (il set di istruzioni intere ad esclusione di moltiplicazione e divisione) per rendere più semplice la spiegazione del progetto di un calcolatore. Come si può vedere, le istruzioni base del MIPS includono tutte le istruzioni più frequenti, e si può stare certi che la comprensione di un calcolatore che esegue le istruzioni base del MIPS fornirà basi sufficienti per comprendere anche elaboratori più complessi.

MIPS base	Nome	gcc	spice	Aritmetiche base + MIPS	Nome	gcc	spice
Somma	add	0%	0%	Somma VM in doppia precisione	add.d	0%	4%
Somma immediata	addi	0%	0%	Sottrazione VM in doppia precisione	sub.d	0%	3%
Somma senza segno	addu	9%	10%	Moltiplicazione VM in doppia precisione	mul.d	0%	5%
Somma immediata senza segno	addiu	17%	1%	Divisione VM in doppia precisione	div.d	0%	2%
Sottrazione senza segno	subu	0%	1%	Caricamento parola VM in singola precisione	l.s	0%	24%
And	and	1%	0%	Immagazzinamento parola VM in singola precisione	s.s	0%	9%
And immediato	andi	2%	1%	Salto se vero in VM	bclt	0%	1%
Scalamento logico a sinistra	sll	5%	5%	Salto se falso in VM	bclf	0%	1%
Scalamento logico a destra	srl	0%	1%	Confronto VM in doppia precisione	c.x.d	0%	1%
Caricamento parte superiore immediato	lui	2%	6%	Spostamento verso VM	mtc1	0%	2%
Caricamento parola	lw	21%	7%	Spostamento da VM	mfc2	0%	2%
Immagazzinamento parola	sw	12%	2%	Conversione da float a intero	cvt	0%	1%
Caricamento byte	lb	1%	0%	Scalamento aritmetico a destra	sra	2%	0%
Immagazzinamento byte	sb	1%	0%	Caricamento metà parola	lh	1%	0%
Salto se uguale (zero)	beq	9%	3%	Salto se minore di zero	bltz	1%	0%
Salto se non uguale (zero)	bne	8%	2%	Salto se maggiore o uguale a zero	bgez	1%	0%
Salto con collegamento	jal	1%	1%	Salto se minore o uguale a zero	blez	0%	1%
Salto con registro	jr	1%	1%				
Set less than	slt	2%	0%				
Set less than immediato	slti	1%	0%				
Set less than senza segno	sltu	1%	0%				
Set less than immediato senza segno	sltiu	1%	0%				

Figura 4.54 Frequenza delle istruzioni MIPS nei due programmi gcc e spice. Il calcolo è basato sull'uscita "pixie" del MIPS I completo. Pixie è uno strumento di misura delle istruzioni distribuito da MIPS. Tutte le istruzioni che sono responsabili di almeno lo 0,5% del totale delle istruzioni eseguite o in gcc o in spice sono state incluse nella tabella, e quindi le istruzioni di moltiplicazione e divisione tra interi non sono elencate in quanto sono responsabili di meno dello 0,5% delle istruzioni eseguite. Le pseudoistruzioni sono convertite in MIPS I prima dell'esecuzione, e quindi non compaiono.

4.12 Inquadramento storico e approfondimenti

La legge di Gresham ("La cattiva moneta fa scappare Iddio") per i calcolatori direbbe "Il Veloce fa scappare il Lento anche se il Veloce è sbagliato".

W. Kahan, 1992

A prima vista potrebbe essere difficile immaginare un argomento di minor interesse della correttezza dell'aritmetica dei calcolatori o della sua accuratezza, ed ancor più difficile capire perché un argomento così vecchio e matematico dovrebbe essere fonte di tante discussioni. L'aritmetica dei calcolatori è tanto antica quanto l'informatica stessa, ed alcune delle nozioni più vecchie sull'argomento, come la riutilizzazione dei registri durante moltiplicazioni e divisioni in sequenza, ancor'oggi esigono rispetto. Maurice Wilkes [1985] ricordò una conversazione a proposito di tale concetto, svoltasi durante la sua visita negli Stati Uniti nel 1946, prima ancora che fosse costruito il primo elaboratore stored-program:

> *...stava per partire un progetto sotto la direzione di von Neumann presso l'Institute of Advanced Studies a Princeton ... Goldstine mi spieghò le caratteristiche principali del progetto, compreso il dispositivo nel quale le cifre del moltiplicatore dovevano venire scritte nella coda dell'accumulatore e scalate fuori mano a mano che la parte meno significativa del prodotto veniva scalata all'interno. Io espressi ammirazione per il modo in cui i registri ed i circuiti per lo scalamento erano organizzati ... e Goldstine notò che idee di quella natura venivano con grande facilità a von Neumann.*

Non ci può essere discussione su questo punto; difficilmente potrebbe esserci dissenso nell'area dell'aritmetica esatta intera, per lo meno sino a che vi è un accordo generale su quale intero rappresenta il risultato corretto. Tuttavia non appena entra in gioco l'aritmetica approssimata, subito nasce la discussione, come se ciò che è "trascurabile" per qualcuno fosse "la cosa più importante" per un altro.

La prima disputa

L'aritmetica in virgola mobile suscitò disaccordo ancor prima di essere implementata. John von Neumann era consapevole della proposta di Konrad Zuse per la realizzazione nel 1939 in Germania di un calcolatore che non fu mai costruito, probabilmente perché la virgola mobile lo rese così complesso da prevederne il completamento dopo la data di conclusione della Seconda Guerra Mondiale prevista dai tedeschi, e di conseguenza von Neumann rifiutò di inserirla nell'elaboratore che costruì a Princeton. In un rapporto di grande peso che scrisse nel 1946 insieme con H. H. Goldstine e A. W. Burks, egli elencò i punti a favore e contro la virgola mobile. A favore:

> *... per mantenere in una somma o prodotto quante più cifre significative sia possibile e ... per liberare l'operatore umano dell'incombenza di stimare ed inserire in un problema dei "fattori di scala", ossia delle costanti moltiplicative che servano a mantenere i numeri entro i limiti accettati dall'elaboratore.*

La virgola mobile fu esclusa per varie ragioni:

> *Non vi è ovviamente nessun dubbio che venga consumato del tempo-uomo nel gestire l'introduzione di opportuni fattori di scala. Ciò che sosteniamo è piuttosto che il tempo richiesto sia solo una percentuale molto piccola del tempo totale che si spenderà per preparare un problema di interesse per la nostra macchina. Il primo vantaggio della virgola mobile è, riteniamo, illusorio. Per disporre della virgola mobile si dovrebbe sprecare una quantità di memoria che potrebbe altrimenti essere usata per contenere un numero maggiore di cifre per parola. Ci sembra quindi non del tutto chiaro se il limitato vantaggio derivante dal disporre della virgola mobile compensi la perdita in termini di capacità di memoria e la crescita in termini di complessità dei circuiti aritmetici e di controllo.*

Il concetto sembra essere quello che la maggior parte dei bit destinati al campo esponente siano dei bit sprecati. L'esperienza ha provato il contrario.

Un approccio di tipo software per la gestione dei numeri reali senza i circuiti per la gestione dei numeri in virgola mobile fu chiamata *floating vectors*; l'idea consisteva nel calcolare al tempo di esecuzione un fattore di scala per un vettore completo di numeri, scegliendo il fattore di scala in maniera tale che il numero più grande nel vettore potesse essere contenuto di misura nel suo campo. A partire dal 1951, James H. Wilkinson aveva usato estesamente questo schema per il calcolo di matrici. Il problema si dimostrò essere il fatto che un programma può incontrare un valore molto alto, e quindi il fattore di scala deve tenere conto di questi numeri grandi e rari: i numeri normali hanno quindi molti 0 iniziali, dal momento che tutti i numeri devono utilizzare lo stesso fattore di scala. L'accuratezza era sacrificata in quanto i bit meno significativi venivano persi sulla destra per far stare gli 0 iniziali; questo spreco divenne ovvio agli utilizzatori dei primi elaboratori che visualizzavano tutti i loro bit di memoria come punti sui tubi a raggi catodici (come gli schermi televisivi), rendendo così visibile la perdita di precisione. Laddove la virgola mobile richiedeva di essere utilizzata, in pratica non esistevano alternative perseguibili.

Di conseguenza i circuiti per la virgola mobile vera e propria si diffusero a causa della loro utilità. Entro il 1957 essi erano presenti pressoché ovunque. Un'unità decimale in virgola mobile era disponibile per l'IBM 650, e presto le serie degli IBM 704, 709, 7090, 7094, ecc. avrebbero offerto hardware per la virgola mobile binaria in singola e doppia precisione.

Il risultato fu quindi che tutti i sistemi gestivano numeri in virgola mobile, ma ogni implementazione era diversa.

Diversità e portabilità

Poiché l'arrotondamento introduce degli errori in quasi tutte le operazioni in virgola mobile, lamentarsi per un po' di errore in più sembra un'inezia. Così per 20 anni nessuno si lamentò molto se tutte quelle operazioni si comportavano in maniera leggermente diversa sulle varie macchine. Se il software richiedeva dei trucchi e delle astuzie per aggirare quelle idiosincrasie e per fornire alla fine i risultati corretti su tutti i bit tranne su alcuni, quei trucchi erano considerati parte dell'arte di programmare. Per lungo tempo i calcoli su matrici ingannarono la maggior parte di coloro che non avevano cognizioni di analisi degli errori, e forse questo continua ad essere vero ancor'oggi. Questa è forse la ragione per cui la gente è ancora sorpresa che i calcoli su matrici numericamente stabili dipendano dalla qualità dell'aritmetica in un numero estremamente ridotto di punti, molti meno di quanti generalmente si creda. I libri di Wilkinson ed alcuni pacchetti software largamente utilizzati come Linpack e Eispack hanno sostenuto la falsa impressione, molto diffusa nei primi anni '70, che un po' di esperienza fosse sufficiente per produrre del software numerico *portabile*.

Portabile significa in questo caso che il software è distribuito in forma di codice sorgente in qualche linguaggio standard che può essere compilato ed eseguito su praticamente tutti gli elaboratori commercialmente significativi, e che svolgerà poi il suo compito nello stesso modo in cui un qualsiasi altro programma svolge il suo compito su quella macchina. Fino a che si è creduto che il software numerico consistesse interamente di formule matematiche indipendenti dall'elaboratore, la sua portabilità è stata data per acquisita; l'errore insito in questa falsa convinzione diverrà chiaro tra poco.

Pacchetti come Linpack e Eispack sono così costosi da sviluppare (oltre cento dollari per riga di Fortran) che non avrebbero potuto essere sviluppati senza il supporto del governo degli Stati Uniti; la loro portabilità era una pre-condizione per tale supporto. Nessuno però pensò di valutare quanto le varie componenti avessero contribuito a tale costo. Una componente era quella algoritmica, legata allo sviluppo di un algoritmo in grado di funzionare su almeno un calcolatore malgrado le limitazioni in termini di arrotondamento e overflow o underflow. Un'altra componente corrispondeva allo sforzo di ingegnerizzazione del software richiesto per raggiungere e rafforzare la portabilità sui diversi calcolatori commercialmente significativi in quel momento; tale componente crebbe divenendo sempre più onerosa mano a mano che fiorivano, intorno agli anni '70, aritmetiche in virgola mobile sempre più diversificate.

E malgrado ciò, quasi nessuno comprendeva quanto tale diversità contribuiva a gonfiare il costo di quei pacchetti software.

Un passo indietro

Una prima dimostrazione di come le diverse aritmetiche potessero portare a costi di sviluppo del software significativamente diversi si ebbe nel 1964. Successe ad una riunione dello SHARE, lo user group degli utenti dei mainframe IBM, in occasione del quale l'IBM annunciò il System/360, il successore della serie 7094. Uno dei relatori descrisse i trucchi a cui aveva dovuto ricorrere per raggiungere un accettabile livello di qualità per la libreria del S/360, qualità che non era comunque così alta come quella che era riuscito ad ottenere precedentemente per il 7094.

Una parte del problema avrebbe potuto essere prevista da von Neumann, se fosse stato ancora vivo: nel 1948 lui e Goldstine avevano pubblicato una lunga analisi di errori, così difficile da seguire e così pessimista che quasi nessuno vi prestò attenzione. Tuttavia essa predisse correttamente il fatto che i calcoli su grandi vettori di dati sarebbero probabilmente stati vittima di un troncamento con una frequenza più elevata della media statistica. Gli IBM S/360 avevano delle memorie più grandi rispetto ai 7094, e così i vettori di dati potevano crescere e diventare più grandi, e così fecero. A peggiorare la situazione, i S/360 possedevano delle parole di memoria per la singola precisione più piccole (32 bit anziché 36) ed utilizzavano una aritmetica meno precisa (esadecimale o base 16 rispetto a binaria o base 2), con una conseguente minor precisione nel caso peggiore (21 bit significativi rispetto a 27) rispetto ai vecchi 7094. Di conseguenza lo stesso software che aveva quasi sempre fornito una (quasi) soddisfacente accuratezza sui 7094, troppo spesso produceva risultati non accurati quando veniva eseguito sui S/360. La via più veloce per recuperare un'accuratezza adeguata consisteva nel sostituire le dichiarazioni di sin-

gola precisione del vecchio codice con altre di doppia precisione, prima di ricompilare per il S/360. Questo fece sì che la doppia precisione del S/360 venisse adoperata molto di più di quanto non fosse previsto.

I maggiori problemi per i primi S/360 furono causati dalla mancanza di un bit di guardia nella doppia precisione. Questa mancanza si manifestò nella moltiplicazione sotto forma di incapacità nell'identificare uguaglianze come $1{,}0*x=x$ poiché moltiplicando x per 1,0, l'ultima cifra esadecimale (4 bit) di x veniva persa. Analogamente se x e y erano molto vicini ma avevano esponenti diversi, la sottrazione causava la perdita dell'ultima cifra dell'operando più piccolo prima di calcolare x-y. Quest'ultima aberrazione nella doppia precisione mise in dubbio un prezioso teorema che la singola precisione rispettava allora (ed oggi): se $1/2 \leq x/y \leq 2$, allora non può verificarsi nessun errore di troncamento durante il calcolo di $x-y$, ed il calcolo è esatto.

Innumerevoli calcoli avevano beneficiato di questo teorema minore, il più delle volte inconsciamente, già decenni prima del suo annuncio formale e della sua dimostrazione: si beneficiava di quel risultato senza conoscerlo.

Le identità ed i teoremi riguardanti le relazioni esatte, che persistevano, malgrado il troncamento, con tutte le ragionevoli implementazioni di una aritmetica approssimata, non furono apprezzate sino a che non li si perdette. In precedenza si pensava che ciò che importava fosse soltanto la precisione (quante cifre significative erano memorizzate) e l'intervallo di rappresentazione (la distanza tra le soglie di overflow e di underflow); poiché la doppia precisione dei S/360 aveva più precisione ed un intervallo di rappresentabilità più ampio dei 7094, ci si aspettava che il software continuasse a funzionare almeno altrettanto bene. Non fu però così.

I programmatori, che facendo carriera erano diventati program manager, erano atterriti dal costo per la conversione del software del 7094 per renderlo eseguibile sui S/360. Un piccolo sottocomitato dello SHARE propose dei miglioramenti alla virgola mobile dei S/360. Tale comitato fu sorpreso e gratificato dal fatto di ottenere dall'IBM una buona parte di ciò che essi avevano richiesto, compresi gli importantissimi bit di guardia. Entro il 1968 questi cambiamenti erano stati estesi, con un costo considerevole, ai S/360 già installati; la cosa peggiore fu il fatto che i clienti persero la loro fiducia nell'infallibilità dell'IBM (una lezione che la Intel imparò 30 anni dopo). I dipendenti della IBM ancora rabbrividiscono al ricordo di quell'incidente.

Quelli che costruirono le bombe

Seymour Cray è stato associato per decenni ai calcolatori CDC e Cray che erano, all'epoca in cui egli li costruiva, i più grandi ed i più veloci esistenti. Egli aveva sempre avuto chiaro che cosa i suoi clienti richiedevano di più: la *velocità*. E gliela diede, anche se, così facendo, in campo aritmetico diede loro anche qualcosa più "interessante" di quanto chiunque altro fosse in grado di dare. Tra i suoi clienti vi erano i grandi laboratori del governo, come quelli di Livermore e di Los Alamos, dove venivano progettate le armi nucleari; le sfide poste da soluzioni aritmetiche "interessanti" erano piuttosto noiose per coloro che dovevano vincere le sfide poste di Madre Natura.

Probabilmente tutti potrebbero imparare a convivere con le idiosincrasie dell'aritmetica, se solo una delle idiosincrasie dei calcolatori dovesse essere tollerata. Al contrario, quando si sommano diverse anomalie di diversi calcolatori, il software muore della peggior morte. Ecco un esempio tratto da una delle macchine di Cray:

```
if (x == 0,0)    y = 17,0 else y = z/x;
```


Questa istruzione può essere interrotta da un errore di divisione per zero? Su un CDC 6600 poteva. Infatti vi era un conflitto tra il sommatore del 6600, nel quale `x` veniva confrontato con `0,0`, ed il moltiplicatore/divisore. Il confronto eseguito dal sommatore prendeva in considerazione i primi 13 bit di `x`, che erano sufficienti per distinguere lo zero dai normali numeri in virgola mobile non-nulli; il moltiplicatore ed il divisore esaminavano invece solo i primi 12 bit. Di conseguenza esistevano dei piccoli numeri che erano non-nulli per il sommatore ma nulli per il moltiplicatore ed il divisore! Per evitare disastri con questi piccoli numeri, i programmatori impararono a sostituire le istruzioni come quella riportata con quest'altra:

```
if (1,0*x == 0,0)    y = 17,0 else y = z/x;
```

Tuttavia quest'istruzione non poteva essere usata con tranquillità all'interno di un pacchetto software con ambizioni di portabilità, dal momento che produce strani malfunzionamenti su altri calcolatori progettati da Cray, quelli venduti dalla Cray Research Inc. Se `x` è così grande che `2,0*x` può produrre un overflow, allora anche `1,0*x` può produrre un overflow! L'overflow si verifica perché i calcolatori Cray verificano l'esponente del prodotto *prima* che l'esponente del prodotto sia stato normalizzato, e questo soltanto per risparmiare il ritardo di una singola porta AND.

Se si pensa che l'istruzione precedente possa ora essere usata con tranquillità all'interno di pacchetti software portabili, dal momento che i calcolatori della generazione dei CDC 6600 non sono più commercialmente significativi, si faccia attenzione, perché essa può produrre un overflow su un computer Cray anche se `z` è quasi piccolo come `x`; il problema in questo caso nasce dal fatto che il Cray non calcola `z/x`, ma `z*(1/x)`, ed il reciproco può produrre un overflow anche se il quoziente desiderato non può assolutamente produrlo. Una difficoltà analoga crea problemi anche nel caso degli Intel i860 usati nei calcolatori massicciamente paralleli. Un eventuale programmatore di codice portabile incontra innumerevoli dilemmi come quello presentato ogni volta che cerca di scrivere dei programmi per l'intera gamma dei calcolatori esistenti.

I calcolatori Cray sono anche afflitti da anomalie legate ad errori di arrotondamento, ben peggiori dell'anomalia legate all'overflow/underflow appena discussa. L'errore peggiore è causato dalla mancanza della cifra di guardia nella somma/sottrazione, problema che affliggeva gli IBM S/360. Ulteriori cattive notizie per il software venivano dal modo in cui Cray risparmiò sul suo moltiplicatore; circa un terzo dei bit generati da un normale vettore di moltiplicatori erano stati eliminati nel suo moltiplicatore, perché avrebbero contribuito per meno di una unità all'ultima posizione del prodotto finale arrotondato di Cray. Di conseguenza un moltiplicatore di Cray produce degli errori per almeno un bit in più di quanto ci si aspetterebbe. Tale errore si aggrava allorché la divisione utilizza tre moltiplicazioni per migliorare un reciproco approssimato del divisore, e moltiplica poi per questo il numeratore. La radice quadrata aggiunge qualche errore di moltiplicazione in più.

La velocità aveva sostituito la lentezza, anche se a volte era leggermente sbagliata.

Rendere sicuro il mondo per la virgola mobile, o viceversa

William Kahan era uno studente all'Università di Toronto nel 1953 quando imparò a programmare il suo calcolatore Ferranti-Manchester Mark-I. Poiché era entrato presto nel settore, Kahan guadagnò una buona esperienza con una vasta gamma di dispositivi e con una buona parte delle personalità attive nel mondo dell'elaborazione dati; il numero degli uni e degli altri era piccolo allora. Egli aveva eseguito

calcoli usando regoli calcolatori, calcolatori meccanici da tavolo, analizzatori differenziali analogici di tavolo, e così via; ad esclusione dei primi, utilizzò tutti i calcolatori elettronici e le macchine calcolatrici di cui si parla in questo libro.

Il desiderio di Kahan di rilasciare del software affidabile destò in lui l'interesse per l'analisi degli errori, che si intensificò durante i due anni di studio post-dottorato in Inghilterra, dove frequentò Wilkinson. Nel 1960 riprese ad insegnare a Toronto, dove era stato acquistato un IBM 7090 e dove gli fu data carta bianca per armeggiare con il sistema operativo, il compilatore Fortran e la libreria runtime. Egli nega di essere arrivato anche solo vicino alla circuiteria del 7090 con un saldatore, ma ammette di aver chiesto di farlo. Una vicenda di quel periodo è illuminante su come le concezioni sbagliate e le anomalie numeriche nei sistemi di elaborazione possono essere causa di incredibili costi nascosti.

Uno studente universitario in ingegneria aeronautica usava il 7090 per simulare le ali che stava progettando per decolli e atterraggi corti. Sapeva che quell'ala sarebbe stata difficile da controllare se le sue caratteristiche avessero permesso un improvviso inizio di stallo, ma pensava di poter evitare quel problema. Le simulazioni tuttavia gli dicevano il contrario. Per essere certo che il troncamento non interferisse, egli aveva ripetuto molti dei suoi calcoli usando la doppia precisione, ottenendo risultati molto simili a quelli ottenuti con la singola; le sue ali andavano improvvisamente in stallo con ambedue le precisioni. Sfiduciato, lo studente rinunciò.

Nel frattempo, Kahan aveva sostituito il programma per i logaritmi della IBM (ALOG) con uno di sua creazione, che sperava potesse dare una migliore accuratezza. Mentre lo controllava, Kahan fece di nuovo eseguire i programmi usando la nuova versione di ALOG; i risultati dello studente cambiarono in maniera significativa: Kahan lo contattò per capire che cosa era successo.

Lo studente era confuso: benché preferisse i risultati prodotti con il nuovo ALOG (essi predicevano uno stallo graduale), egli sapeva che questi dovevano essere sbagliati poiché non coincidevano con i risultati ottenuti con la doppia precisione. Le differenze tra i risultati della singola e della doppia precisione scomparvero qualche giorno più tardi quando arrivò una nuova versione del software IBM per il 7090 per la gestione dei numeri in doppia precisione (il 7090 non aveva un hardware per la doppia precisione). Lo studente andò avanti a scrivere la tesi su quel lavoro e nella costruzione delle ali, e queste si comportarono come previsto. Ma la storia non finisce qui.

Nel 1963, il 7090 fu sostituito con un più veloce 7094, dotato di hardware per la virgola mobile in doppia precisione e di un set di istruzioni per il resto praticamente identico a quello del 7090. Solo in doppia precisione, e solo quando si usava il nuovo hardware le ali andavano di nuovo bruscamente in stallo. Venne speso un mucchio di tempo per capire perché, e si scoprì che l'hardware del 7094, come il software obsoleto del 7090 e quello successivo dei primi S/360, mancava del bit di guardia in doppia precisione. Come molti programmatori che utilizzavano quelle macchine e quelle di Cray, lo studente scoprì un trucco per compensare la mancanza del bit di guardia, e scrisse l'espressione `(0,5-x)+0,5` al posto di `1,0-x`. Oggi chi dovesse spiegare perché sia necessario un tale trucco arrossirebbe, ma questo risolse il problema dello studente.

Nel frattempo il fascino della California stava facendo presa su Kahan e la sua famiglia: arrivarono a Berkeley, e lui all'Università della California. Gli si presentò una buona opportunità nel 1974, quando alcuni problemi di accuratezza spinsero i progettisti di macchine calcolatrici della Hewlett-Packard a cercare un consulente. Il consulente scelto fu Kahan, ed il suo lavoro fece migliorare drasticamente l'accuratezza delle macchine della HP, ma questa è un'altra storia. La proficua collaborazione con persone che gli erano congeniali, tuttavia, lo rafforzò in vista della successiva e cruciale occasione.

Questa si presentò nel 1976, quando John F. Palmer della Intel fu incaricato di specificare la "migliore aritmetica possibile" per tutta la linea di prodotti della Intel. L'8086 era imminente e per l'8086 già si pensava ad un coprocessore in virgola mobile come l'8087 (un *coprocessore* è semplicemente un chip aggiuntivo che rende più veloci una parte delle operazioni svolte da un processore; in questo caso accelera i calcoli in virgola mobile).

Palmer aveva conseguito il suo Dottorato di Ricerca pochi anni prima a Stanford, e sapeva chi chiamare per avere dei consigli di alto livello: Kahan. Essi misero insieme un progetto che ovviamente sarebbe stato impossibile solo pochi anni prima, e che sembrava non del tutto possibile per quei tempi. Ma un nuovo gruppo israeliano di dipendenti Intel, guidato di Rafi Navé si sentì sfidato a dimostrare le proprie capacità agli americani e propose di mettere qualcosa di impossibile su un chip, l'8087.

Fino ad allora l'aritmetica in virgola mobile, che aveva visto versioni veramente diverse tra i mainframe, era divenuta caotica tra i microprocessori, ognuno dei quali poteva ospitare fino ad una dozzina di versioni diverse dell'aritmetica, nel firmware su ROM o nel software. Robert G. Stewart, un ingegnere molto attivo nelle attività della IEEE, si stufò di questa anarchia e propose che la IEEE redigesse una proposta di standard per la virgola mobile. Nello stesso tempo nella Silicon Valley si sparse la voce che la Intel stava per mettere su un chip qualcosa di incredibile in termini di virgola mobile, molto al di là di quello che i concorrenti avevano in mente. La concorrenza doveva trovare un modo per rallentare la Intel, e così formarono un comitato che facesse ciò che Steward aveva richiesto.

Le prime riunioni del comitato si svolsero alla fine del 1977 e videro una pletora di proposte in competizione provenienti da innumerevoli fonti; le riunioni proseguirono sino al 1985 quando venne ufficializzato lo standard IEEE 754 per la rappresentazione in virgola mobile. La proposta vincente fu molto prossima a quella sottomessa da Kahan, dal suo studente Jerome T. Coonen, e da Harold S. Stone, un professore in visita a Berkeley in quel periodo. La loro proposta era basata sul progetto Intel, ovviamente con il permesso della Intel, nella versione semplificata da Coonen. La loro armoniosa combinazione di caratteristiche, quasi nessuna delle quali nuova, aveva alla fine polarizzato una maggiore approvazione all'interno del comitato e all'esterno (da parte ad esempio di esperti come Wilkinson) di ogni altra proposta, ma in ogni caso essi dovettero guadagnarsi un supporto pressoché unanime all'interno del comitato per conquistare l'approvazione della IEEE, e questo richiese molto tempo.

I primi chip compatibili con lo standard IEEE 754

Nel 1980 la Intel si stancò di aspettare e rilasciò l'8087 in modo che potesse essere usato nel PC IBM. L'architettura in virgola mobile dell'8087 dovette essere adeguata allo spazio lasciato inutilizzato dai codici operativi dell'8086, rendendo impossibile la presenza di due operandi per istruzione, come si trovano nel resto dell'8086. Di qui la decisione di usare un operando per istruzione e di adottare uno stack: "Il compito del progettista è di fare di Necessità Virtù" (la storia scritta da Kahan nel 1990 sulla scelta dell'architettura a stack per l'8087 è di piacevole lettura).

Piuttosto che adottare la classica architettura a stack, che non permette di evitare che sotto-espressioni frequenti siano estratte e inserite dalla memoria sulla cima di uno stack costituito da registri, la Intel cercò di combinare un register file piatto con uno stack. Il ragionamento alla base della scelta era che forzare un operando a coincidere con la cima dello stack non è una soluzione così cattiva, poiché richiede soltanto l'esecuzione di una istruzione `FXCH` (che scambia i registri), ottenendo lo stesso risultato di un'istruzione a due operandi, senza contare che la `FXCH` è molto più veloce delle operazioni in virgola mobile dell'8087.

Poiché le espressioni in virgola mobile non sono così complesse, Kahan pensò che uno stack di otto registri si sarebbe riempito soltanto raramente. Di conseguenza premette perché l'8087 usasse questo schema ibrido, con l'aggiunta che lo stack overflow o lo stack underflow dovessero generare un interrupt verso l'8086, così che l'interrupt software potesse dare l'illusione allo scrittore di compilatori di poter disporre di uno stack illimitato per i dati in virgola mobile.

L'Intel 8087 fu realizzato in Israele, e le 7500 miglia e 10 fusi orari di distanza resero difficili le comunicazioni con la California. Secondo Palmer e Morse (*The 8087 Primer*, J. Wiley, New York, 1984, pag. 93):

> *Sfortunatamente, nessuno provò a scrivere un gestore software dello stack prima della costruzione dell'8087, e a quel punto fu troppo tardi; ciò che era troppo complesso da eseguire in hardware si rivelò esserlo ancora di più in software. Ciò che si scoprì mancare era la possibilità di determinare efficientemente se una operazione non valida fosse davvero dovuta ad uno stack overflow. ...Inoltre mancava la possibilità di far ripartire l'istruzione che aveva provocato lo stack overflow...*

Il risultato fu che le eccezioni dello stack erano troppo lente per poter essere gestite in software. Come disse Kahan [1990]:

> *Di conseguenza quasi tutti i compilatori dei linguaggi di alto livello producono codice inefficiente per la famiglia 80x87, degradando tipicamente di un 50% le prestazioni del chip, con load e store spurie che servono soltanto ad evitare stack overflow e underflow ...*
>
> *Sono ancor'oggi dispiaciuto che l'implementazione dello stack dell'8087 non sia stata così pulita come era nelle mie originarie intenzioni. ... Se si fosse realizzato il progetto originario, i compilatori odierni utilizzerebbero l'8087 ed i suoi discendenti in maniera più efficiente, ed i concorrenti della Intel potrebbero più facilmente commercializzare imitazioni dell'80x87 più veloci ma compatibili.*

Nel 1982, la Motorola annunciò il suo 68881, che fu utilizzato nelle Sun 3 e nei Macintosh II: la Apple era stata un sostenitore della proposta sin dall'inizio. Un altro studente di Berkeley, George S. Taylor, progettò ben presto una implementazione ad alta velocità dello standard proposto, destinata ad uno dei primi superminicomputer (lo ELXSI 6400). Lo standard stava diventando un dato di fatto prima ancora che l'inchiostro della proposta finale fosse asciutto.

La corsa all'adozione immediata dello standard diede all'industria informatica la falsa impressione che lo IEEE 754, come molti altri standard, fosse facilmente implementabile seguendo una ricetta standard. Questo non era vero: solo l'entusiasmo e l'ingenuità dei primi implementatori lo fecero sembrare semplice.

Infatti implementare correttamente lo IEEE 754 richiede una diligente e straordinaria attenzione per i dettagli; per permettere una esecuzione veloce ci vogliono una competenza ed una semplicità di progetto straordinarie. Se i manager dell'industria avessero capito questo, non avrebbero affermato così precipitosamente, come scelta politica, "Siamo compatibili con tutti i possibili standard."

Lo standard IEEE 754 oggi

L'industria informatica odierna è preda di una ragnatela di standard che evolvono continuamente mano a mano che la tecnologia cambia. Gli standard per la virgola mobile IEEE 754/854 (sono praticamente la stessa cosa) si ergono in una sorta di splendido isolamento soltanto perché nessuno desidera ripetere la lunga e prolungata rissa che circondò la sua nascita allorquando, con una generosità senza precedenti, i rappresentanti degli interessi hardware accondiscesero alle richieste di quei pochi che rappresentavano gli interessi del software matematico e numerico.

Sfortunatamente la comunità degli scrittori di compilatori non era rappresentata adeguatamente nella zuffa, ed alcune caratteristiche non tengono nel giusto conto le problematiche dei linguaggi e dei compilatori. Tale comunità è stata lenta nel rendere disponibili ai programmatori di applicazioni le caratteristiche insolite dello IEEE 754. La gestione umana delle eccezioni è una di queste caratteristiche insolite, e l'arrotondamento diretto un'altra. Senza il supporto dei compilatori queste caratteristiche si sono atrofizzate.

I punti di maggior successo dello IEEE 754 sono il fatto che sia uno standard ampiamente implementato, e dotato di un formato comune per la virgola mobile, che richieda un'accuratezza minima per raggiungere mezza ulp nel bit meno significativo, e che le operazioni debbano essere commutative.

Al momento attuale gli standard IEEE 754/854 sono stati implementati con un considerevole livello di fedeltà in almeno una parte delle linee di prodotto di tutti i costruttori di calcolatori nordamericani. Le uniche eccezioni di rilievo sono il DEC VAX, i discendenti degli IBM S/370, ed i supercomputer vettoriali della Cray Research, ma tutte e tre stanno per essere sostituite da calcolatori compatibili con lo standard. Anche la Cray Research, che è ora una divisione della Silicon Graphics, ha annunciato che i successori del calcolatore vettoriale T90 saranno "in parte" compatibili, per facilitare il trasferimento di file di dati e di software portabile tra i Cray ed i calcolatori attraverso i quali gli utenti dei Cray devono oggi fare accesso alle loro macchine.

Nel 1989 la Association for Computing Machinery (ACM), riconoscendo i benefici derivati all'industria informatica dallo IEEE 754, onorò Kahan con il Turing Award; nell'accettarlo, egli ringraziò i suoi numerosi collaboratori per l'aiuto prestato, ed i suoi avversari per i loro errori.

In conclusione... non tutto il male viene per nuocere.

Letture consigliate

Se si vuole saperne di più sulla virgola mobile, le due pubblicazioni di David Goldberg [1991, 1995] sono un buon punto di partenza; esse abbondano di riferimenti ad ulteriori letture. Alcune delle vicende narrate in precedenza sono tratte dagli scritti di Kahan [1972, 1983]. L'ultima parola sullo stato dell'arte nell'aritmetica su calcolatore si trova spesso negli *Atti* degli ultimi Symposium on Computer Arithmetic, sponsorizzato dalla IEEE, che si svolge ogni due anni; la tredicesima edizione si è tenuta nel 1997.

Burks, A. W. , H. H. Goldstine, e J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," *Report to the U.S. Army Ordnance Dept.*, p. 1; anche riportato in *Papers of John von Neumann*, W. Aspray e A. Burks, editori, MIT Press, Cambridge, MA, e Tomash Publishers, Los Angeles, pag. 97-146, 1987.

Questo articolo è un classico e riporta alcune argomentazioni contro l'hardware per la virgola mobile.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys* 23(1), pag. 5-48.
Un'altra buona introduzione all'aritmetica in virgola mobile, con l'enfasi sugli aspetti software.

Goldberg, D. [1995]. "Computer arithmetic," *Appendix A of Computer Architecture: A Quantitative Approach*, seconda edizione, J. L. Hennessy e D. A. Patterson, Morgan Kaufmann Publishers, San Francisco.
Un'introduzione più approfondita all'aritmetica intera e in virgola mobile, con l'enfasi sull'hardware. Copre in sole 10 pagine i Paragrafi 4.6-4.8 di questo libro, dedicando poi 45 pagine ad argomenti più avanzati.

Kahan, W. [1972]. "A survey of error-analysis," in *Info. Processing71* (Proc. IFIP Congress 71 in Ljubljana), vol. 2, pag. 1214-39, North-Holland Publishing, Amsterdam.
Quest'opera generale è una ricca fonte di vicende sull'importanza della precisione nell'aritmetica.

Kahan, W. [1983]. "Mathematics written in sand," *Proc. Amer.Stat. Assoc. Joint Summer Meetings of 1983, Statistical Computing Section*, pag. 12-26.
Il titolo fa riferimento al silicio ed è un'altra fonte di aneddoti che illustrano l'importanza dell'aritmetica accurata.

Kahan, W. [1990]. "On the advantage of 8087's stack," dispense didattiche non pubblicate, Computer Science Division, University of California at Berkeley.
Che cosa avrebbe potuto essere l'architettura in virgola mobile dell'8087.

Kahan, W. [1997]. Disponibile tramite link alla pagina di Kahan alla URL *www.mkp.com/books_catalog/cod/links.htm.*
Una collezione di appunti relativi alla rappresentazione in virgola mobile, tra i quali "Beastly Numbers" (un altro baco del Pentium, meno famoso), "Notes on the IEEE Floating Point Arithmetic" (include alcuni commenti sul fatto che alcune particolarità dello standard stiano atrofizzandosi), e "The Baleful Effects of Computing benchmarks" (sull'insana attenzione per la velocità a scapito della correttezza, dell'accuratezza, della facilità d'uso, della flessibilità, ecc.).

Koren, I. [1993]. *Computer Arithmetic Algorithms*, Prentice Hall, Englewood Cliffs, NJ.
Un libro di testo destinato agli studenti universitari dei corsi avanzati, che spiega i principi fondamentali dell'aritmetica di base così come le operazioni complesse quali le funzioni logaritmiche e trigonometriche.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.
Questa collezione di ricordi di un pioniere dell'informatica comprende la descrizione dei circuiti standard per la moltiplicazione e la divisione, sviluppati da von Neuman,.

4.13 Parole chiave

I seguenti termini riprendono le idee chiave presentate nel capitolo. Si controlli il Glossario per la definizione dei termini su cui ci si sente insicuri.

algoritmo di Booth
arrotondamento
bit meno significativo
bit più significativo
divisore
doppia precisione
esadecimale
esponente
guardia
mantissa
normalizzato
notazione polarizzata
notazione scientifica
operazione AND
overflow
porta logica AND
porta logica OR esclusivo
quoziente
resto
singola precisione
sticky bit
underflow
unità aritmetico logica (arithmetic logic unit, o ALU)
unità nell'ultima posizione (ulp)
virgola mobile

4.14 Esercizi

Mai arrendersi, mai arrendersi, mai, mai, mai, in nessuna cosa, piccola o grande, importante o trascurabile, mai arrendersi.

Winston Churchill, saluto alla Harrow School, 1941

4.1 [3] <§4.2> Si converta 512_{dieci} in un numero binario in complemento a due su 32 bit.

4.2 [3] <§4.2> Si converta -1.023_{dieci} in un numero binario in complemento a due su 32 bit.

4.3 [5] <§4.2> Si converta $-4.000.000_{dieci}$ in un numero binario in complemento a due su 32 bit.

4.4 [5] <§4.2> Quale numero decimale rappresenta il seguente numero binario in complemento a due?

$1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0000\ 1100_{due}$

4.5 [5] <§4.2> Quale numero decimale rappresenta il seguente numero binario in complemento a due?

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{due}$

4.6 [5] <§4.2> Quale numero decimale rappresenta il seguente numero binario in complemento a due?

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{due}$

4.7 [5] <§4.2> Quale numero binario rappresenta il numero esadecimale $7fff\ fffa_{esa}$? E quale numero decimale?

4.8 [5] <§4.2> Quale numero esadecimale rappresenta il seguente numero binario in complemento a due?

$1100\ 1010\ 1111\ 1110\ 1111\ 1010\ 1100\ 1110_{due}$

4.9 [5] <§4.2> Perché il MIPS non ha una istruzione di sottrazione con immediato?

4.10 [10] <§4.2> Si trovi la più breve sequenza di istruzioni MIPS per la determinazione del valore assoluto di un intero in complemento a due. Si converta la seguente istruzione (accettata dall'assemblatore MIPS):

```
abs   $t2, $t3
```

Tale istruzione indica che il registro `$t2` contiene una copia del registro `$t3` se il registro `$t3` è positivo, ed il complemento a due del registro `$t3` se il registro `$t3` è negativo. Suggerimento: il problema può essere risolto con tre istruzioni.

4.11 [10] <§4.2> Due amici, Enrico e Davide, stanno discutendo. Enrico dice: "Tutti gli interi maggiori di zero e divisibili per sei hanno esattamente due 1 nella loro rappresentazione binaria." Davide non è d'accordo. Egli dice: "Non è vero, ma tali numeri hanno tutti un numero pari di 1 nella loro rappresentazione." Chi ha ragione, Enrico, Davide, o nessuno dei due? Suggerimento: si cerchino dei controesempi.

4.12 [15] <§4.4> Si consideri il seguente codice, usato per implementare l'istruzione

```
sllv  $s0, $s1, $s2
```

che utilizza i 5 bit meno significativi del valore nel registro `$s2` per specificare di quante posizioni il registro `$s1` debba essere scalato a sinistra:

```
          .data
mask:     .word  0xfffff83f
          .text
start:    lw     $t0, mask
          lw     $s0, shifter
          and    $s0, $s0, $t0
          andi   $s2, $s2, 0x1f
          sll    $s2, $s2, 6
          or     $s0, $s0, $s2
          sw     $s0, shifter
shifter:  sll    $s0, $s1, 0
```

Si aggiungano i commenti al codice e si scriva qualche frase per descrivere come funziona. Si noti che le due istruzioni `lw` sono pseudoistruzioni che usano un'etichetta per specificare l'indirizzo di memoria che contiene la parola di dato di caricare. Perché si può supporre che la scrittura di codice "automodificante" come questo sia una cattiva idea (spesso neppure consentita)?

4.13 [10] <§4.2> Se `A` è un indirizzo su 32 bit, una sequenza di istruzioni come la seguente:

```
lui  $t0, A_superiore
ori  $t0, $t0, A_inferiore
lw   $s0, 0($t0)
```

può tipicamente essere usata per caricare la parola all'indirizzo `A` in un registro (in questo caso `$s0`). Si consideri la seguente alternativa, più efficiente:

```
lui  $t0, A_superiore_modificato
lw   $s0, A_inferiore($t0)
```

Si descriva come `A_superiore` è modificato per permettere a questo semplice codice di funzionare. Suggerimento: `A_superiore` deve essere modificato perché `A_inferiore` sarà esteso con il segno).

4.14 [15] <§§3.4, 4.2, 4.8> Il Quadro di Insieme al fondo del Paragrafo 4.8 afferma che i bit non hanno un significato implicito. Data la sequenza di bit:

```
1000 1111 1110 1111 1100 0000 0000 0000
```

cosa rappresenta, assumendo che sia

a. un intero in complemento a due?
b. un intero senza segno?
c. un numero in virgola mobile in singola precisione?
d. un'istruzione MIPS?

Potrebbero essere utili le Figure 3.18, 4.48 ed A.18.

4.15 [10] <§§4.2, 4.4, 4.8> Questo esercizio è simile all'Esercizio 4.14, ma questa volta si usi la combinazione di bit:

```
0000 0000 0000 0000 0000 0000 0000 0000
```


4.16 [10] <§4.3> Una delle differenze tra l'architettura SPARC della Sun e l'architettura MIPS qui presentata consiste nel fatto che l'istruzione load word sulla SPARC può specificare l'indirizzo o come somma del contenuto di due registri o come contenuto di un registro più uno spiazzamento costante (cioè come fa il MIPS). L'articolo "An Analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks" ("Analisi sull'uso dei set di istruzioni del MIPS e delle SPARC sui programmi campione SPEC"), di R. F. Cmelik, S. I. Kong, D. R. Ditzel ed E. J. Kelly, presentato alla *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, Aprile 1991, riporta l'informazione che sull'architettura SPARC il 15% delle istruzioni load del programma gcc usa la versione registro-registro (senza che il registro sia mai `$zero`). Si assuma che lo stesso sia vero per il MIPS, se fosse modificato in maniera da avere questa opzione di indirizzamento aggiuntiva per le istruzioni `lw`. Usando i dati della figura 4.54, quale percentuale di istruzioni del programma gcc potrebbero essere eliminate con questa modifica architetturale? Perché?

4.17 [10] <§4.3> Si trovi la sequenza più breve di istruzioni MIPS per determinare se la somma di due registri, ad esempio `$t3` e `$t4`, genera un riporto. Si scriva uno 0 o un 1 nel registro `$t2` se il riporto è 0 o 1, rispettivamente. Suggerimento: il problema può essere risolto con due istruzioni.

4.18 [15] <§4.3> {Es. 4.17} Si trovi la sequenza più breve di istruzioni MIPS per eseguire la somma intera in doppia precisione. Si assuma che un intero in complemento a due su 64 bit si trovi nei registri `$t4` e `$t5` ed un altro sia contenuto nei registri `$t6` e `$t7`. La somma deve essere scritta nei registri `$t2` e `$t3`. In questo esempio la parola più significativa di ciascun intero su 64 bit si trova nei registri pari, e la parola meno significativa nei registri dispari. Suggerimento: il problema può essere risolto con quattro istruzioni.

4.19 [15] <§4.3> Si supponga che tutte le istruzioni di salto condizionato ad eccezione di `beq` e `bne` siano eliminate dal set di istruzioni del MIPS insieme con `slt` e tutte le sue varianti (`slti`, `sltu`, `sltui`). Si mostri come eseguire

```
slt  $t0, $s0, $s1
```

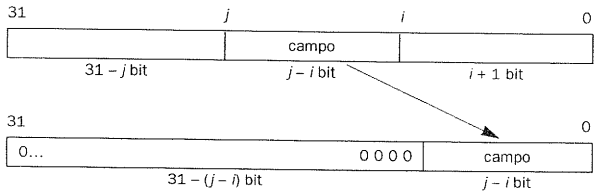
utilizzando il set di istruzioni modificato in cui `slt` non è disponibile. Suggerimento: la soluzione del problema richiede due istruzioni in più.

4.20 [10] <§4.4> La seguente sequenza di istruzioni MIPS potrebbe essere usata per implementare una nuova istruzione con due operandi corrispondenti a registri. Si dia un nome all'istruzione e se ne descriva il funzionamento. Si noti che il registro `$t0` è utilizzato come registro temporaneo.

```
srl  $s1, $s1, 1    #
sll  $t0, $s0, 31   # Queste 4 istruzioni implementano
srl  $s0, $s0, 1    # "new $s0 $s1"
or   $s1, $s1, $t0  #
```

4.21 [5] <§4.4> Anziché usare un apposito moltiplicatore hardware, è possibile eseguire la moltiplicazione utilizzando delle istruzioni di scalamento e somma. Questa soluzione è particolarmente interessante quando si deve moltiplicare per piccole costanti. Si supponga di voler mettere in `$s1` cinque volte il valore di `$s0`, ignorando qualsiasi eventuale possibilità di overflow. Si mostri una sequenza minima di istruzioni MIPS per eseguire tale operazione senza utilizzare un'istruzione di moltiplicazione.

4.22 [15] <§4.4> Alcuni calcolatori hanno delle istruzioni apposite per estrarre un campo arbitrario di bit da un registro di 32 bit e per scriverlo nei bit meno significativi di un altro registro. La figura seguente mostra l'operazione eseguita:



Si trovi la più breve sequenza di istruzioni MIPS in grado di estrarre un campo dal registro `$s0` e metterlo nel registro `$s1` usando i valori costanti i=7 e j=19. Suggerimento: la soluzione del problema richiede due istruzioni.

4.23 [15] <§4.5> La ALU implementa l'istruzione set on less than (`slt`) utilizzando soltanto il bit di segno del sommatore. Si provi un'operazione set on less than che usi i valori -7_{dieci} e 6_{dieci}. Per permettere di seguire più facilmente l'esempio, ci si limiti alla rappresentazione binaria su 4 bit: 1001_{due} e 0110_{due}.

$$1001_{due}-0110_{due}=1001_{due}+1010_{due}=0011_{due}$$

Questo risultato farebbe pensare che –7 > 6, che è chiaramente falso: quindi nella decisione si deve tenere conto dell'overflow. Si modifichi la ALU di 1 bit della figura 4.17 in maniera tale da gestire correttamente `slt`. Per risparmiare tempo, si eseguano le modifiche su una fotocopia della figura.

4.24 [20] <§4.6> Si trovi la più breve sequenza di istruzioni MIPS in grado di eseguire la moltiplicazione tra interi in doppia precisione. Si cerchi di risolvere il problema con 35 istruzioni o meno. Si assuma che un intero senza segno su 64 bit si trovi nei registri `$t4` e `$t5` ed un altro nei registri `$t6` e `$t7`. Il prodotto su 128 bit deve essere messo nei registri `$t0`, `$t1`, `$t2` e `$t3`. In questo esempio la parola più significativa si trova nei registri con indice più basso e la parola meno significativa in quelli con indice più alto. Suggerimento: si scriva la formula per $(a\times 2^{32}+b)\times(c\times 2^{32}+d)$.

4.25 [5] <§4.8> Si mostri la rappresentazione binaria prevista dallo IEEE 754 per il numero in virgola mobile 10_{dieci} in singola e doppia precisione.

4.26 [5] <§4.8> Si ripeta l'Esercizio 4.25, sostituendo il numero 10_{dieci} con $10{,}5_{dieci}$.

4.27 [10] <§4.8> Si ripeta l'Esercizio 4.25, sostituendo il numero 10_{dieci} con $0{,}1_{dieci}$.

4.28 [10] <§4.8> Si ripeta l'Esercizio 4.25, sostituendo il numero 10_{dieci} con la frazione decimale –2/3.

4.29 [10] <§4.8> Si scriva un semplice programma C che acquisisca un numero in virgola mobile e mostri la sua rappresentazione esadecimale.

4.30 [10] <§4.8> Si scriva un semplice programma C++ che acquisisca un numero in virgola mobile e mostri la sua rappresentazione esadecimale.

4.31 [10] <§4.8> Un numero in singola precisione rappresentato secondo lo standard IEEE 754 è immagazzinato in memoria all'indirizzo `X`. Si scriva una sequenza

di istruzioni MIPS per moltiplicare per 2 il numero in `X` e memorizzare il risultato di nuovo in `X`. Si esegua il tutto senza utilizzare alcuna istruzione in virgola mobile (non ci si preoccupi dell'overflow).

4.32 [10] <§4.11> Si trovino le 10 istruzioni MIPS eseguite più di frequente per il programma gcc (si veda la figura 4.54). Le si elenchi in ordine di frequenza, dalle più usate alle meno usate. Si mostri la posizione nella graduatoria, il nome e la percentuale di istruzioni eseguite per ciascuna istruzione. Se vi sono delle situazioni di parità per una certa posizione della graduatoria, si elenchino tutte le istruzioni con la stessa posizione, anche se questo significa elencare più di 10 istruzioni.

4.33 [10] <§4.11> Si ripeta l'Esercizio 4.32, sostituendo il programma gcc con il programma spice.

4.34 <§4.11> {Es. 4.32, 4.33} Le seguenti domande sono relative alla frequenza relativa delle istruzioni nei vari programmi.

a. [5] Quali istruzioni si trovano sia nella risposta all'Esercizio 4.32 che in quella all'Esercizio 4.33?

b. [5] Quale percentuale delle istruzioni eseguite da gcc è dovuta alle istruzioni identificate nell'Esercizio 4.34a?

c. [5] Quale percentuale delle istruzioni eseguite da gcc è dovuta alle istruzioni identificate nell'Esercizio 4.32?

d. [5] Quale percentuale delle istruzioni eseguite da spice è dovuta alle istruzioni identificate nell'Esercizio 4.34a?

e. [5] Quale percentuale delle istruzioni eseguite da spice è dovuta alle istruzioni identificate nell'Esercizio 4.32?

4.35 [10] <§4.11> {Es. 4.32-4.34} Se si stesse progettando un elaboratore per eseguire il set di istruzioni del MIPS, quali sarebbero le cinque istruzioni che si dovrebbe cercare di rendere più veloci possibile, sulla base delle risposte agli Esercizi dal 4.32 al 4.34? Si spieghi perché.

4.36 [15] <§§2.3, 4.11> Utilizzando la figura 4.54, si calcoli il numero medio di cicli di clock per istruzione (CPI) per il programma gcc. La figura 4.55 riporta il CPI medio per categoria di istruzioni, tenendo in considerazione gli effetti della cache ed altro. Si assuma che le istruzioni che non compaiono nella tabella abbiano un CPI pari a 1,0.

Categoria di istruzioni	CPI medio
Load e store	1,4
Salti condizionati	1,8
Salti	1,2
Moltiplicazione intera	10,0
Divisione intera	30,0
Somma e sottrazione in virgola mobile	2,0
Moltiplicazione in virgola mobile in singola precisione	4,0
Moltiplicazione in virgola mobile in doppia precisione	5,0
Divisione in virgola mobile in singola precisione	12,0
Divisione in virgola mobile in doppia precisione	19,0

Figura 4.55 Valori di CPI per le categorie di istruzioni MIPS.

4.37 [15] <§§2.3, 4.11> Si ripeta l'Esercizio 4.36, sostituendo il programma gcc con il programma spice.

4.38 [due settimane] Si scriva un simulatore per un sottoinsieme del set di istruzioni del MIPS utilizzando le istruzioni MIPS e il simulatore SPIM descritto nell'Appendice A. Il simulatore esegue programmi assemblati a mano posizionati nel segmento dati del simulatore SPIM ed usa `$v0` e `$v1` per l'input e l'output. Altre parti del segmento dati possono essere usate per immagazzinare il contenuto della memoria ed i valori dei registri della macchina virtuale considerata. L'implementazione può utilizzare qualunque istruzione MIPS, ma il simulatore deve supportare soltanto un sottoinsieme del set delle istruzioni (ad esempio le istruzioni introdotte nei Capitoli 5 e 6). Dettagli ulteriori su questo esercizio sono disponibili all'indirizzo *www.mkp.com/cod2e.htm*.

4.39 [una settimana] {Es. 4.38} Si aggiunga un gestore di eccezioni al simulatore sviluppato nell'Esercizio 4.38. Il simulatore dovrebbe generare una eccezione simulata se si fa accesso tramite una istruzione `lw`, `sw` o `jr` ad una parola non correttamente allineata. Il gestore delle eccezioni dovrebbe visualizzare un messaggio che identifica l'indirizzo incriminato (all'interno del simulatore) e poi riallineare l'accesso, eseguire l'istruzione, e riprendere l'esecuzione del programma che si sta simulando. Dettagli ulteriori su questo esercizio sono disponibili all'indirizzo *www.mkp.com/cod2e.htm*.

In maggior dettaglio

Le istruzioni logiche

Il set completo delle istruzioni MIPS comprende altre due operazioni logiche non menzionate sinora: `xor` e `nor`. L'operazione `xor` corrisponde all'OR esclusivo, mentre `nor` corrisponde a not OR. La tabella che segue definisce queste operazioni bit per bit. Tali istruzioni saranno poi utilizzate nei due esercizi che seguono.

A	B	A xor B	A nor B
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	0

4.40 [15] <§4.4> Si mostri la sequenza minima di istruzioni MIPS che implementa una nuova istruzione, denominata `swap`, che scambia due registri. Al termine della sequenza, il registro Destinazione ha il valore originario del registro Sorgente ed il registro Sorgente ha il valore originario del registro Destinazione.
Si converta l'istruzione

```
swap  $s0, $s1
```

La parte complessa della sequenza sta nel fatto che essa *deve utilizzare solo i due registri indicati*! Suggerimento: il problema può essere risolto con tre istruzioni, se si utilizzano delle nuove istruzioni logiche. Qual è il valore di (A xor B xor A)?

4.41 [5] <§4.4> Si mostri la sequenza minima di istruzioni MIPS che implementa una nuova istruzione, denominata `not`, che calcola il complemento a uno del registro Sorgente e lo scrive nel registro Destinazione. Si converta la seguente istruzione (accettata dall'assemblatore MIPS):

```
not  $s0, $s1
```

Suggerimento: il problema può essere risolto in due istruzioni, se si utilizzano le nuove istruzioni logiche.

4.42 [20] <§4.5> Un semplice controllo per l'overflow durante la somma consiste nel verificare se il segnale di CarryIn verso il bit più significativo *non* ha lo stesso valore del segnale CarryOut uscente dal bit più significativo. Si dimostri che questo controllo coincide con quello della figura 4.4.

4.43 [10] <§4.5> Si disegnino le porte logiche per il bit Somma di un sommatore, sulla base dell'equazione introdotta nel Paragrafo 4.5.

4.44 [5] <§4.5> Si riscriva l'equazione per la logica di carry-lookahead per un sommatore a 16 bit introdotta nel Paragrafo 4.5, usando una nuova notazione. Innanzitutto si assegnino dei nomi ai segnali di CarryIn dei singoli bit del sommatore; in particolare, si usi c4, c8, c12, ... anziché C1, C2, C3, ... Inoltre sia $P_{i,j}$ il segnale propaga per i bit da *i* a *j*, e sia $G_{i,j}$ il segnale genera per i bit da *i* a *j*. Per esempio, l'equazione

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

può essere riscritta come

$$c8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c0)$$

Tale notazione più generale può essere utile nella creazione di sommatori a maggior parallelismo.

4.45 [15] <§4.5> {Es. 4.44} Si scrivano le equazioni per la logica di carry-lookahead per un sommatore di 64 bit utilizzando la nuova notazione introdotta nell'Esercizio 4.44, e come blocchi elementari i sommatori a 16 bit. Si includa nella soluzione un disegno simile a quello della figura 4.24.

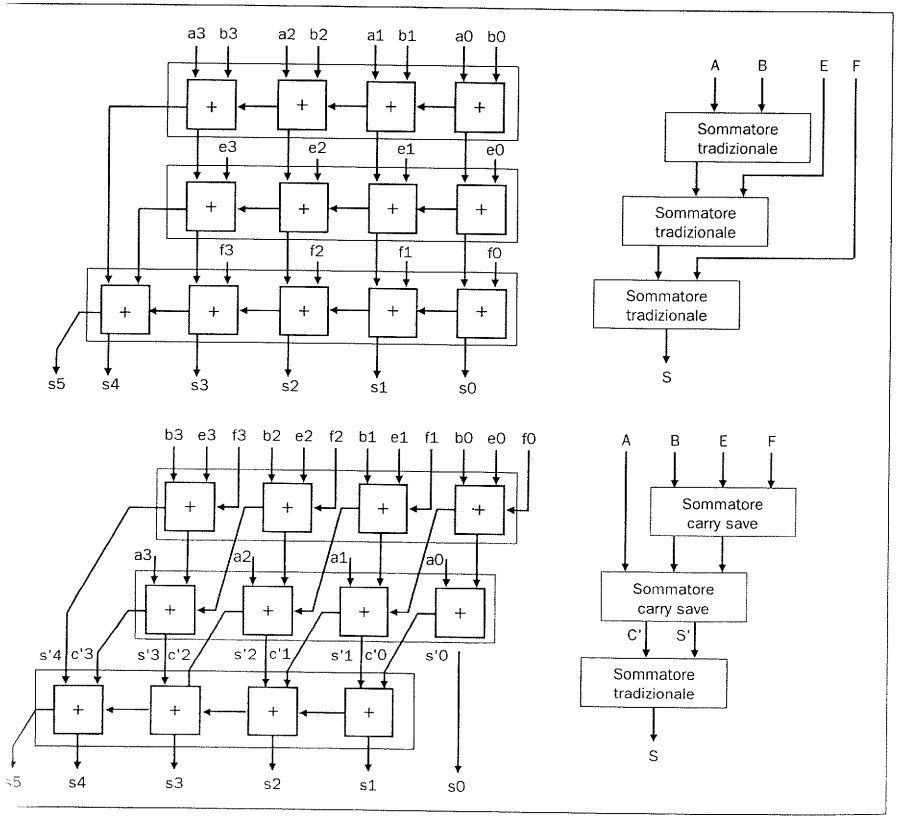
4.46 [10] <§4.5> Si calcolino ora le prestazioni relative dei sommatori. Si assuma che l'hardware corrispondente a ciascuna equazione contenente soltanto termini OR ed AND, come le equazioni per p_i e g_i introdotte nel Paragrafo 4.5, richieda un'unità di tempo T. Le equazioni che consistono dell'OR di vari termini AND, come le equazioni per c1, c2, c3 e c4, richiederanno quindi due unità di tempo (2T), poiché sarà necessario un tempo T per produrre i termini AND e poi un ulteriore tempo T per produrre il risultato dell'OR. Si calcolino i valori assoluti e relativi in termini di prestazioni per i sommatori su 4 bit, sia per quelli a propagazione del riporto (ripple carry) che per quelli di tipo carry lookahead. Se i termini nelle equazioni sono successivamente definiti da altre equazioni, si sommino gli appropriati ritardi per tali equazioni intermedie, e si continui ricorsivamente sino a che nell'equazione non si utilizzano i bit effettivi di ingresso del sommatore. Si includa un disegno di ciascun sommatore, etichettato con i ritardi calcolati, evidenziando il cammino del ritardo relativo al caso peggiore.

4.47 [15] <§4.5> Si ripeta l'Esercizio 4.46, calcolando questa volta le velocità relative tra un sommatore di 16 bit basato sulla sola propagazione del riporto, un som-

matore a propagazione del riporto tra gruppi di 4 bit che usano il carry lookahead, ed un sommatore che adotta lo schema di carry lookahead del Paragrafo 4.5.

4.48 [15] <§4.5> {Es. 4.45} Si ripeta l'Esercizio 4.46, calcolando questa volta le velocità relative di un sommatore a 64 bit che utilizza la sola propagazione del riporto, un sommatore che usa la propagazione del riporto tra gruppi di 4 bit che usano il carry lookahead, un sommatore che usa la propagazione del riporto tra gruppi di 16 bit che usano il carry lookahead, ed un sommatore che adotta lo schema di carry lookahead dell'Esercizio 4.45.

4.49 [10] <§4.5> Vi sono occasioni in cui si desidera sommare insieme una serie di numeri. Si supponga di voler sommare quattro numeri di 4 bit (A, B, E, F) utilizzando dei full adder di 1 bit: si ignori per ora il carry lookahead. Probabilmente si tenderà a collegare i sommatori di 1 bit secondo l'organizzazione riportata nella parte alta della figura 4.56. Nella parte bassa vi è una nuova organizzazione dei full



ra 4.56 Somma tradizionale tramite propagazione del riporto e tramite carry save di quattro numeri di 4 bit **bit.** I dettagli sono mostrati sulla sinistra ed i singoli segnali sono riportati in minuscolo, mentre i blocchi corrispondenti di alto livello sono a destra, con i segnali complessivi in maiuscolo. Si noti che la somma di 4 numeri di n bit può richiedere n+2 bit.

adder: si provi a sommare quattro numeri usando entrambe le organizzazioni per convincersi del fatto che si ottiene lo stesso risultato.

4.50 [5] <§4.5> {Es. 4.49} Si assuma che il ritardo relativo a ciascun sommatore di 1 bit sia 2T. Si calcoli il tempo necessario per sommare quattro numeri su 4 bit nell'organizzazione riportata in alto rispetto a quella riportata in basso nella figura 4.56.

In maggior dettaglio

I sommatori carry save

Gli Esercizi 4.49 e 4.50 rendono ragione di una organizzazione che utilizza il sommatore di 1 bit della figura 4.10 in una maniera che non era stata illustrata sinora. Benché questo componente hardware sia semplice e veloce, il problema nasce dal tentativo di calcolare il segnale di CarryIn in maniera rapida da parte di diversi sommatori.

Si può pensare al sommatore come ad un dispositivo in grado di sommare insieme tre ingressi (a_i, b_i, c_i) e di produrre due uscite (s, c_{i+1}), Quando si devono soltanto sommare due numeri, questa osservazione è di ben poca utilità, ma quando si stanno sommando più di due operandi è possibile ridurre il costo del riporto. L'idea è quella di formare due somme indipendenti, chiamate S' (bit di somma) e C' (bit di riporto); alla fine del processo, è necessario sommare C' ed S' insieme utilizzando un normale sommatore. Questa tecnica, basata sul fatto di ritardare la propagazione del riporto sino alla fine della somma dei numeri, è chiamata *carry save addition*. Lo schema a blocchi in basso a destra nella figura 4.56 mostra l'organizzazione risultante, che comprende due livelli di sommatori carry save collegati da un singolo sommatore di tipo normale.

4.51 [10] <§4.5> {Es. 4.47, 4.50} Si calcoli il ritardo necessario per sommare quattro numeri di 16 bit utilizzando un sommatore completo di tipo carry lookahead rispetto ad un sommatore di tipo carry save con un sommatore di tipo carry lookahead che esegue la somma conclusiva. L'unità di tempo T degli Esercizi 4.46 e 4.50 è la stessa.

4.52 [20] <§4.5, 4.6> {Es. 4.47} Il caso forse più frequente in cui si devono sommare insieme molti numeri in un calcolatore è quando si desidera moltiplicare con maggiore rapidità, utilizzando più sommatori per sommare molti numeri in un singolo ciclo di clock. Se confrontato con l'algoritmo di moltiplicazione della figura 4.32, uno schema basato su carry save con diversi sommatori è in grado di eseguire la moltiplicazione in maniera più veloce di oltre 10 volte.

L'esercizio consiste nel valutare il costo e la velocità di un moltiplicatore combinatorio per la moltiplicazione di due numeri positivi su 16 bit. Si assuma di avere 16 termini intermedi M15, M14, ... , M0, chiamati *prodotti parziali*, che contengono il moltiplicando in AND con i bit del moltiplicatore m15, m14, ... , m0.

L'idea consiste nell'utilizzare i sommatori carry save per ridurre gli n operandi a $2/3n$ in gruppi paralleli di 3, e nel ripetere tale operazione sino a che si ottengono due grandi numeri da sommare tra loro con un sommatore tradizionale.

Innanzitutto si mostri lo schema a blocchi dei sommatori carry save di 16 bit per sommare i 16 termini, come mostrato a destra nella figura 4.56, e poi si calcolino i ritardi nella somma dei 16 numeri. Si confrontino i tempi ottenuti con lo schema di moltiplicazione iterativa della figura 4.32, ma si assumano soltanto 16 iterazioni utilizzando un sommatore di 16 bit che sfrutta un carry lookahead completo, la cui velocità è stata calcolata nell'Esercizio 4.47.

4.53 [30] <§4.6> L'idea originale alla base dell'algoritmo di Booth stava nella riduzione del numero di operazioni evitando di eseguire un'operazione quando vi era una stringa di 0 o 1. Si riconsideri l'algoritmo del Paragrafo 4.6, trattando 3 bit alla volta e calcolando il moltiplicando 2 bit alla volta. Si riempia la seguente tabella per determinare la codifica di Booth a 2 bit:

Bit attuali		Bit precedente	Operazione	Motivo
ai + 1	*ai*	*ai* – 1		
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Si assuma di poter disporre nei registri sia del moltiplicando che dello stesso moltiplicato per due. Si illustri la ragione dell'operazione svolta ad ogni passo, e si mostri un esempio su 6 bit che possa essere eseguito più rapidamente usando questo algoritmo. Suggerimento: si provi la strategia divide et impera; si verifichi quali sarebbero le operazioni da eseguire in ciascuno degli otto casi della tabella usando un algoritmo di Booth a 2 bit, e poi si ottimizzi la coppia di operazioni.

4.54 [30] <§§4.6, 4.7> L'algoritmo di divisione della figura 4.40 è detto divisione *con ripristino*, dal momento che ogni volta che il risultato della sottrazione del divisore dal dividendo è negativo si deve ri-sommare il divisore al dividendo per ripristinare il valore originario. Si ricordi che lo scalamento a sinistra coincide con la moltiplicazione per due. Si dia ancora un'occhiata al valore nella metà sinistra del Resto, a partire dal passo 3b dell'algoritmo di divisione e poi andando al passo 2:

$$(\text{Resto}+\text{Divisore})\times 2-\text{Divisore}$$

Tale valore viene creato tramite il ripristino del Resto mediante la somma del Divisore, scalando la somma a sinistra, e poi sottraendo il Divisore. Semplificando il risultato si ottiene:

$$\text{Resto}\times 2+\text{Divisore}\times 2-\text{Divisore}=\text{Resto}\times 2+\text{Divisore}$$

Sulla base di questa osservazione si scriva un algoritmo di divisione *senza ripristino* che utilizzi la notazione della figura 4.40, il quale non esegua la somma del Divisore al Resto nel passo 3b. Si mostri che l'algoritmo funziona dividendo $0000\ 0111_{due}$ per 0010_{due}.

4.55 [5] <§4.8> Si sommi $6{,}42_{dieci}\times 10^1$ a $9{,}51_{dieci}\times 10^2$, assumendo di disporre di tre sole cifre significative, prima con cifre di guardia e di arrotondamento e poi senza.

4.56 [5] <§4.8> Si ripeta l'Esercizio 4.55, questa volta utilizzando i numeri $8{,}76_{dieci}\times 10^1$ e $1{,}47_{dieci}\times 10^2$.

4.57 [25] <§4.8> Si deduca l'algoritmo per la divisione in virgola mobile come si è fatto per la somma e la moltiplicazione nel Paragrafo 4.8. Prima si divida $1{,}110_{dieci}\times 10^{10}$ per $1{,}100_{dieci}\times 10^{-5}$, mostrando gli stessi passi eseguiti allora, e poi si deduca l'algoritmo di divisione in virgola mobile utilizzando un formato simile a quello dell'algoritmo di moltiplicazione della figura 4.46.

4.58 [30] <§4.8> L'approfondimento al fondo del Paragrafo 4.8 illustra le quattro modalità di arrotondamento previste dallo standard IEEE 754, nonché lo *sticky bit*, necessario in aggiunta ai 2 bit chiamati *guardia* ed *arrotondamento*. Quello di guardia è il primo bit, mentre quello di arrotondamento è il secondo, e lo sticky bit mantiene l'informazione relativa al fatto che i bit rimanenti siano 0 o no. Si riempia la seguente tabella con le equazioni logiche funzioni di guardia (g), arrotondamento (r) e sticky bit (s) per il risultato della somma in virgola mobile che crea Somma. Sia p il numero appropriato di bit nella mantissa per una certa precisione, e Somma_p sia il p-simo bit più significativo di Somma. Una casella vuota indica che i p bit più significativi della somma sono stati correttamente arrotondati. Se si scrive un'equazione nella casella, un'equazione falsa significa che i p bit sono stati correttamente arrotondati; un'equazione vera significa che si deve sommare 1 al p-esimo bit più significativo di Somma.

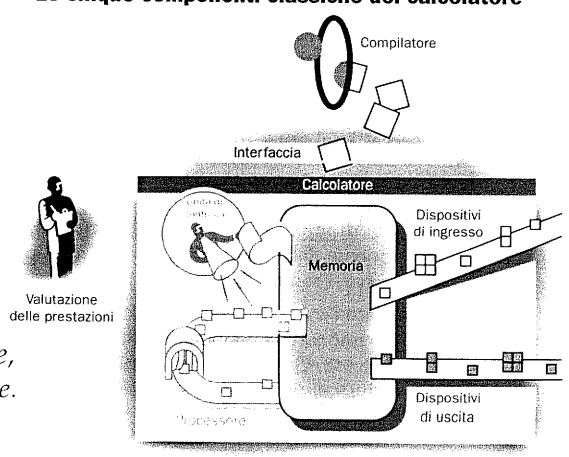
Modalità di arrotondamento	Somma>>0	Somma<0
Verso $-\infty$		
Verso $+\infty$		
Troncamento		
Pari più vicino		

4.59 [30] <§4.8>L'approfondimento al fondo del Paragrafo 4.8 cita il fatto che lo IEEE 754 prevede due simboli speciali che possono essere operandi in virgola mobile: infinito e Non-Numero (NaN). Vi sono anche dei numeri piccoli denominati *denorm*, che non sono normalizzati. Poiché questi numeri e simboli speciali non sono usati molto spesso, si usano a volte delle implementazioni che impiegano una combinazione di tecniche hardware e software. Per esempio, anziché usare dei circuiti complessi per gestire questi casi speciali, si genera un'eccezione ed essi sono gestiti via software. Esistono molte opzioni implementative e ciascuna di queste ha le proprie caratteristiche peculiari in termini di prestazioni. In questo esercizio si chiede di confrontare diversi elaboratori dal punto di vista delle operazioni in virgola mobile, nel caso in cui gli operandi siano sia numeri normali che tali valori particolari. Nell'enunciare le conclusioni ci si accerti di confrontare le prestazioni dei diversi elaboratori l'uno con l'altro, e di descrivere le relative somiglianze e diversità. Quale impatto è probabile che tali conclusioni possano avere sui progettisti del software che devono decidere se fare uso o meno di queste particolarità dello standard IEEE 754?

4.60 [30] <§4.5> Se si ha accesso ad un calcolatore che contiene un processore MIPS, si scriva un ciclo in linguaggio assembler che forzi i registri `$k0` (`$26`) e `$k1` (`$27`) ad un valore iniziale, e poi iteri per qualche secondo controllando il valore di questi registri. Se ne stampi il valore se questi cambiano. Si veda l'approfondimento al fondo del Paragrafo 4.3 per avere una spiegazione del perché cambiano. Si è in grado di trovare una spiegazione dei valori osservati?

Il processore: unità di elaborazione dati e unità di controllo

Le cinque componenti classiche del calcolatore



Se una questione è importante, nessun dettaglio è trascurabile.

Proverbio francese

5.1 Introduzione

Nel Capitolo 2 si è visto che le prestazioni di un sistema sono determinate da tre fattori chiave: il numero di istruzioni, il periodo del clock ed il numero di cicli di clock per istruzione (CPI). Il compilatore e l'architettura del set di istruzioni, esaminati nei Capitoli 3 e 4, determinano il numero di istruzioni necessarie ad eseguire un certo programma, mentre sia il periodo del clock che il numero di cicli di clock per istruzione dipendono dall'implementazione del processore.

In questo capitolo si illustrerà il progetto delle unità di elaborazione dati e di controllo per due diverse implementazioni del set di istruzioni MIPS; in particolare si progetterà un'implementazione di un sottoinsieme del set di istruzioni base del MIPS composto da:

- Istruzioni di accesso alla memoria load word (`lw`) e store word (`sw`).
- Istruzioni logico-aritmetiche `add`, `sub`, `and`, `or` e `slt`.
- Istruzioni di salto branch equal (`beq`) e jump (`j`), che verranno aggiunte per ultime.

Questo sottoinsieme non comprende tutte le istruzioni intere (ad esempio mancano moltiplicazioni e divisioni) ed ignora le istruzioni in virgola mobile; tuttavia sarà sufficiente ad illustrare i principi basilari per la realizzazione dell'unità di elaborazione e per il progetto dell'unità di controllo: l'implementazione delle istruzioni rimanenti sarebbe del tutto simile.

Nell'esaminare questa realizzazione si avrà l'opportunità di vedere come l'architettura del set di istruzioni determini molti aspetti dell'implementazione e come la scelta di diverse strategie di implementazione abbia impatto sulla frequenza di clock e sul CPI della macchina. Molti principi progettuali introdotti nel Capitolo 3, come le linee guida *Rendere veloce l'evento più frequente* e *Semplicità e regolarità sono correlate*, saranno illustrati con un'analisi dell'implementazione; inoltre, molti dei concetti utilizzati in questo capitolo e nel prossimo per realizzare l'implementazione del sottoinsieme di istruzioni MIPS sono applicabili al progetto di un ampio spettro di elaboratori, dai sistemi ad alte prestazioni ai microprocessori general-purpose fino ai processori special-purpose, che sono utilizzati sempre più frequentemente in prodotti che vanno dai videoregistratori alle automobili.

Quadro d'insieme dell'implementazione

Nei Capitoli 3 e 4 sono state esaminate le istruzioni MIPS base, comprese quelle logico-aritmetiche su numeri interi, quelle di accesso alla memoria e quelle di salto: la modalità di implementazione di ciascuna di esse è in larga misura simile, indipendentemente dalla classe di istruzioni di appartenenza. Per ciascuna istruzione i primi due passi sono identici:

1. Inviare l'uscita del contatore di programma (program counter, PC) all'unità di memoria che contiene il programma e prelevare l'istruzione dalla memoria stessa. Questa operazione è indicata in inglese con il termine *fetch*, che significa appunto prelevare.

2. Leggere uno o due registri, selezionati mediante i campi dell'istruzione. Per l'istruzione load word è sufficiente un solo registro, ma la maggior parte delle altre istruzioni richiede la lettura di due registri.

Compiuti questi due passi, le azioni necessarie al completamento dell'istruzione dipendono dalla sua classe; fortunatamente all'interno della stessa classe (accesso alla memoria, logico-aritmetica, salto) le azioni sono in larga misura le stesse, indipendentemente dall'esatto codice operativo.

Si possono riscontrare delle somiglianze anche tra diverse classi di istruzioni; per esempio, tutte le classi di istruzioni utilizzano l'unità logico-aritmetica (ALU) dopo aver letto i registri: le istruzioni di accesso alla memoria usano la ALU per il calcolo dell'indirizzo, le istruzioni logico-aritmetiche per l'esecuzione dell'operazione ed i salti per eseguire i confronti. Come si vede facilmente, la semplicità e la regolarità del set di istruzioni semplificano l'implementazione, rendendo simile l'esecuzione di istruzioni appartenenti a classi diverse.

Dopo l'utilizzo della ALU, le azioni richieste per completare le istruzioni differiscono per le diverse classi: un'istruzione di accesso alla memoria dovrà accedervi per scrivere il dato (store) o per leggerlo (load); un'istruzione logico-aritmetica dovrà scrivere il risultato calcolato dalla ALU in un registro; le istruzioni di salto, infine, potrebbero dover modificare l'indirizzo della prossima istruzione da eseguire, in funzione dell'esito del confronto.

La figura 5.1 riporta una vista ad alto livello di un'implementazione del MIPS; nel resto di questo capitolo tale schema verrà raffinato specificandone i dettagli, il che richiederà l'aggiunta di ulteriori unità funzionali, l'aumento del numero di connessioni tra le unità, ed ovviamente l'inserimento dell'unità di controllo per stabilire le azioni che devono essere intraprese per le istruzioni delle diverse classi. Prima di iniziare a creare un'implementazione più completa, è conveniente rivedere alcuni principi sul progetto delle reti logiche.

Convenzioni logiche e temporizzazione

Nel progettare un sistema, si devono decidere le modalità di funzionamento dell'implementazione logica e di temporizzazione del clock: questo paragrafo riprende alcune idee fondamentali sulla logica digitale, che verranno utilizzate pesantemente in tutto il capitolo. Il lettore che non abbia una buona preparazione nel campo della progettazione dei circuiti digitali troverà utile studiare l'Appendice B prima di proseguire la lettura; in particolare, il Paragrafo B.9 presenta le parole chiave introdotte nell'Appendice B e può essere utilizzato come una veloce verifica delle proprie conoscenze sulla progettazione digitale.

Nel progetto dei circuiti logici è spesso conveniente modificare la corrispondenza tra un segnale vero o falso dal punto di vista logico ed i livelli di tensione alti e bassi: in alcune parti del progetto un segnale affermato dal punto di vista logico potrebbe avere un livello elettrico basso, mentre in altre parti i segnali elettricamente alti sono considerati affermati. Per coerenza, verrà utilizzato il termine *affermato* per indicare un segnale vero dal punto di vista logico, ed il verbo *affermare* per specificare che un segnale deve essere forzato al livello logico vero.

Le unità funzionali nell'implementazione del MIPS consistono di due tipi diversi di elementi logici: elementi che operano sui dati ed elementi che memorizzano lo stato. Gli elementi che operano sui valori dei dati sono di tipo *combinatorio*, il che significa che le loro uscite dipendono solamente dagli ingressi nello stesso istante: dati gli stessi valori di ingresso, un elemento combinatorio produrrà sempre lo

stesso valore di uscita. La ALU mostrata in figura 5.1 e discussa in dettaglio nel Capitolo 4 è un elemento combinatorio: dato un insieme di ingressi produce sempre la stessa uscita perché non contiene alcun elemento di memoria.

Altri elementi del progetto non sono combinatori, bensì memorizzano lo *stato*: un elemento può memorizzare informazioni di stato solo se contiene al suo interno degli elementi di memoria. Tali elementi verranno detti *elementi di stato*: infatti, pur staccando la spina al calcolatore, sarebbe possibile farlo ripartire dallo stesso punto caricando negli elementi di stato i valori che contenevano prima dell'incidente; inoltre, salvando e ripristinando gli elementi di stato, il calcolatore si comporterebbe come se non avesse mai perso l'alimentazione. Di conseguenza gli elementi di stato caratterizzano completamente il sistema; nella figura 5.1 le memorie di programma e di dato ed i registri sono esempi di elementi di stato.

Un elemento di stato possiede almeno due ingressi ed un'uscita: gli ingressi sono il valore da scrivere nell'elemento ed il clock, che determina l'istante in cui il valore viene scritto; l'uscita fornisce il valore che era stato scritto in un ciclo di clock precedente. Per esempio, uno dei più semplici elementi di stato è il flip-flop di tipo D (si veda d'Appendice B), che ha esattamente due ingressi (il valore ed il clock) ed un'uscita. Oltre ai flip-flop, l'implementazione del MIPS appena illustrata in figura 5.1 utilizza altri due tipi di elementi di stato: le memorie ed i registri. Il clock viene utilizzato per determinare l'istante in cui avviene l'operazione di scrittura, mentre la lettura può avvenire in ogni istante.

I componenti logici che contengono uno stato sono anche detti *sequenziali* in quanto la loro uscita dipende sia dagli ingressi che dal valore dello stato interno. Per esempio, l'uscita dell'unità funzionale che rappresenta i registri dipende sia dal numero del registro specificato sugli ingressi che dal valore che era stato scritto in precedenza in quel registro. L'Appendice B discute in maggior dettaglio il comportamento e la struttura degli elementi combinatori e sequenziali.

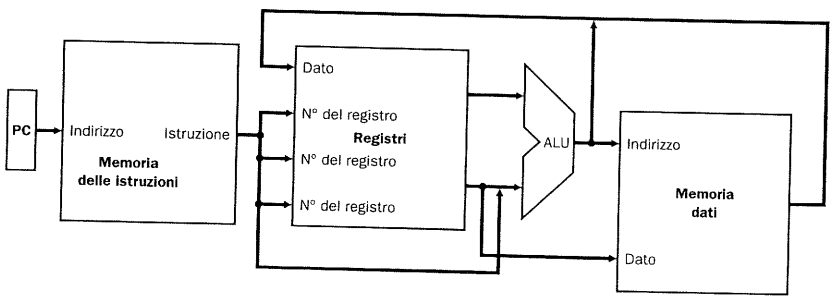


Figura 5.1 Una visione astratta dell'implementazione del sottoinsieme di istruzioni MIPS contenente le principali unità funzionali e le principali interconnessioni. Tutte le istruzioni iniziano utilizzando il program counter per fornire alla memoria l'indirizzo dell'istruzione; dopo aver letto l'istruzione, i registri da utilizzare come operandi sono ricavati dai campi dell'istruzione. Una volta recuperati gli operandi dai registri, se ne potranno elaborare i valori per calcolare un indirizzo di memoria (per load e store), un risultato aritmetico (per istruzioni logico-aritmetiche intere) o un confronto (per i salti). Se l'istruzione è logico-aritmetica, il risultato calcolato dalla ALU deve essere scritto in un registro; se è una load o store, il valore della ALU viene utilizzato come indirizzo per memorizzare il valore di un registro o per trasferire un valore dalla memoria in un registro; il risultato ottenuto dalla ALU o dalla memoria è poi scritto all'interno del register file. I salti richiederanno l'utilizzo della ALU per il calcolo dell'indirizzo della prossima istruzione; come si vedrà, questo richiederà della logica di controllo aggiuntiva.

Metodologia di temporizzazione

La *metodologia di temporizzazione* definisce quando i segnali possono essere scritti e quando possono essere letti: è importante definire la temporizzazione delle operazioni di lettura e scrittura perché se un segnale venisse letto contemporaneamente alla sua scrittura, il valore letto potrebbe essere quello precedente, quello appena scritto, o addirittura una combinazione dei due! Ovviamente i progetti dei calcolatori non possono tollerare questa imprevedibilità: la metodologia di temporizzazione viene definita in modo da prevenire il verificarsi di circostanze di questo tipo.

Per semplicità si assumerà di utilizzare una metodologia di temporizzazione di tipo *sensibile ai fronti* (*edge-triggered*), ossia i valori memorizzati all'interno del sistema sono modificati solamente in corrispondenza di un fronte del segnale di clock: tutti gli elementi di stato aggiornano il proprio valore in corrispondenza di un fronte del clock. Poiché solo gli elementi di stato possono memorizzare un valore, le reti combinatorie devono ricevere i propri ingressi da un insieme di elementi di stato e scrivere il valore delle proprie uscite in un insieme di elementi di stato: gli ingressi sono i valori scritti in un ciclo di clock precedente, mentre i valori di uscita potranno essere utilizzati in un ciclo successivo.

La figura 5.2 mostra due elementi di stato ai lati di un blocco composto da logica combinatoria, che lavora in un solo ciclo di clock: i segnali si devono propagare a partire dall'elemento di stato 1, attraversare la logica combinatoria e raggiungere l'elemento di stato 2 nel lasso di tempo di un periodo di clock. Il tempo necessario ai segnali per raggiungere l'elemento di stato 2 definisce quindi la durata minima del periodo di clock.

Per semplicità, nel seguito non verrà riportato esplicitamente alcun segnale di controllo dell'operazione di scrittura per gli elementi di stato che vengono scritti ad ogni fronte attivo del clock, mentre, se un elemento di stato non viene aggiornato ad ogni ciclo, è necessario un segnale esplicito di controllo della scrittura. Sia il segnale di clock che il segnale di controllo della scrittura sono ingressi: l'elemento di stato viene modificato solamente quando il segnale di controllo della scrittura è affermato e si verifica un fronte attivo del clock.

La metodologia sensibile ai fronti permette di leggere il contenuto di un registro, inviarne il valore attraverso uno o più blocchi di logica combinatoria e scrivere lo

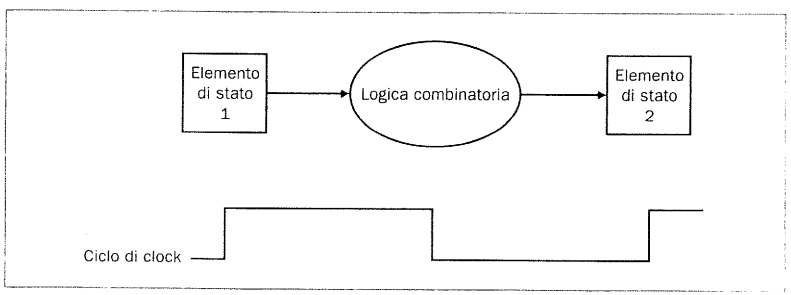


Figura 5.2 Logica combinatoria, elementi di stato e clock sono interdipendenti. In un sistema digitale sincrono il clock determina quando gli elementi di stato debbono scrivere i valori di ingresso nella loro memoria interna. Gli ingressi di un elemento di stato devono raggiungere uno stato stabile (ossia, devono aver raggiunto un valore destinato a non cambiare almeno fino al prossimo fronte del clock) prima che il fronte attivo del clock causi un aggiornamento dello stato; si assumerà che tutti gli elementi di stato, incluse le memorie, siano sensibili ai fronti.

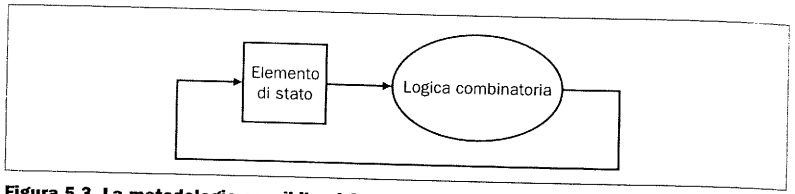


Figura 5.3 La metodologia sensibile ai fronti permette di leggere e scrivere un elemento di stato nello stesso ciclo di clock senza creare una condizione critica che potrebbe causare indeterminatezza nei valori dei dati. Il ciclo di clock deve avere una durata sufficiente a garantire che gli ingressi siano stabili al verificarsi del fronte attivo del clock. Non può verificarsi alcuna reazione ciclica del segnale in un solo ciclo di clock grazie all'aggiornamento sensibile ai fronti dell'elemento di stato: se fosse possibile un tale ritorno di segnale questo circuito non funzionerebbe correttamente. I progetti presentati in questo capitolo e nel prossimo si basano sulla metodologia di temporizzazione sensibile ai fronti e su strutture simili a quella indicata in figura.

stesso registro nel medesimo ciclo di clock, come mostrato in figura 5.3. Visto che gli ingressi del blocco di logica combinatoria non possono cambiare se non in corrispondenza del fronte stabilito, è indifferente assumere che le scritture avvengano sul fronte di salita del clock o su quello di discesa; con la metodologia di temporizzazione sensibile ai fronti *non* vi è rischio di reazione ciclica all'interno dello stesso periodo di clock, ed il circuito di figura 5.3 può funzionare correttamente. L'Appendice B discuterà brevemente alcuni vincoli temporali aggiuntivi (come i tempi di set-up e di hold) e presenterà altre metodologie di temporizzazione.

La maggior parte degli elementi di stato e delle reti logiche possiede ingressi ed uscite di ampiezza pari a 32 bit, essendo questa l'ampiezza di quasi tutti i dati trattati dal processore; verrà indicato chiaramente quando qualche unità abbia ingressi o uscite di ampiezza diversa da 32. Nelle figure si indicheranno con delle linee più spesse i *bus*, ossia i segnali di ampiezza maggiore di 1 bit. In alcuni casi si dovranno unire due o più bus per formare un bus risultante di ampiezza maggiore, ad esempio per ottenere un bus di 32 bit a partire da due bus di 16 bit; tale operazione verrà indicata chiaramente mediante le etichette sui bus stessi. Saranno presenti delle frecce per chiarire la direzione del flusso di dati tra gli elementi, ed un diverso colore identificherà i segnali di controllo rispetto ai segnali di dato, distinzione che diverrà più chiara nel seguito di questo capitolo.

L'implementazione del sottoinsieme MIPS

Verrà presentata inizialmente un'implementazione semplificata (detta implementazione elementare) che utilizzerà un solo ciclo di clock, di ampia durata, per ciascuna istruzione e che seguirà lo schema generale riportato in figura 5.1. In questo primo progetto l'esecuzione di ogni istruzione inizia in corrispondenza di un fronte del clock e viene completata entro il fronte successivo.

Sebbene più semplice da comprendere, questo approccio non è conveniente, poiché più lento di un'implementazione che esegua classi di istruzioni diverse in un diverso numero di cicli di clock e disponga di un clock decisamente più veloce. In seguito verrà quindi esaminata una seconda implementazione che utilizza più cicli di clock per istruzione, più realistica ma più complessa dal punto di vista della logica di controllo.

In questo capitolo la logica di controllo verrà specificata in termini di equazioni logiche o macchine a stati finiti: da queste rappresentazioni si può sintetizzare un'implementazione hardware grazie ai moderni programmi di progettazione assistita da calcolatore (CAD, computer-aided design), come illustrato nell'Appendice C. Verso la fine del capitolo si tratterà l'implementazione delle eccezioni, argomento già citato nel Capitolo 4.

5.2 Progetto dell'unità di elaborazione dati

Un metodo ragionevole per iniziare il progetto dell'unità di elaborazione consiste nell'esaminare i componenti principali richiesti per l'esecuzione di ciascuna classe di istruzioni MIPS; si può procedere analizzando quali elementi dell'unità di elaborazione sono necessari a ciascuna istruzione ed utilizzarli costruendo un frammento dell'unità di elaborazione per ciascuna classe di istruzioni. Gli elementi dell'unità di elaborazione saranno mostrati insieme ai relativi segnali di controllo.

Il primo elemento necessario è un componente in cui memorizzare le istruzioni del programma: si utilizza un'unità di memoria, che come detto è un elemento di stato, per ricordare le istruzioni e fornirle in uscita a fronte di un indirizzo in ingresso, come mostrato in figura 5.4. L'indirizzo dell'istruzione deve a sua volta essere memorizzato in un elemento di stato, che verrà chiamato *contatore di programma* (o *program counter*, PC), anch'esso indicato in figura 5.4. È infine necessario un sommatore per incrementare PC ed ottenere l'indirizzo dell'istruzione successiva: esso è un elemento combinatorio che si può ottenere a partire dalla ALU progettata nel capitolo precedente semplicemente collegando i segnali di controllo in modo da specificare sempre un'operazione di somma. Ad una ALU collegata in tal modo verrà attribuita l'etichetta *Somma* come in figura 5.4, per indicare che è stata permanentemente convertita in un sommatore e non può svolgere le altre funzioni di una ALU.

Per eseguire una qualsiasi istruzione occorre innanzitutto prelevare l'istruzione stessa dalla memoria; per prepararsi ad eseguire la prossima istruzione, occorre inoltre incrementare il program counter in modo che punti all'istruzione successi-

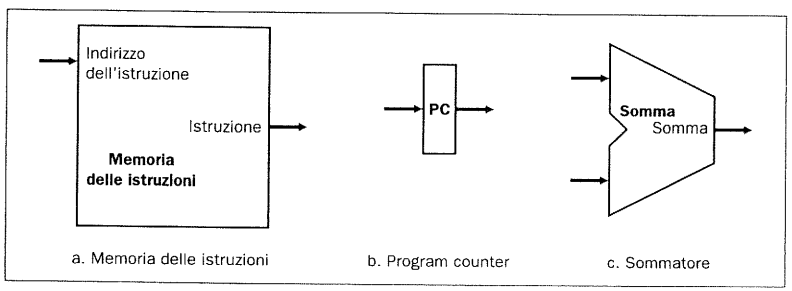


Figura 5.4 Sono richiesti due elementi di stato per memorizzare ed accedere alle istruzioni ed un sommatore per calcolare l'indirizzo dell'istruzione successiva. Gli elementi di stato sono la memoria delle istruzioni ed il program counter. La memoria delle istruzioni fornisce solamente un accesso in lettura, dal momento che l'unità di elaborazione non scrive nuove istruzioni; essa può quindi essere trattata come un blocco logico combinatorio. L'uscita in ogni istante corrisponde al contenuto della locazione specificata dall'ingresso di indirizzo, e non è necessario alcun segnale di controllo dell'operazione di lettura, semplificando così il progetto (la memoria delle istruzioni dovrà essere scritta solamente nella fase di caricamento del programma, ma questa estensione non è difficile da integrare e verrà inizialmente ignorata per semplicità). Il program counter è un registro a 32 bit che viene scritto al termine di ogni ciclo di clock e non richiede quindi un segnale di controllo della scrittura. Il sommatore è una ALU connessa in modo da calcolare sempre la somma dei suoi due ingressi a 32 bit e da fornire in uscita il risultato.

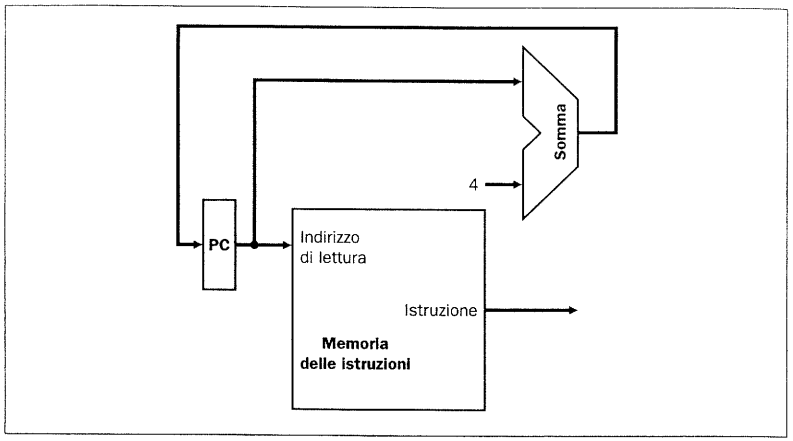


Figura 5.5 Una porzione dell'unità di elaborazione utilizzata per prelevare le istruzioni ed incrementare il program counter. L'istruzione letta è utilizzata dalle altre parti dell'unità di elaborazione.

va, che si trova 4 byte più avanti. La parte unità di elaborazione che gestisce queste operazioni, mostrata in figura 5.5, usa i tre elementi presentati in figura 5.4.

Si considerino ora le istruzioni di formato-R (si veda la figura 3.19): tutte leggono due registri, eseguono un'operazione con la ALU sul contenuto di tali registri ed infine scrivono il risultato: sono dette *istruzioni di tipo-R* o *istruzioni logico-aritmetiche* poiché realizzano operazioni aritmetiche o logiche. Questa classe di istruzioni include `add`, `sub` e `slt`, introdotte nel Capitolo 3, oltre a `and` e `or`, introdotte nel Capitolo 4. Si ricordi che un tipico esempio di tali istruzioni è `add $t1, $t2, $t3`, che legge `$t2` e `$t3` e scrive `$t1`.

I registri a 32 bit del processore sono raccolti in una struttura detta *register file* (letteralmente: archivio dei registri), che è un insieme di registri in cui ciascun registro può essere letto o scritto specificando il numero d'ordine del registro all'interno dell'insieme. Il register file contiene lo stato dei registri della macchina. È inoltre necessaria una ALU per operare sui valori letti dai registri.

Le istruzioni di formato-R hanno tre registri come operandi, per cui in ciascuna istruzione si dovranno leggere due parole di dato dal register file e poi scriverne una. Per ogni parola letta dai registri è necessario prevedere un ingresso al register file che specifichi il numero del registro da leggere ed un'uscita del register file sulla quale verrà riportato il valore letto dal registro. Per scrivere la parola di dato serviranno due ingressi: il primo per specificare il *numero del registro* da scrivere ed il secondo per fornire il *dato* da scrivere nel registro. Il register file fornisce in qualunque momento in uscita il contenuto dei registri i cui numeri sono specificati sugli ingressi "Registro letto", mentre le scritture sono controllate da un segnale di controllo, che deve essere affermato in corrispondenza del fronte del clock affinché la scrittura avvenga. Complessivamente serviranno quattro ingressi (tre per i numeri dei registri ed uno per il dato) e due uscite (entrambe per i dati), come indicato in figura 5.6. Gli ingressi che specificano il numero dei registri sono ampi 5 bit e specificano uno dei 32 registri ($32=2^5$), mentre i bus degli ingressi e delle uscite di dato sono di 32 bit ciascuno.

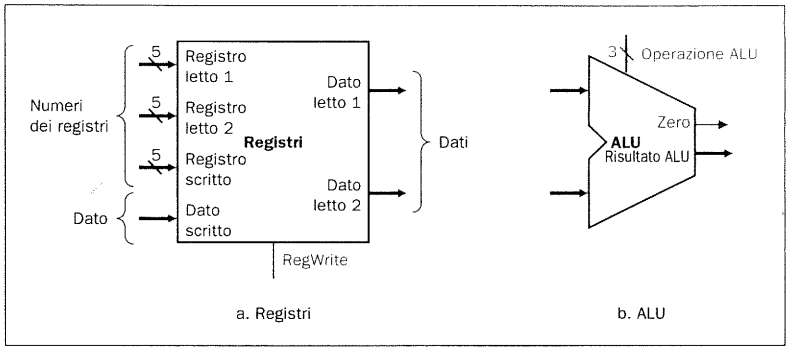


Figura 5.6 I due elementi richiesti per implementare le operazioni di formato-R sono il register file e la ALU. Il register file contiene tutti i registri e dispone di due porte di lettura e una porta di scrittura; il progetto di register file a più porte è discusso nel Paragrafo B.5 dell'Appendice B. In ogni istante, il register file fornisce in uscita il contenuto dei registri corrispondenti agli ingressi "Registro letto" senza richiedere alcun segnale di controllo; viceversa, la scrittura di un registro deve essere indicata esplicitamente affermando un segnale di controllo. Si ricordi che le scritture sono attive sul fronte, per cui tutti gli ingressi interessati (cioè il valore da scrivere, il numero di registro ed il segnale di controllo) devono essere validi al verificarsi del fronte del clock. Essendo le scritture attive sui fronti, il progetto può tranquillamente leggere e scrivere lo stesso registro nello stesso ciclo di clock: la lettura fornirà il valore scritto in un ciclo di clock precedente, mentre il valore scritto potrà essere letto a partire dal ciclo di clock successivo. Gli ingressi che specificano al register file i numeri dei registri hanno ampiezza pari a 5 bit, mentre le linee che trasportano i dati sono di 32 bit. L'operazione compiuta dalla ALU è controllata dal segnale "Operazione ALU", ampio 3 bit, utilizzando la ALU progettata nel capitolo precedente (figura 4.19). L'uscita Zero di rilevamento dello zero della ALU verrà utilizzata tra breve per implementare i salti; l'uscita di overflow verrà utilizzata solo a partire dal Paragrafo 5.6, in cui verranno discusse le eccezioni.

La ALU mostrata in figura 5.6 è controllata dal segnale di 3 bit descritto nel Capitolo 4 e riceve due ingressi di 32 bit producendo un risultato su 32 bit.

L'unità di elaborazione per le istruzioni di tipo-R, che utilizza il register file e la ALU della figura 5.6, è rappresentata in figura 5.7: dato che i numeri dei registri provengono dai campi dell'istruzione, un segnale contenente l'istruzione e proveniente dal circuito di figura 5.5 è collegato agli ingressi opportuni del register file.

Si considerino a questo punto le istruzioni MIPS load word e store word, che hanno la forma generale `lw $t1, valore_offset($t2)` e `sw $t1, valo-`

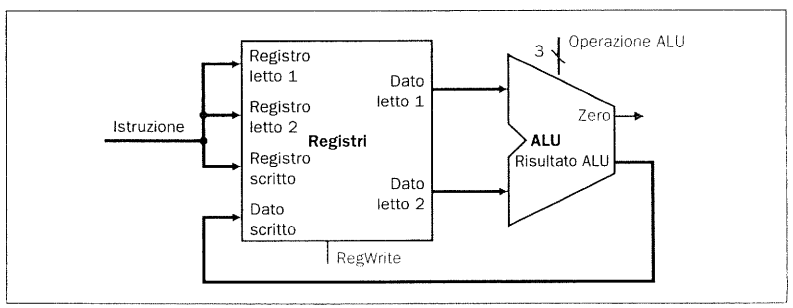


Figura 5.7 L'unità di elaborazione per le istruzioni di tipo-R. La ALU discussa nel Capitolo 4 può essere controllata in modo da fornire tutte le funzioni elementari richieste dalle istruzioni di tipo-R.

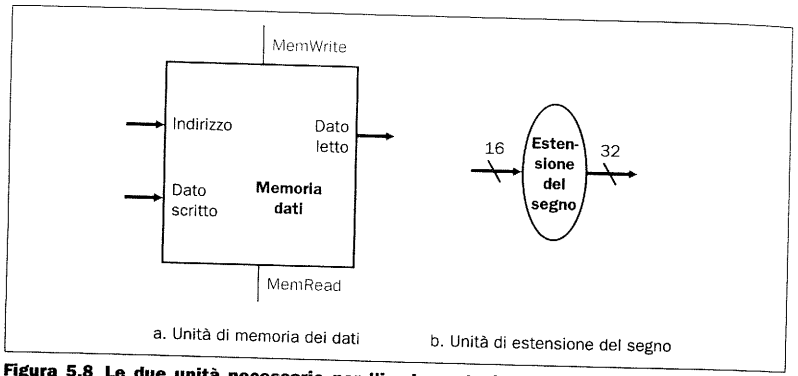


Figura 5.8 Le due unità necessarie per l'implementazione delle istruzioni di load e di store, oltre al register file e alla ALU di figura 5.6, sono l'unità di memoria dati e l'unità di estensione del segno. L'unità di memoria è un elemento di stato con due ingressi, per l'indirizzo ed i dati da scrivere, ed una sola uscita per il risultato della lettura; vi sono segnali di controllo separati per la lettura e la scrittura, sebbene in ogni ciclo di clock uno solo di essi possa essere affermato. L'unità di estensione del segno ha un ingresso a 16 bit che in uscita viene esteso ad un numero di 32 bit (come già indicato nel Capitolo 4). Per quanto riguarda le operazioni di scrittura si farà l'assunzione che l'unità di memoria sia attiva sui fronti. Sebbene il segnale di abilitazione alla scrittura nei chip di memoria reali non sia sensibile ai fronti, questo progetto può essere facilmente adattato per funzionare anche in quel caso: per una discussione più approfondita sul funzionamento dei chip di memoria reali si veda il Paragrafo B.5 dell'Appendice B.

`re_offset($t2)`: tali istruzioni calcolano un indirizzo di memoria sommando il registro base, in questo caso `$t2`, al campo offset di 16 bit con segno contenuto nell'istruzione. Se l'istruzione è una store il valore da memorizzare deve essere letto dal register file, dove risiede in `$t1`; se è una load il valore letto dalla memoria deve essere scritto all'interno del register file nel registro specificato, ossia in `$t1`. In definitiva serviranno sia il register file che la ALU riportati in figura 5.6.

Inoltre saranno necessarie (figura 5.8) un'unità per estendere ad un numero di 32 bit con segno l'offset del campo di 16 bit dell'istruzione tramite un'operazione di estensione del segno, oltre ad un'unità di memoria dati in cui scrivere o da cui leggere. La memoria dati viene scritta durante le istruzioni store e sarà pertanto dotata dei segnali di controllo di lettura e di scrittura, di un ingresso di indirizzo e di un ingresso per i dati che si stanno scrivendo.

La figura 5.9 indica come combinare questi elementi nel costruire l'unità di elaborazione per un'istruzione load word o store word, assumendo che l'istruzione stessa sia già stata prelevata. Gli ingressi che specificano i numeri dei registri al register file provengono dai campi dell'istruzione, come pure il valore dell'offset che, dopo l'estensione del segno, diviene il secondo ingresso della ALU.

L'istruzione `beq` ha tre operandi: due registri che vengono confrontati per determinarne l'uguaglianza ed un offset a 16 bit utilizzato per calcolare l'indirizzo di destinazione del salto a partire dall'indirizzo dell'istruzione `beq` stessa; la sua forma è `beq $t1, $t2, offset`. Per implementare questa istruzione occorre calcolare l'indirizzo di destinazione del salto sommando il campo offset dell'istruzione (dopo averlo esteso a 32 bit con segno) a PC. Nella definizione delle istruzioni di salto condizionato vi sono due dettagli (si veda il Capitolo 3) ai quali è bene prestare attenzione:

- L'architettura del set di istruzioni specifica che il valore base per il calcolo dell'indirizzo di salto è l'indirizzo dell'istruzione che segue quella di salto.

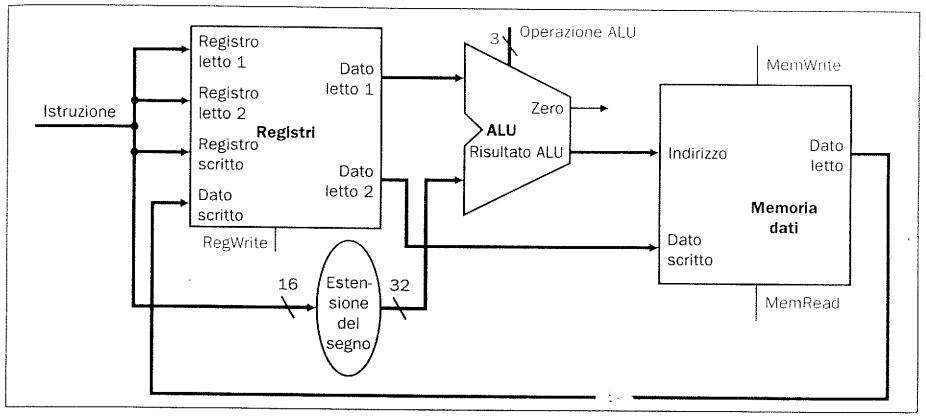


Figura 5.9 L'unità di elaborazione nell'esecuzione di una load o di una store effettua un accesso ai registri seguito dal calcolo di un indirizzo di memoria, una lettura o scrittura dalla memoria e, se l'istruzione è una load, una scrittura nel register file.

Dato che la parte di unità di elaborazione che si occupa del prelievo dell'istruzione calcola già il valore PC+4 (ossia l'indirizzo dell'istruzione seguente), risulterà semplice utilizzarlo come base per il calcolo dell'indirizzo di destinazione del salto.

- L'architettura stabilisce altresì che il campo offset è da intendersi scalato di 2 bit a sinistra, risultando espresso in termini di numero di parole; di fatto tale scalamento aumenta lo spazio di indirizzamento dell'offset. Si dovrà quindi scalare il campo di offset di due bit a sinistra.

Oltre a calcolare l'indirizzo di destinazione del salto è necessario determinare se la prossima istruzione da eseguire sia quella presente nella posizione di memoria successiva o quella all'indirizzo di destinazione. Quando la condizione è vera (ossia gli operandi sono uguali), l'indirizzo di destinazione del salto diventa il nuovo PC e si dice che il salto condizionato è stato *eseguito*; se gli operandi sono diversi, il valore di PC, una volta incrementato, viene posto nello stesso registro PC (come per ogni altra istruzione), e si dice che il salto è *non eseguito*.

In definitiva per i salti condizionati l'unità di elaborazione deve compiere due operazioni: calcolare l'indirizzo di destinazione del salto e confrontare i contenuti dei registri. Inoltre, come si vedrà tra poco, si dovrà anche modificare la porzione dell'unità di elaborazione relativa al prelievo dell'istruzione; tale unità è riportata in figura 5.10. Per il calcolo dell'indirizzo di destinazione vengono utilizzati un'unità di estensione del segno come quella di figura 5.8 ed un sommatore; i valori dei registri usati come operandi nel confronto provengono dal register file di figura 5.6 (di cui non si utilizza la funzionalità di scrittura) e sono inviati alla ALU progettata nel Capitolo 4. Poiché la ALU fornisce un segnale di uscita che indica se il risultato è pari a 0, si possono inviare i due operandi alla ALU indicando (tramite i segnali di controllo) di effettuare una sottrazione: se l'uscita Zero della ALU è affermata, i valori sono uguali; sebbene l'uscita Zero indichi sempre se il risultato è 0, essa è utilizzata solamente per il test di uguaglianza nei salti condizionati. In seguito si vedrà come debbano essere collegati i segnali di controllo della ALU all'interno dell'unità di elaborazione.

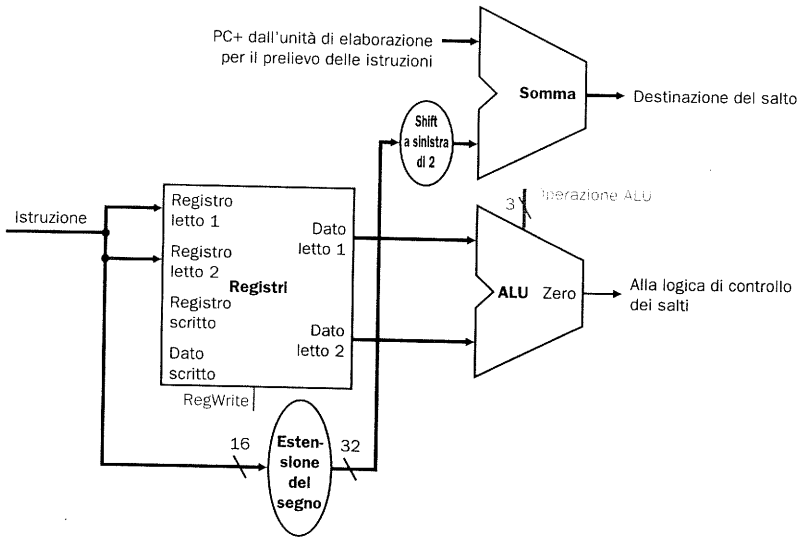


Figura 5.10 Per i salti condizionati l'unità di elaborazione utilizza la ALU per valutare la condizione di salto, ed un sommatore a parte per determinare la destinazione del salto, calcolata come somma del PC incrementato e dei 16 bit meno significativi dell'istruzione (lo spiazzamento del salto), scalati a sinistra di 2 bit ed estesi a 32 bit con segno. L'unità etichettata Shift a sinistra di 2 corrisponde semplicemente ad un reinstradamento dei segnali che aggiunge 00_{due} all'estremità inferiore del campo offset a valle dell'estensione del segno; essendo costante l'entità dello scalamento non è richiesto alcun componente dedicato. Siccome è noto che l'offset è stato esteso a partire da 16 bit, lo scalamento scarterà solamente dei "bit di segno". La logica di controllo deciderà se PC deve essere caricato con il valore di PC incrementato o con la destinazione del salto, in funzione dell'uscita Zero della ALU.

L'istruzione di salto incondizionato (jump) opera sostituendo i 28 bit meno significativi di PC con i 26 bit meno significativi dell'istruzione scalati a sinistra di 2 posizioni; lo scalamento si può effettuare concatenando 00 all'offset del salto (come già descritto in un approfondimento del Capitolo 3).

Dopo aver esaminato le parti di unità di elaborazione richieste per ciascuna classe di istruzioni, è possibile combinarle in un'unica unità di elaborazione ed aggiungere la logica di controllo che completi l'implementazione. Le unità di elaborazione mostrate nelle figure 5.5, 5.7, 5.9 e 5.10 saranno i blocchi costituenti di due diverse implementazioni: nel prossimo paragrafo si costruirà un'implementazione che richiede esattamente un ciclo di clock per ogni istruzione; nel Paragrafo 5.4 si esaminerà invece un'implementazione che per ciascuna istruzione utilizza più cicli di clock di minor durata.

Approfondimento: Nel set di istruzioni MIPS i salti condizionati sono *ritardati*, nel senso che l'istruzione immediatamente successiva a quella di salto viene sempre eseguita, *indipendentemente* dal fatto che la condizione di salto sia vera o falsa. Quando la condizione è falsa, l'esecuzione procede come in un salto normale; quando è vera, un salto ritardato esegue innanzitutto l'istruzione successiva a quella di salto, per poi saltare all'indirizzo di destinazione specificato. La motivazione che ha portato ai salti ritardati nasce dagli effetti della pipeline sui salti (si veda il Paragrafo 6.6); per semplicità in questo capitolo si ignoreranno i salti ritardati e si implementerà una versione non ritardata dell'istruzione `beq`.

5.3 Un'implementazione elementare

In questo paragrafo si esaminerà quella che può essere vista come la più semplice implementazione possibile del sottoinsieme MIPS scelto: si costruirà un'unità di elaborazione dati elementare e la relativa unità di controllo fondendo le porzioni di unità di elaborazione presentate nel paragrafo precedente ed aggiungendo gli opportuni segnali di controllo. Questa implementazione elementare copre le istruzioni load word (`lw`), store word (`sw`), branch equal (`beq`) e le istruzioni logico-aritmetiche `add`, `sub`, `and`, `or` e set on less than (`slt`); in seguito si estenderà il progetto con l'istruzione di jump (`j`).

Progetto dell'unità di elaborazione elementare

Si supponga di voler costruire un'unità di elaborazione a partire dai frammenti esaminati nelle figure 5.5, 5.7, 5.9 e 5.10: l'unità di elaborazione elementare complessiva potrebbe cercare di eseguire un'istruzione per ogni ciclo di clock. Questo significa che nessuna risorsa può essere utilizzata più di una volta per istruzione e che ogni elemento usato più di una volta deve essere duplicato: servirà quindi una memoria delle istruzioni separata da quella dei dati. Sebbene alcune unità funzionali debbano essere duplicate nel combinare le porzioni del paragrafo precedente, molti elementi potranno essere condivisi dai flussi di esecuzione di istruzioni diverse.

Per condividere un elemento dell'unità di elaborazione tra due diverse classi di istruzioni potrebbe essere necessario prevedere dei collegamenti multipli agli ingressi dell'elemento e far sì che un segnale di controllo selezioni uno di questi ingressi: tale scelta viene comunemente effettuata da un elemento detto *multiplexer*, anche se sarebbe più appropriato chiamarlo *selettore dati*. Il multiplexer, introdotto nel capitolo precedente (figura 4.8), sceglie tra diverse linee di ingresso in base al valore dei suoi segnali di controllo.

Esempio

Composizione di più unità di elaborazione

L'unità di elaborazione per le istruzioni logico-aritmetiche (o di tipo-R) della figura 5.7 e quella per le istruzioni di accesso alla memoria della figura 5.9 sono molto simili; le differenze principali sono:

- Il secondo ingresso della ALU è un registro (per le istruzioni di tipo-R) anziché la metà meno significativa dell'istruzione alla quale è stato esteso il segno (per le istruzioni di accesso alla memoria).
- Il valore memorizzato nel registro destinazione proviene dalla ALU (per le istruzioni di tipo-R) anziché dalla memoria (per l'istruzione load).

Si descriva come combinare le due unità di elaborazione utilizzando dei multiplexer e senza duplicare le unità funzionali comuni alle figure 5.7 e 5.9. Si ignorino i segnali di controllo dei multiplexer.

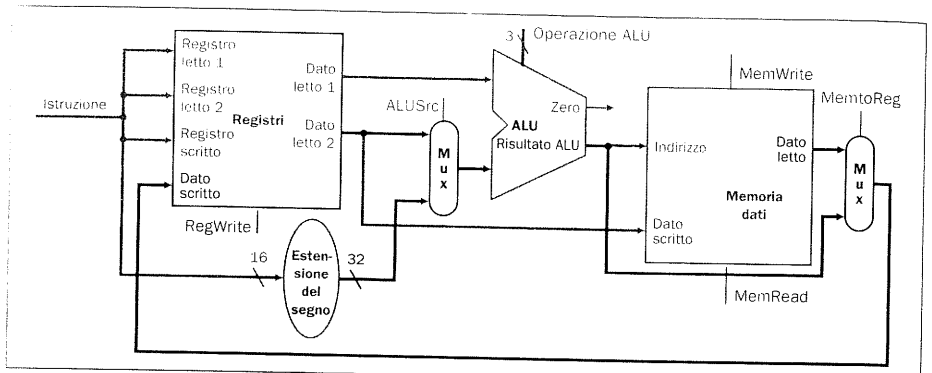


Figura 5.11 Composizione delle unità di elaborazione per le istruzioni di accesso alla memoria e per le istruzioni di tipo-R. Questo esempio evidenzia come si possa ottenere un'unica unità di elaborazione combinando i componenti delle figure 5.7 e 5.9 ed aggiungendo dei multiplexer. I multiplexer e le connessioni aggiunte sono evidenziati in figura; sono inoltre indicati i segnali di controllo dei multiplexer.

Soluzione

Per combinare le due unità di elaborazione, ed utilizzare un solo register file ed una sola ALU, si devono prevedere due fonti diverse sia per il secondo ingresso della ALU sia per i dati scritti nel register file. Quindi occorre porre un multiplexer all'ingresso della ALU ed un altro all'ingresso dati del register file, come indicato in figura 5.11.

La parte di unità di elaborazione dati per il reperimento dell'istruzione riportata in figura 5.5 può essere facilmente aggiunta all'unità di figura 5.11: la figura 5.12 ne mostra il risultato. L'unità di elaborazione risultante comprende una memoria per le istruzioni ed una memoria separata per i dati; inoltre richiede sia un sommatore che una ALU, dovendo nello stesso ciclo di clock usare il sommatore per incrementare PC e la ALU per eseguire l'istruzione. Si possono ora combinare tutte le parti ed ottenere un'unità di elaborazione elementare per l'architettura MIPS aggiungendo l'unità di elaborazione dei salti della figura 5.10: la figura 5.13 mostra l'unità di elaborazione così ottenuta. L'istruzione di salto condizionato utilizza la ALU principale per il confronto dei registri usati come operandi e perciò occorre continuare ad utilizzare il sommatore di figura 5.10 per calcolare l'indirizzo di destinazione dei salti. Un ulteriore multiplexer seleziona il valore da scrivere in PC tra l'indirizzo dell'istruzione successiva (PC+4) e l'indirizzo di destinazione del salto.

Avendo completato questa unità di elaborazione dati elementare, si può aggiungere l'unità di controllo: essa deve essere in grado di elaborare i propri ingressi e di generare un segnale di scrittura per ciascun elemento, un segnale di controllo per la selezione di ciascun multiplexer ed i segnali di controllo della ALU. Il controllo della ALU è per molti versi particolare, perciò conviene progettarlo prima del resto dell'unità di controllo.

L'unità di controllo per la ALU

Si ricordi dal Capitolo 4 che la ALU ha tre ingressi di controllo e che solo cinque delle otto combinazioni possibili sono utilizzate. La figura 4.20 indicava le seguenti combinazioni:

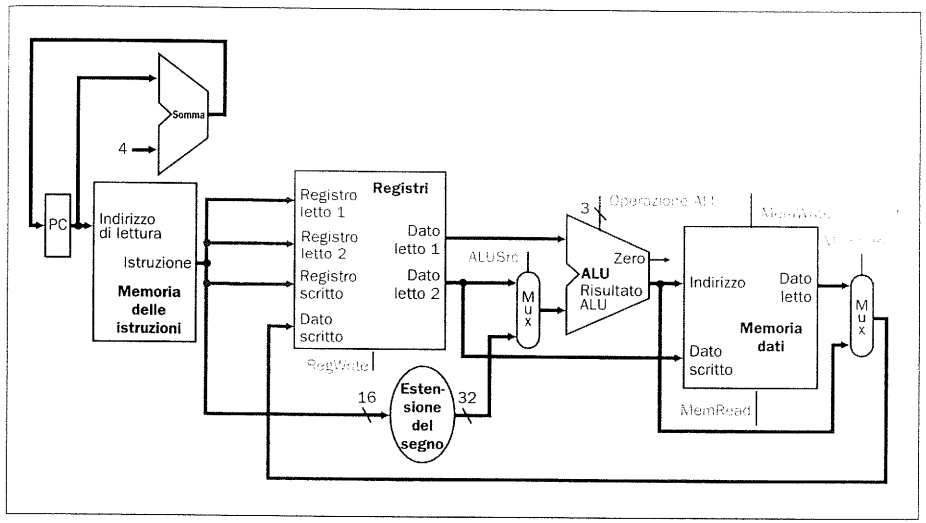


Figura 5.12 La parte di unità di elaborazione per il reperimento delle istruzioni della figura 5.5 è stata integrata nell'unità di elaborazione della figura 5.11 che gestisce le istruzioni di accesso alla memoria ed aritmetiche. La parte aggiunta è evidenziata. Il risultato è un'unità di elaborazione che supporta molte operazioni del set di istruzioni MIPS; le principali parti mancanti sono i salti condizionati ed incondizionati.

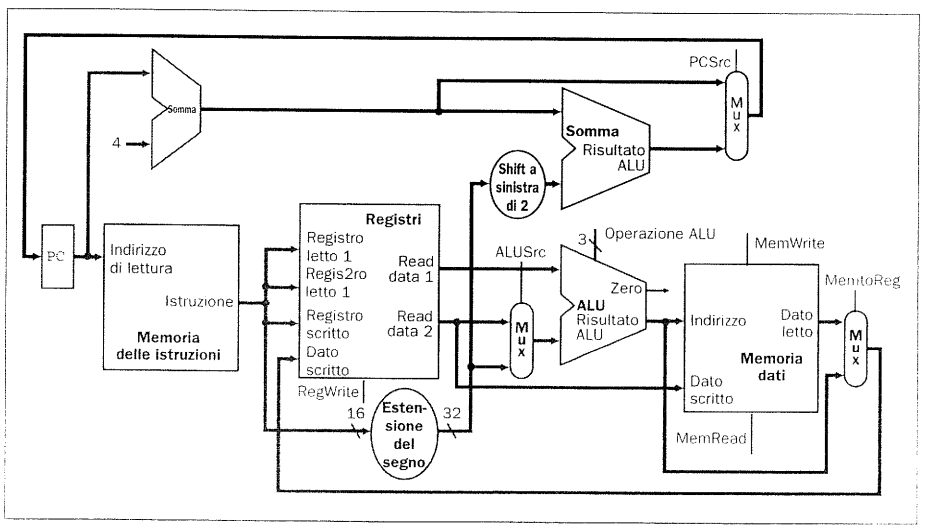


Figura 5.13 L'unità di elaborazione elementare per l'architettura MIPS combina gli elementi richiesti dalle diverse classi di istruzioni. Questa unità di elaborazione è in grado di eseguire le istruzioni principali (load/store word, operazioni con la ALU e salti) in un solo ciclo di clock. Le aggiunte rispetto alla figura 5.12, tutte evidenziate, sono utilizzate per l'implementazione dei salti condizionati e includono i componenti provenienti dalla figura 5.10. È stato altresì aggiunto un multiplexer poiché il valore caricato in PC può essere quello di PC incrementato o quello dell'indirizzo di destinazione. Il supporto per le istruzioni di salto incondizionato verrà aggiunto in seguito.

Ingresso di controllo della ALU	Funzione
000	AND
001	OR
010	somma
110	sottrazione
111	set on less than

In funzione della classe di istruzioni, la ALU dovrà compiere una di queste cinque operazioni: per le istruzioni load word e store word la ALU calcola l'indirizzo di memoria con una somma, per le istruzioni di tipo-R compie una delle cinque azioni (AND, OR, somma, sottrazione o set on less than) in funzione del valore del campo funct (funzione) nei 6 bit meno significativi dell'istruzione (si veda il Capitolo 3), per l'istruzione branch on equal effettua una sottrazione.

Si possono generare i 3 bit di controllo della ALU utilizzando una piccola unità di controllo che riceva in ingresso il campo funzione dell'istruzione ed un campo di controllo di 2 bit, chiamato ALUOp: esso indica se l'operazione da eseguire è una somma per load e store (00), una sottrazione per `beq` (01) o deve essere determinata dall'operazione codificata nel campo funct (10). L'uscita di tale unità di controllo della ALU è un segnale di 3 bit che controlla direttamente la ALU generando una delle cinque combinazioni di 3 bit appena viste.

La figura 5.14 riporta i valori da assegnare agli ingressi di controllo della ALU in funzione del segnale di controllo ALUOp e del codice funzione di 6 bit. Per completezza è anche riportata la relazione tra i bit ALUOp ed il codice operativo dell'istruzione; in seguito si vedrà come generare i bit ALUOp all'interno dell'unità di controllo principale.

L'utilizzo di livelli multipli di decodifica (per esempio, l'unità di controllo principale genera i bit ALUOp, che vengono a loro volta usati come ingressi per l'unità di controllo della ALU, la quale genera gli effettivi segnali di controllo dell'unità ALU) è una tecnica implementativa frequente. L'uso di più livelli di controllo può ridurre le dimensioni dell'unità di controllo principale, mentre avere più unità di controllo di dimensioni ridotte può aumentare la velocità delle operazioni legate al controllo; queste ottimizzazioni sono importanti, in quanto spesso l'unità di controllo è critica dal punto di vista delle prestazioni.

Codice operativo	ALUOp	Operazione	Campo funct	Azione della ALU	Ingresso di controllo della ALU
LW	00	load word	XXXXXX	somma	010
SW	00	store word	XXXXXX	somma	010
Branch equal	01	branch equal	XXXXXX	sottrazione	110
Tipo-R	10	somma	100000	somma	010
Tipo-R	10	sottrazione	100010	sottrazione	110
Tipo-R	10	AND	100100	and	000
Tipo-R	10	OR	100101	or	001
Tipo-R	10	set on less than	101010	set on less than	111

Figura 5.14 Il valore dei bit di controllo della ALU dipende dai bit di controllo ALUOp e dai diversi codici di funzione per le istruzioni di tipo-R. Il codice operativo, riportato nella prima colonna, determina il valore dei bit ALUOp (tutte le codifiche sono espresse in binario). Si noti che quando il codice ALUOp vale 00 o 01 i valori di uscita non dipendono dal campo funzione: in questo caso si dice che tale valore è "indifferente" (o *don't care*), e viene indicato con XXXXXX. Invece quando ALUOp vale 10 si usa il codice di funzione per determinare il valore degli ingressi di controllo della ALU.

ALUOp		Campo funct						Operazione
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Figura 5.15 La tabella di verità dei tre bit di controllo della ALU (detti Operazione). Gli ingressi sono i campi ALUOp e funct. Sono riportati solo gli elementi per cui i segnali di controllo della ALU sono affermati; sono stati aggiunti alcuni valori indifferenti (*don't care*): per esempio, ALUOp non utilizza la codifica 11, quindi la tabella di verità contiene i codici 1X e X1 anziché 10 e 01. Inoltre, quando il campo funzione viene utilizzato, i primi due bit (F5 e F4) di queste istruzioni sono sempre 10, quindi sono termini indifferenti e sono stati sostituiti da XX nella tabella di verità.

Vi sono diversi metodi per implementare la corrispondenza tra i campi ALUOp (di 2 bit) e funct (di 6 bit) ed i tre bit di controllo dell'operazione della ALU; poiché interessa una piccola frazione dei 64 valori possibili del campo funzione e poiché quest'ultimo è utilizzato solamente quando i bit ALUOp valgono 10, si può utilizzare un piccolo circuito logico che riconosce il sottoinsieme dei valori possibili e calcola correttamente i bit di controllo della ALU.

Come primo passo del progetto di questo circuito è utile creare una tabella di verità, riportata in figura 5.15, per le combinazioni interessanti del campo funzione e dei bit ALUOp: tale tabella di verità riporta il valore dei 3 bit dell'ingresso di controllo della ALU in funzione dei due campi di ingresso. Dato che la tabella è molto ampia (2^8=256 elementi) e che per molte combinazioni di ingresso il valore dei segnali di controllo della ALU è indifferente, sono riportate solamente le righe di interesse. In tutto questo capitolo si utilizzerà la convenzione di riportare nelle tabelle di verità solo gli elementi che devono essere affermati, tralasciando quelli che sono completamente a zero oppure indifferenti (questa convenzione ha uno svantaggio, discusso nel Paragrafo C.2 dell'Appendice C).

Poiché in molti casi non interessano i valori di alcuni ingressi, al fine di compattare le tabelle si includono dei termini con valore indifferente: tale valore è rappresentato da una X in una delle colonne di ingresso ed indica che l'uscita non dipende dal valore dell'ingresso corrispondente. Per esempio, quando i bit ALUOp sono 00, come nella prima riga della tabella in figura 5.15, i segnali di controllo della ALU sono sempre fissati a 010, indipendentemente dal codice di funzione: in tal caso gli ingressi del codice di funzione sono indifferenti. In seguito si vedranno altri tipi di termini indifferenti; il lettore che non abbia familiarità con il concetto di termini indifferenti è invitato a consultare l'Appendice B per maggiori informazioni.

Una volta costruita la tabella di verità, questa può essere ottimizzata e convertita in porte logiche; tale processo è di natura completamente meccanica. Il processo di traduzione ed il risultato finale sono riportati nel Paragrafo C.2 dell'Appendice C.

Progetto dell'unità di controllo principale

Avendo descritto il progetto di una ALU che, grazie alla logica di controllo aggiuntiva, può utilizzare il codice di funzione ed un segnale di 2 bit come ingressi di controllo, si può tornare a discutere il resto dell'unità di controllo, iniziando dall'identificazione dei campi delle istruzioni e dei segnali di controllo richiesti dall'unità di

ampo	0	rs	rt	rd	shamt	funct
osizioni dei bit	31-26	25-21	20-16	15-11	10-6	5-0

Istruzione di tipo-R

ampo	35 o 43	rs	rt	address
osizioni dei bit	31-26	25-21	20-16	15-0

Istruzione load o store

ampo	4	rs	rt	address
osizioni dei bit	31-26	25-21	20-16	15-0

Istruzione di salto condizionato

igura 5.16 Le tre classi di istruzioni (tipo-R, load/store e salto condizionato) utilizzano due diversi formati di istruzione. L'istruzione di salto incondizionato ha un formato ancora diverso, che verrà esaminato tra breve. (a) Formato delle istruzioni di tipo-R, aventi un codice operativo pari a 0: esse hanno tre operandi di tipo egistro, rs, rt e rd; i campi rs e rt sono le sorgenti, rd è la destinazione. La funzione svolta dalla ALU è codificata el campo funct e verrà decodificata dall'unità di controllo della ALU progettata nel paragrafo precedente. Le struzioni di tipo-R che verranno implementate sono `add`, `sub`, `and`, `or` e `slt`: essendo il campo shamt utilizzato olamente per le operazioni di scalamento, verrà ignorato in questo capitolo. (b) Formato delle istruzioni load odice operativo = 35_{dieci}) e store (codice operativo = 43_{dieci}): il registro rs è il registro base che viene sommato al ampo address di 16 bit per ottenere l'indirizzo di memoria; in una load, rt è il registro destinazione del valore etto, mentre in una store è il registro sorgente il cui valore deve essere scritto in memoria. (c) Formato dell'istruione branch equal (codice operativo = 4_{dieci}): i registri rs ed rt sono i registri sorgente che vengono confrontati per leterminarne l'uguaglianza; il campo address a 16 bit viene esteso a 32 bit, scalato e sommato a PC per calcolare indirizzo di destinazione del salto.

elaborazione costruita nella figura 5.13. Per definire le modalità di connessione dei campi di un'istruzione all'unità di elaborazione conviene rivedere i formati delle tre classi di istruzioni: tipo-R, salto condizionato e load/store; tali formati sono ripresi in figura 5.16.

Su questo formato di istruzioni vi sono parecchie osservazioni importanti che influiranno sul progetto:

- Il campo op, anche detto *opcode* o codice operativo, è sempre contenuto nei bit 31-26; si indicherà questo campo con Op[5-0].
- I due registri da leggere sono sempre specificati dai campi rs e rt, nelle posizioni 25-21 e 20-16; ciò è vero per le istruzioni di tipo-R, branch equal e store.
- Il registro base per le istruzioni load e store è sempre nei bit 25-21 (rs).
- L'offset a 16 bit per branch equal, load e store è sempre nelle posizioni 15-0.
- Il registro destinazione può essere in due posizioni: per una load è nelle posizioni 20-16 (rt), mentre per un'istruzione di tipo-R è nelle posizioni 15-11 (rd). Sarà quindi necessario un multiplexer per selezionare il campo dell'istruzione che indica il numero del registro da scrivere.

Utilizzando queste informazioni si possono aggiungere all'unità di elaborazione elementare le etichette che riportano i numeri dei bit dell'istruzione ed un multiplexer aggiuntivo (per l'ingresso "Registro scritto" del register file): la figura 5.17 riporta queste aggiunte, oltre all'unità di controllo della ALU, ai segnali di scrittura per gli elementi di stato, al segnale di lettura della memoria dati ed ai segnali di controllo dei multiplexer. Siccome tutti i multiplexer hanno due ingressi, ciascuno di essi richiede un solo segnale di controllo.

La figura 5.17 mostra sette linee di controllo di un bit ciascuna ed il segnale di controllo ALUop di 2 bit: mentre il funzionamento di ALUOp è già stato definito, è

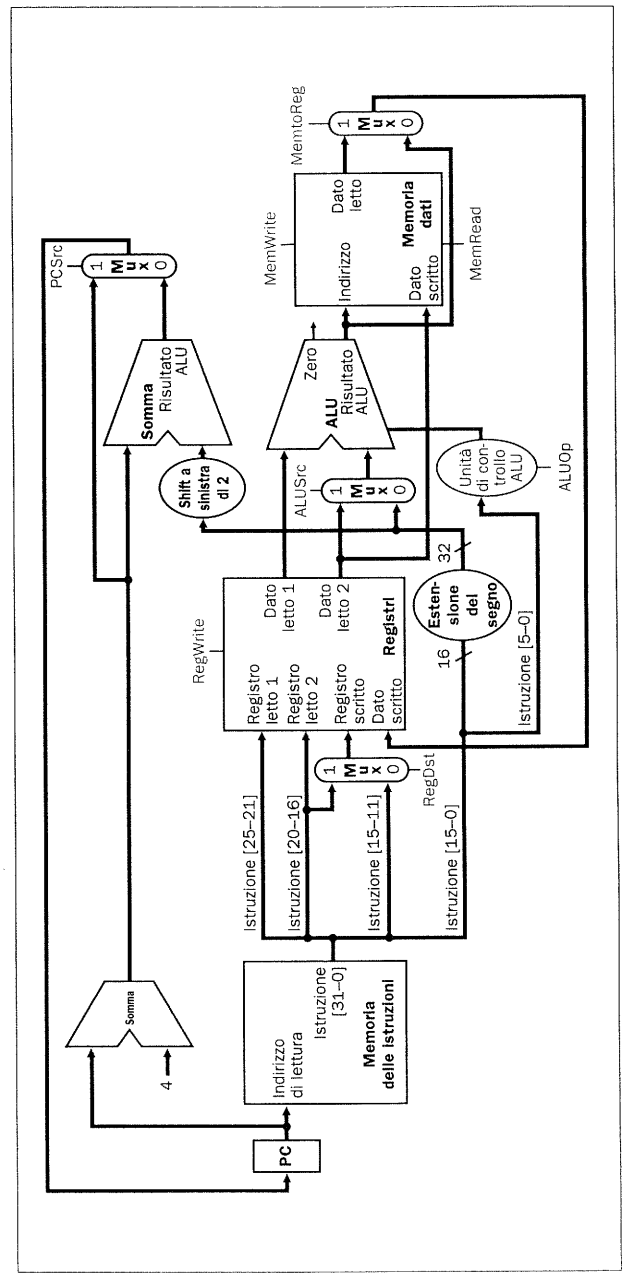


Figura 5.17 L'unità di elaborazione della figura 5.13 completa di tutti i multiplexer necessari e con l'identificazione di tutti i segnali di controllo. I segnali di controllo sono indicati con il colore ed è stata aggiunta anche l'unità di controllo della ALU. PC non necessita di segnali di controllo in quanto viene scritto al termine di ogni ciclo di clock; la logica di controllo dei salti determina se viene scritto PC incrementato o l'indirizzo di destinazione del salto.

utile determinare informalmente come devono operare i restanti segnali di controllo durante l'esecuzione delle istruzioni. La figura 5.18 descrive la funzione di queste sette linee di controllo.

Avendo visto la funzione di ciascuno di questi segnali di controllo, ci si può concentrare sul modo di calcolarli: l'unità di controllo calcolerà tutti i segnali di controllo, tranne uno, basandosi esclusivamente sul codice operativo dell'istruzione. L'eccezione è costituita dal segnale di controllo PCSrc: esso deve essere affermato se l'istruzione è branch on equal (decisione che l'unità di controllo è in grado di prendere) *e* se l'uscita Zero della ALU, utilizzata per il confronto di uguaglianza, è affermata. Per generare il segnale PCSrc occorre dunque collegare in AND un segnale proveniente dall'unità di controllo, che verrà chiamato *Branch*, con l'uscita Zero della ALU.

I nove segnali di controllo (i sette della figura 5.18 ed i due per ALUOp) si possono ora calcolare sulla base dei sei segnali di ingresso dell'unità di controllo, ossia dei bit del codice operativo: l'unità di elaborazione completa dell'unità di controllo e dei segnali di controllo è riportata in figura 5.19.

Prima di iniziare a scrivere un insieme di equazioni o una tabella di verità per l'unità di controllo è utile cercare di definirne informalmente la funzione: dato che il valore di ciascuna linea di controllo dipende solamente dal codice operativo, si definirà se ciascun segnale debba valere 0, 1 o *don't care* (X) per i diversi valori di opcode. La figura 5.20 definisce il valore dei segnali di controllo per ciascun valore di opcode, ed è ottenuta combinando direttamente le informazioni contenute nelle figure 5.14, 5.18 e 5.19.

Nome del segnale	Effetto quando non affermato	Effetto quando affermato
RegDst	Il numero del registro destinazione per Registro scritto proviene dal campo rt (bit 20-16).	Il numero del registro destinazione per Registro scritto proviene dal campo rd (bit 15-11).
RegWrite	Nessuno.	Nel registro specificato dall'ingresso Registro scritto è scritto il valore presente sull'ingresso Dato scritto.
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2).	Il secondo operando della ALU è la versione estesa dei 16 bit inferiori dell'istruzione.
PCSrc	Il valore di PC viene sostituito dall'uscita del sommatore che calcola il valore di PC+4.	Il valore di PC viene sostituito dall'uscita del sommatore che calcola la destinazione del salto.
MemRead	Nessuno.	Il contenuto della cella di memoria dati determinato dall'ingresso Indirizzo è posto sull'uscita Dato letto.
MemWrite	Nessuno.	Il contenuto della cella di memoria dati determinato dall'ingresso Indirizzo è sostituito dal valore presente sull'ingresso Dato scritto.
MemtoReg	Il valore inviato all'ingresso Dato scritto dei registri proviene dalla ALU.	Il valore inviato all'ingresso Dato scritto dei registri proviene dalla memoria dati.

Figura 5.18 L'effetto dei sette segnali di controllo. Quando il segnale di controllo di 1 bit di un multiplexer a 2 vie viene affermato, il multiplexer seleziona l'ingresso etichettato con 1; in caso contrario, cioè se il segnale di controllo è non affermato, il multiplexer seleziona l'ingresso 0. Si ricordi che tutti gli elementi di stato ricevono il clock come ingresso implicito e che il clock controlla le operazioni di scrittura: il clock non è mai manipolato esternamente ad un elemento di stato, in quanto ciò potrebbe causare problemi di temporizzazione (si veda l'Appendice B per la discussione di questo problema).

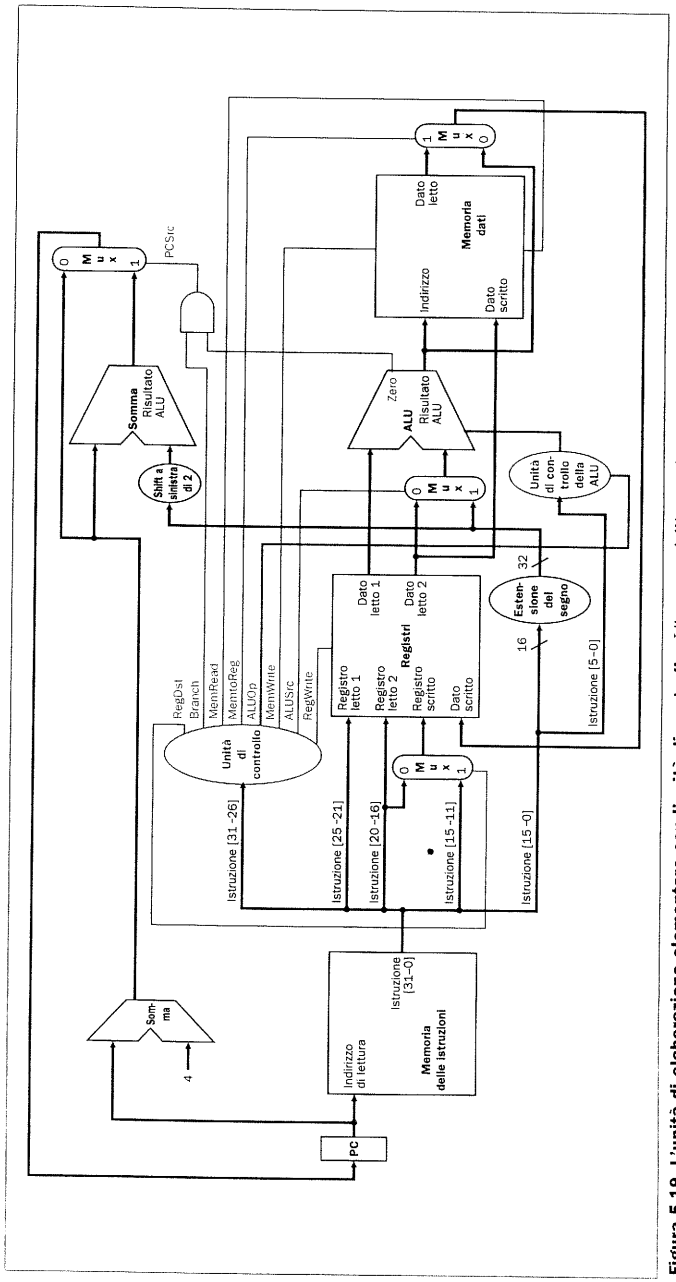


Figura 5.19 L'unità di elaborazione elementare con l'unità di controllo. L'ingresso dell'unità di controllo è il campo opcode di 6 bit proveniente dall'istruzione; le uscite consistono di tre segnali di 1 bit utilizzati per controllare altrettanti multiplexer (RegDst, ALUSrc e MemtoReg), tre segnali per controllare letture e scritture nel register file e nella memoria dati (RegWrite, MemRead e MemWrite), un segnale di 1 bit utilizzato per determinare la possibilità di un salto (Branch) e un segnale di 2 bit per il controllo della ALU (ALUOp). Una porta AND combina il segnale di controllo del salto con l'uscita Zero della ALU: l'uscita della porta AND determina la scelta del prossimo valore di PC. PCSrc non proviene più direttamente dall'unità di controllo, ma è un segnale derivato e verrà quindi reso anonimo nelle prossime figure.

Istruzione	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
formato-R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Figura 5.20 Il valore delle linee di controllo è completamente determinato a partire dal campo opcode dell'istruzione. La prima riga della tabella corrisponde alle istruzioni di formato-R (`add`, `sub`, `and`, `or` e `slt`), per le quali i campi registro sorgente sono rs e rt ed il campo registro destinazione è rd: ciò determina come calcolare i segnali ALUSrc e RegDst. Inoltre le istruzioni di tipo-R scrivono un registro (RegWrite=1) ma non leggono né scrivono la memoria dati. Quando il segnale di controllo Branch è a zero, PC viene aggiornato incondizionatamente con PC+4, altrimenti PC viene sostituito con la destinazione del salto, a patto che anche l'uscita Zero della ALU sia alta. Il campo ALUOp delle istruzioni di tipo-R è posto a 10 per indicare che i segnali di controllo della ALU dovranno essere generati a partire dal campo funct. La seconda e la terza riga riportano i segnali di controllo per `lw` e `sw`, dove ALUSrc e ALUOp assumono un valore tale da determinare il calcolo dell'indirizzo, mentre MemRead e MemWrite determinano il tipo di accesso alla memoria. Infine, durante una load RegDst e RegWrite sono pilotati per garantire che il valore venga scritto nel registro rt. L'istruzione di salto è simile alle istruzioni di tipo-R in quanto i registri rs e rt sono inviati alla ALU, mentre il campo ALUOp indica una sottrazione (01), utilizzata per controllare l'uguaglianza. Si noti che il campo MemtoReg è irrilevante quando il segnale RegWrite è 0 (dal momento che il registro non viene scritto, il valore del dato presente sulla porta di scrittura del registro non viene utilizzato), per cui il valore di MemtoReg nelle ultime due righe è sostituito da una X per indicarne la condizione di indifferenza. Si possono anche aggiungere valori indifferenti a RegDst ogniqualvolta RegWrite sia pari a 0. Questo tipo di valore indifferente deve essere aggiunto dal progettista, in quanto dipende dalla conoscenza del funzionamento dell'unità di elaborazione.

Il funzionamento dell'unità di elaborazione

Con le informazioni contenute nelle figure 5.18 e 5.20 si potrebbe progettare la logica dell'unità di controllo; tuttavia, prima di procedere con il progetto, si esaminerà l'uso che ciascuna istruzione fa dell'unità di elaborazione: nelle prossime figure si riporterà il flusso di tre diverse classi di istruzioni attraverso l'unità di elaborazione. I segnali di controllo affermati e gli elementi dell'unità di elaborazione di volta in volta coinvolti saranno evidenziati graficamente. Si noti che un multiplexer il cui segnale di controllo vale 0 compie un'azione ben definita, anche se il suo segnale di controllo non è evidenziato; i segnali di controllo a più bit saranno evidenziati se almeno uno dei segnali componenti è affermato.

La prima analisi è relativa ad un'istruzione di tipo-R, come `add $t1, $t2, $t3`. Anziché considerare l'unità di elaborazione come un unico blocco di logica combinatoria, è più semplice pensare che l'istruzione venga eseguita attraverso una serie di passi, concentrandosi sulla porzione di unità di elaborazione associata a ciascun passo. L'esecuzione di un'istruzione di tipo-R prevede quattro passi:

1. Viene prelevata un'istruzione dalla memoria ed incrementato PC, come indicato in figura 5.21, dove le unità attive ed i segnali di controllo affermati sono evidenziati; i segnali che risulteranno affermati nei passi successivi delle istruzioni di tipo-R sono in grigio, mentre in grigio chiaro sono riportate le unità non utilizzate in alcun passo dalle istruzioni di tipo-R. Nei passi successivi si seguirà la stessa convenzione.
2. Vengono letti dal register file (figura 5.22) due registri, `$t2` e `$t3`, mentre l'unità di controllo principale calcola i valori da attribuire alle linee di controllo.
3. La ALU elabora i dati letti dal register file, grazie ai segnali di controllo generati utilizzando il codice di funzione (bit 5-0, cioè il campo funct dell'istruzione). Questo passo è illustrato in figura 5.23.

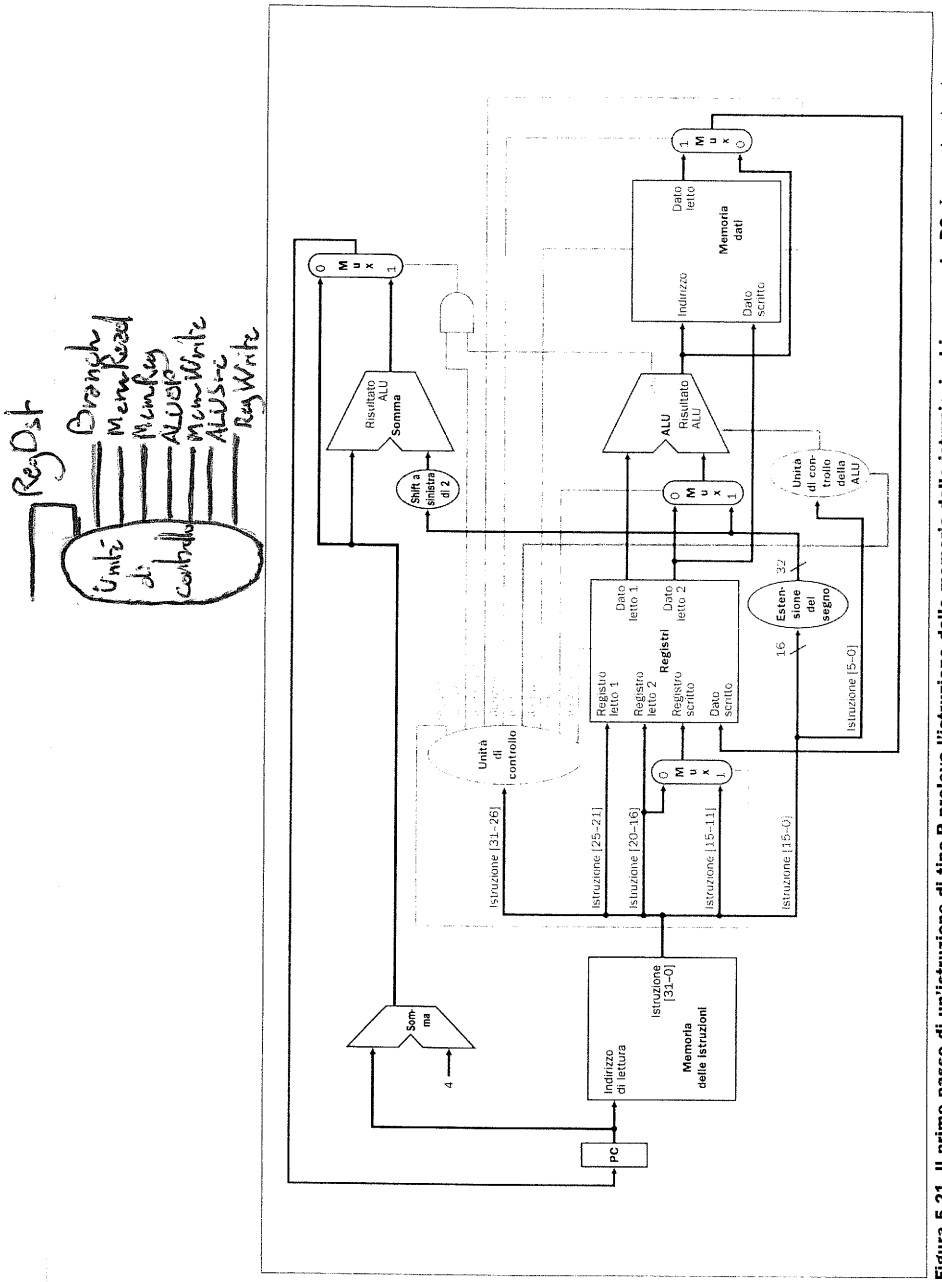


Figura 5.21 Il primo passo di un'istruzione di tipo-R preleva l'istruzione dalla memoria delle istruzioni ed incrementa PC. Le parti attive in questo passo sono evidenziate, mentre quelle disegnate in grigio chiaro non sono attive in questo passo ma lo diverranno in quelli successivi.

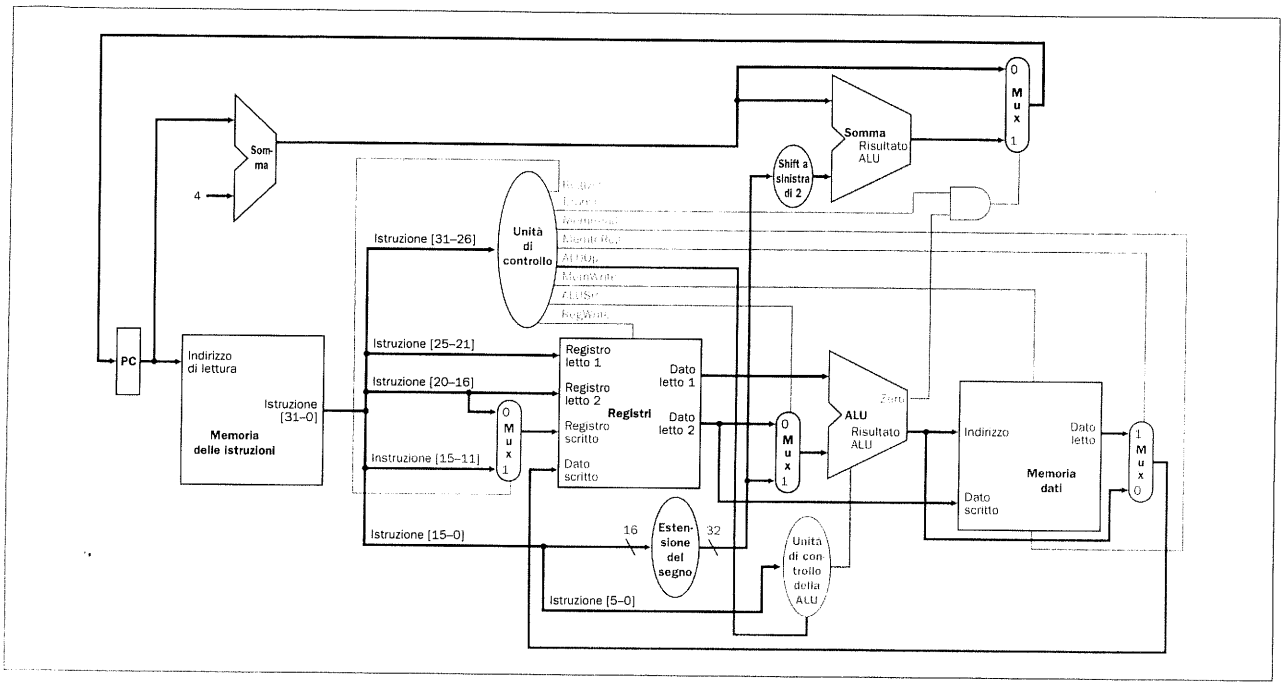


Figura 5.22 La seconda fase dell'esecuzione di un'istruzione di tipo-R legge i due registri sorgente dal register file. L'unità di controllo principale utilizza anche il campo opcode per determinare il valore dei segnali di controllo; queste unità diventano attive in aggiunta a quelle già attive durante il reperimento dell'informazione, riportate nella figura 5.21.

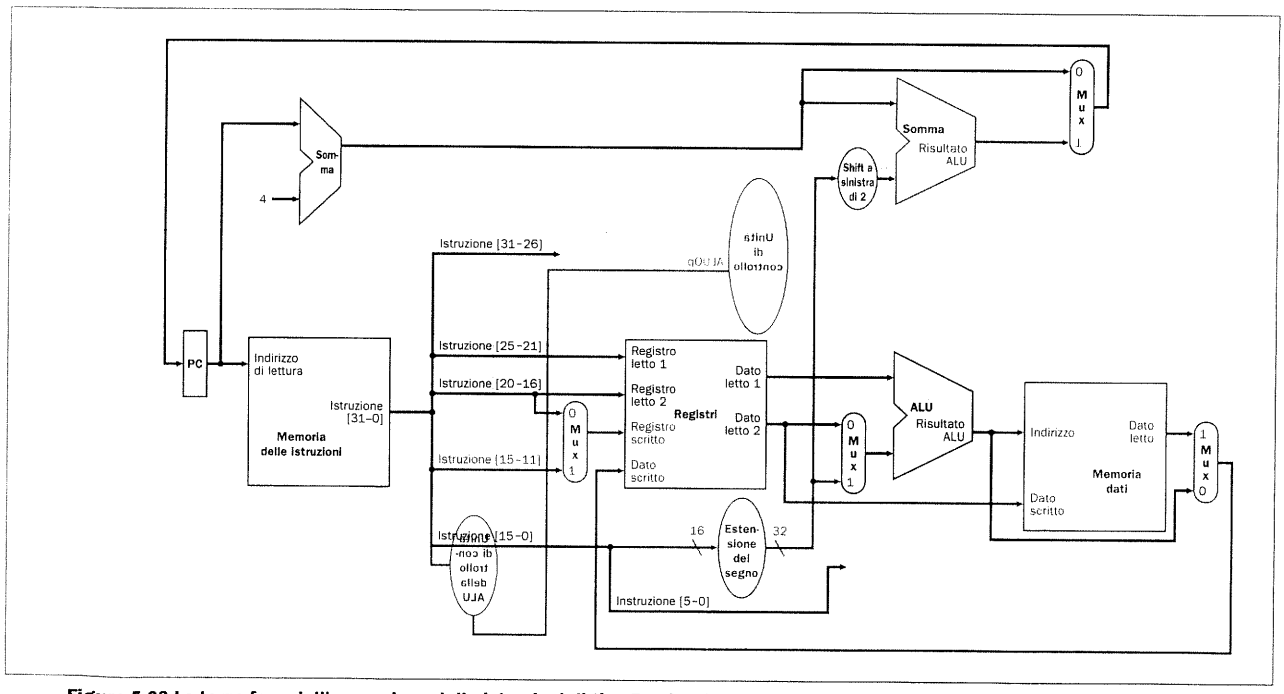


Figura 5.23 La terza fase dell'esecuzione delle istruzioni di tipo-R coinvolge la ALU, che elabora i valori dei registri utilizzati come operandi. Tutti i valori delle linee di controllo sono stati calcolati e l'unità di controllo della ALU ha calcolato la propria uscita; la ALU può quindi operare sui dati.

4. Il risultato calcolato dalla ALU viene scritto nel register file utilizzando i bit 15-11 dell'istruzione per selezionare il registro destinazione (`$t1`), come indicato in figura 5.24.

Si ricordi che questa implementazione è di tipo combinatorio, ossia non si comporta realmente come una sequenza di quattro passi distinti: l'unità di elaborazione opera effettivamente in un singolo ciclo di clock ed i segnali interni variano in modo difficilmente prevedibile durante tale ciclo. I segnali si stabilizzano pressappoco nell'ordine dei passi sopra citati in quanto il flusso delle informazioni segue tale ordine. In definitiva, la figura 5.24 non indica solo l'azione dell'ultimo passo, ma fondamentalmente rappresenta l'operazione svolta dall'intera unità di elaborazione alla fine del ciclo di clock.

Si può illustrare l'esecuzione di un'istruzione load word, come ad esempio

```
lw $t1, offset($t2)
```

con un metodo analogo a quello di figura 5.24: la figura 5.25 mostra le unità funzionali attive ed i segnali di controllo affermati. L'istruzione load può essere decomposta in cinque passi (così come l'istruzione di tipo-R era stata decomposta in quattro):

1. Viene prelevata un'istruzione dalla memoria ed incrementato PC.
2. Viene letto il valore di un registro (`$t2`) dal register file.
3. La ALU somma il valore letto dal register file ai 16 bit meno significativi dell'istruzione (`offset`), estesi a 32 bit con segno.
4. La somma calcolata dalla ALU è utilizzata come indirizzo per la memoria dati.
5. Il dato proveniente dall'unità di memoria è scritto nel register file: il registro destinazione è ricavato dai bit 20-16 dell'istruzione (`$t1`).

Si può infine esaminare con la stessa metodologia l'esecuzione di un'istruzione branch on equal, come `beq $t1, $t2, offset`: essa si comporta in modo molto simile ad un'istruzione di formato-R, ma l'uscita della ALU è utilizzata per scegliere se scrivere PC+4 o l'indirizzo di destinazione del salto nel registro PC. I quattro passi dell'esecuzione sono riportati in figura 5.26:

1. Viene prelevata un'istruzione dalla memoria ed incrementato PC.
2. Vengono letti due registri, `$t2` e `$t3`, dal register file.
3. La ALU calcola la differenza fra i valori dei dati letti dal register file, mentre il valore di PC+4 viene sommato ai 16 bit meno significativi dell'istruzione (`offset`), dopo averli estesi con segno e scalati a sinistra di due posizioni, ottenendo come risultato l'indirizzo di destinazione del salto.
4. Il valore Zero calcolato dalla ALU è utilizzato per decidere da quale sommatore prelevare il valore da memorizzare nel registro PC.

Nel prossimo paragrafo si esamineranno processori realmente sequenziali, nei quali cioè ciascun passo è eseguito in un ciclo di clock distinto.

Completamento dell'unità di controllo

Avendo esaminato come vengono eseguite le istruzioni nei diversi passi, si può procedere all'implementazione dell'unità di controllo. La funzione svolta da tale unità può essere determinata con precisione a partire dal contenuto della figura 5.20: le sue uscite sono le linee di controllo, mentre l'ingresso è il campo opcode

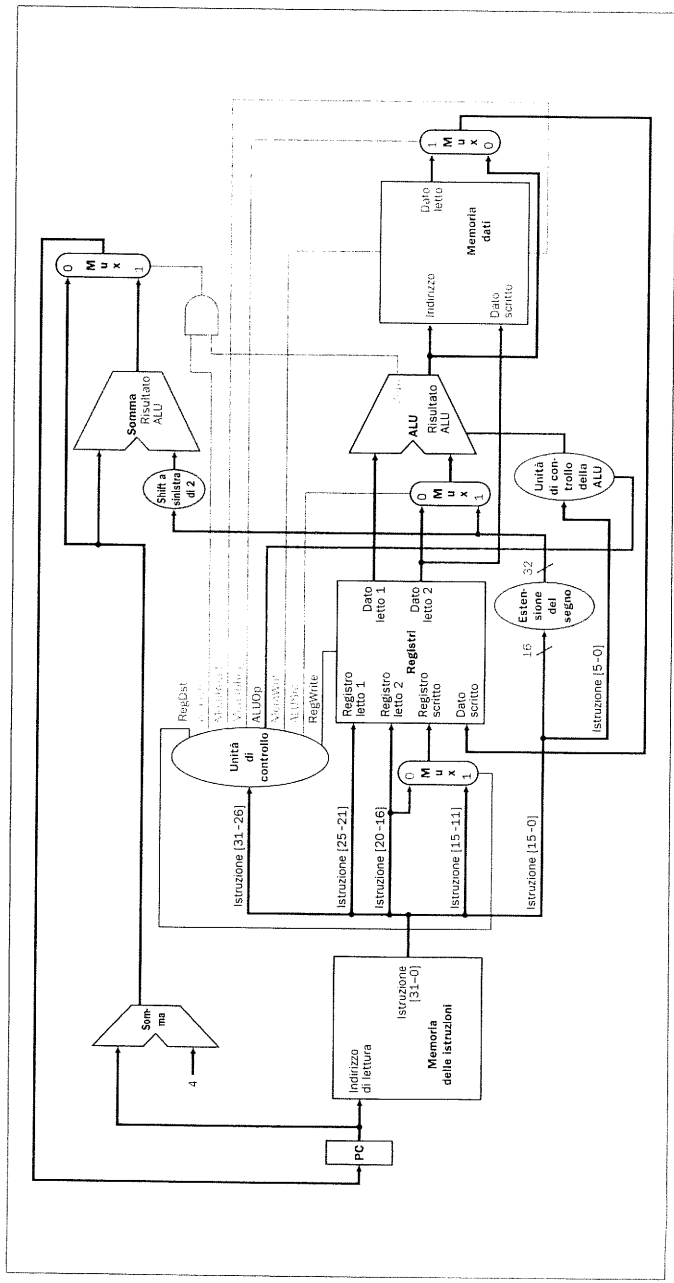


Figura 5.24 Il passo finale per le istruzioni di tipo-R prevede la scrittura del risultato; le relative unità attive sono state aggiunte a quelle dei tre passi precedenti, già indicate in figura 5.23. Al termine di questa fase avviene anche l'aggiornamento di PC. Essendo l'unità di elaborazione combinatoria, questo passo riporta tutte le unità attive e tutti i segnali di controllo affermati, una volta raggiunto lo stato stabile. Si osservi che l'istruzione verrà eseguita correttamente anche se utilizza lo stesso registro sia come ingresso che come uscita (come in `add R1, R1, R1`): il valore letto dai registri è il valore di `R1` scritto al termine di un ciclo di clock precedente, mentre il risultato non verrà scritto nel registro fino al prossimo fronte del clock.

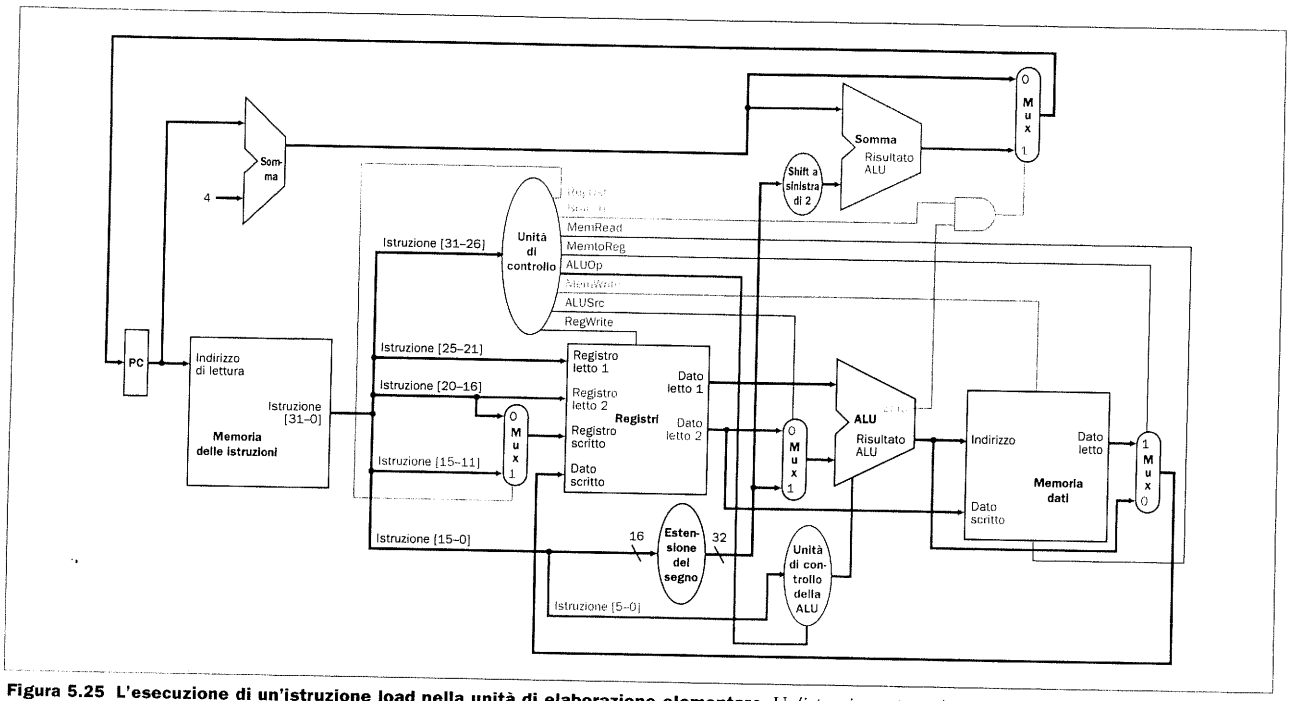


Figura 5.25 L'esecuzione di un'istruzione load nella unità di elaborazione elementare. Un'istruzione store si comporterebbe in modo molto simile: le differenze principali consterebbero dei segnali di controllo della memoria, che indicherebbero una scrittura anziché una lettura, del secondo registro letto, che conterrebbe il dato da memorizzare, e dell'operazione di scrittura del dato nel register file, che non avverrebbe.

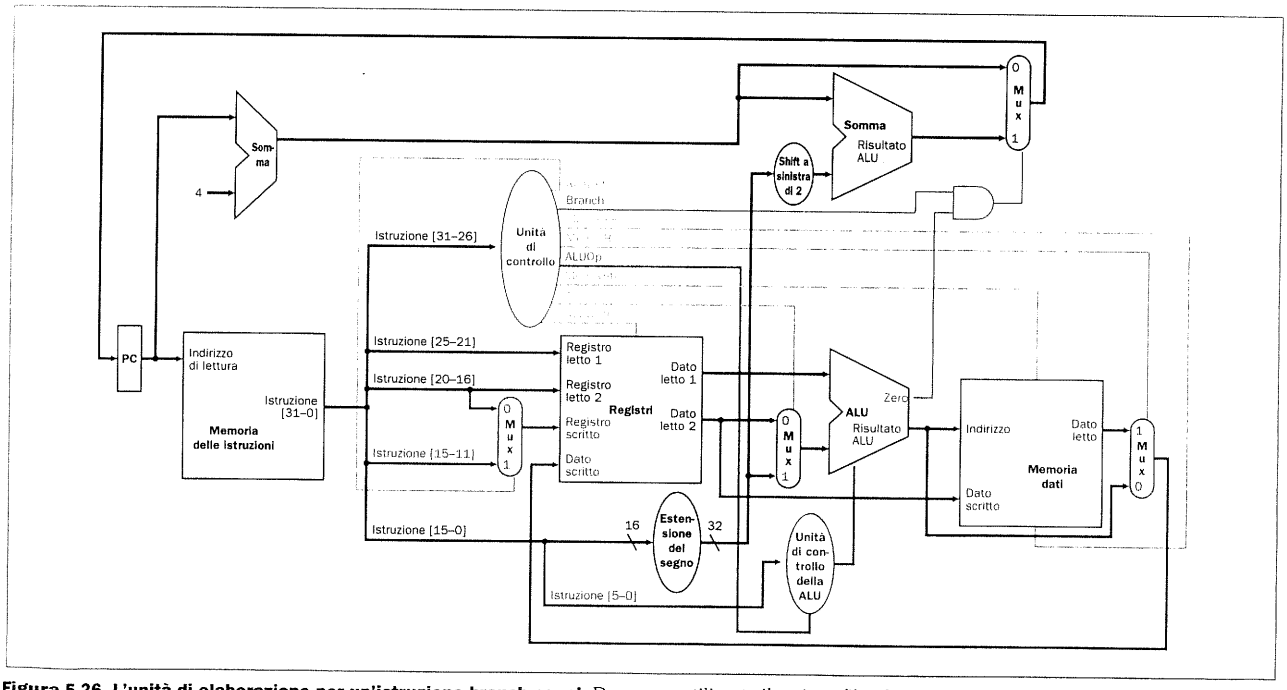


Figura 5.26 L'unità di elaborazione per un'istruzione branch equal. Dopo aver utilizzato il register file e la ALU per il confronto, l'uscita Zero è usata per scegliere il prossimo valore del program counter tra i due possibili.

a 6 bit, Op[5-0]. Si può quindi creare una tabella di verità per ciascuna delle uscite; per agevolare tale operazione, la tabella seguente riporta la codifica dei diversi codici operativi presenti in figura 5.20, sia come numeri decimali che come sequenza dei bit che diverranno l'ingresso dell'unità di controllo:

Nome	Opcode In decimale	Opcode in binario					
		Op5	Op4	Op3	Op2	Op1	Op0
formato-R	0_{dieci}	0	0	0	0	0	0
lw	35_{dieci}	1	0	0	0	1	1
sw	43_{dieci}	1	0	1	0	1	1
beq	4_{dieci}	0	0	0	1	0	0

Utilizzando queste informazioni, si può descrivere la logica dell'unità di controllo in una grande tabella di verità che combina tutte le uscite (figura 5.27): essa specifica completamente la funzione dell'unità di controllo e potrà essere implementata direttamente con porte logiche in modo del tutto automatico (come descritto nel Paragrafo C.2 dell'Appendice C).

A questo punto è possibile aggiungere l'istruzione jump; l'estensione dell'unità operativa e dell'unità di controllo per trattare altre istruzioni del set di istruzioni può essere realizzata seguendo la stessa metodologia.

Ingresso o uscita	Nome del segnale	formato-R	lw	sw	beq
Ingressi	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Uscite	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figura 5.27 La funzione dell'unità di controllo per l'implementazione a singolo ciclo di clock è completamente specificata da questa tabella di verità. La metà superiore della tabella fornisce le combinazioni di segnali di ingresso corrispondenti ai quattro codici operativi che determinano i valori delle uscite (si ricordi che Op[5-0] corrisponde ai bit 31-26 dell'istruzione, ossia al campo op). La parte inferiore fornisce invece le uscite: l'uscita RegWrite ad esempio è affermata per due diverse combinazioni degli ingressi. Se si considerano solamente i quattro opcode riportati, la tabella si può semplificare utilizzando delle condizioni di indifferenza tra gli ingressi: per esempio, si può identificare un'istruzione di formato-R con l'espressione $\overline{Op5} \bullet \overline{Op2}$, essendo questa sufficiente a distinguere le istruzioni di formato-R dalle lw, sw e beq. Dato che in un'implementazione completa i restanti codici operativi MIPS risulterebbero utilizzati, queste semplificazioni non verranno sfruttate.

Esempio

Implementazione dei salti incondizionati

La figura 5.19 riporta l'implementazione di molte delle istruzioni esaminate nel Capitolo 3. Una classe di istruzioni assente è quella relativa ai salti incondizionati: si estendano l'unità operativa e l'unità di controllo della figura 5.19 in modo da trattare anche l'istruzione jump. Si descriva come calcolare il valore di tutti i segnali di controllo aggiunti.

Campo	2	address
Posizioni dei bit	31-26	25-0

Figura 5.28 Formato dell'istruzione jump (opcode=2). L'indirizzo di destinazione di un'istruzione jump si ottiene concatenando i 4 bit più significativi del PC corrente + 4 con il campo address a 26 bit dell'istruzione jump, ed aggiungendo 00 come 2 bit meno significativi.

Soluzione

L'istruzione jump è in qualche modo simile ad un'istruzione branch, ma calcola in modo diverso il valore di PC per la destinazione e non è condizionata. Come per i salti condizionati, i 2 bit inferiori dell'indirizzo di salto sono sempre 00_{due}, i successivi 26 bit inferiori dell'indirizzo a 32 bit provengono dal campo immediato di 26 bit dell'istruzione, come indicato in figura 5.28. I 4 bit superiori dell'indirizzo che sostituirà quello presente in PC provengono dal PC dell'istruzione di salto, incrementato di 4. Si può quindi implementare la jump memorizzando in PC la concatenazione dei seguenti bit:

- i 4 bit più significativi del valore corrente di PC + 4 (sono i bit 31-28 dell'indirizzo dell'istruzione che segue la jump)
- il campo immediato di 26 bit dell'istruzione jump
- i bit 00_{due}.

La figura 5.29 mostra i segnali di controllo aggiunti rispetto alla figura 5.19 per l'implementazione della jump: un ulteriore multiplexer viene utilizzato per selezionare la fonte del nuovo valore di PC, che può essere PC incrementato (PC+4), l'indirizzo di destinazione di un salto condizionato, o l'indirizzo di destinazione della jump. Un segnale di controllo aggiuntivo, denominato *Jump*, pilota il multiplexer aggiunto, e viene affermato solo quando l'istruzione è una jump, cioè quando l'opcode è pari a 2.

Perché non si utilizzano implementazioni a singolo ciclo?

Sebbene il progetto a singolo ciclo funzioni correttamente, non viene utilizzato nelle implementazioni moderne a causa della sua inefficienza: per comprendere ciò si noti che il periodo di clock deve avere la stessa durata in tutte le istruzioni previste, quindi il parametro CPI (definito nel Capitolo 2) è pari ad 1. Il periodo di clock è determinato dal cammino più lungo nel sistema, che probabilmente è un'istruzione di load poiché utilizza cinque unità funzionali in serie: la memoria delle istruzioni, il register file, la ALU, la memoria dati ed il register file. Sebbene CPI sia pari ad 1, le prestazioni complessive di un'implementazione a singolo ciclo probabilmente non saranno eccellenti, poiché per molte classi di istruzioni sarebbe sufficiente un periodo di clock più breve.

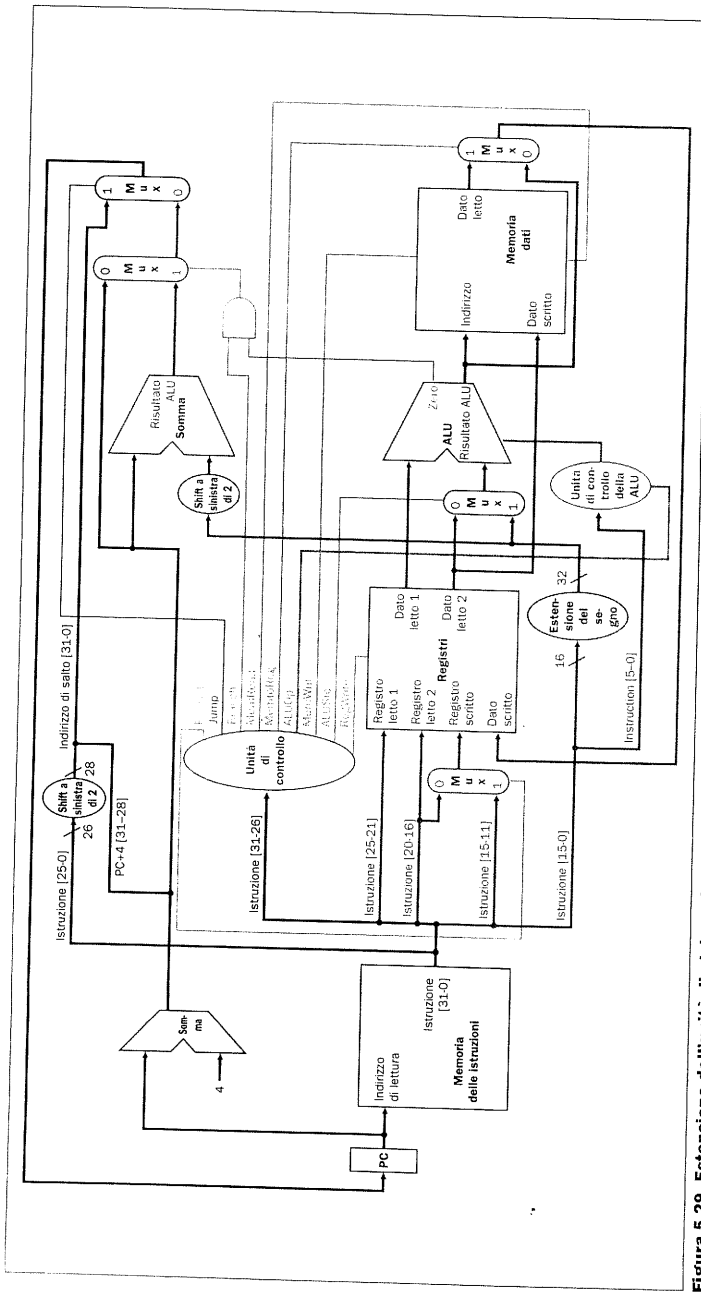


Figura 5.29 Estensione dell'unità di elaborazione elementare e della relativa unità di controllo per implementare l'istruzione jump. Viene utilizzato un multiplexer aggiuntivo (in alto a destra) per selezionare la destinazione del jump o, in alternativa, la destinazione del branch o l'istruzione seguente; il multiplexer è controllato dal segnale di controllo denominato Jump. Per ottenere l'indirizzo di destinazione del salto si procede nel seguente modo: si scalano a sinistra di 2 bit i 26 bit meno significativi dell'istruzione (aggiungendo in tal modo 00 come bit di ordine inferiore) e si concatenano al risultato ottenuto i 4 bit superiori di PC+4 in modo da ottenere un indirizzo su 32 bit.

Prestazioni dei sistemi a singolo ciclo

Esempio

Si assuma che i tempi di calcolo per le unità funzionali principali dell'implementazione in esame siano i seguenti:

- Unità di memoria: 2 ns (nanosecondi)
- ALU e sommatori: 2 ns
- Register file (lettura o scrittura): 1 ns

Assumendo che i multiplexer, l'unità di controllo, gli accessi a PC, l'unità di estensione del segno e le interconnessioni non introducano ritardi aggiuntivi, quale implementazione tra le seguenti risulta più veloce, e di quanto?

1. Un'implementazione in cui ciascuna istruzione viene eseguita in 1 ciclo di clock di lunghezza fissa.
2. Un'implementazione in cui ciascuna istruzione viene eseguita in 1 ciclo di clock adottando un clock di lunghezza variabile, pari al minimo richiesto per ciascuna istruzione (un tale approccio non è facilmente realizzabile, ma il confronto permetterà di vedere quanto viene sacrificato eseguendo tutte le istruzioni in un solo ciclo di clock di lunghezza fissa).

Per confrontare le prestazioni, si ipotizzi la seguente combinazione di istruzioni: 24% di load, 12% di store, 44% di istruzioni di formato-R, 18% di salti condizionati, 2% di salti incondizionati.

Soluzione

Si può iniziare dal confronto dei tempi di esecuzione della CPU, ricordando dal Capitolo 2 che

tempo di esecuzione della CPU = numero di istruzioni × CPI × periodo del clock

Essendo CPI pari ad 1, l'espressione si può semplificare come

tempo di esecuzione della CPU = numero di istruzioni × periodo del clock

A questo punto è sufficiente trovare il periodo del clock, ossia la durata del ciclo di clock, per le due implementazioni, essendo il numero di istruzioni ed il valore di CPI identici nelle due implementazioni. Il cammino critico per le diverse classi di istruzioni risulta essere il seguente:

Classe di istruzioni	Unità funzionali utilizzate				
Formato-R	Prelievo dell'istruzione	Accesso a registri	ALU	Accesso a registri	
Load word	Prelievo dell'istruzione	Accesso a registri	ALU	Accesso alla memoria	Accesso a registri
Store word	Prelievo dell'istruzione	Accesso a registri	ALU	Accesso alla memoria	
Branch	Prelievo dell'istruzione	Accesso a registri	ALU		
Jump	Prelievo dell'istruzione				

Con questi cammini critici si può calcolare la durata richiesta per ciascuna classe di istruzioni:

Classe di istruzioni	Memoria delle istruzioni	Lettura dei registri	Operazione della ALU	Memoria dei dati	Scrittura dei registri	Totale
Formato-R	2	1	2	0	1	6 ns
Load word	2	1	2	2	1	8 ns
Store word	2	1	2	2		7 ns
Branch	2	1	2			5 ns
Jump	2					2 ns

Il ciclo di clock di un sistema con lo stesso clock per tutte le istruzioni sarà determinato dall'istruzione più lenta, che dura 8 ns (queste misure sono approssimate, vista la semplicità del modello di temporizzazione adottato; in realtà la temporizzazione dei moderni sistemi digitali è più complessa, e ad esempio spesso è consentito il "prestito" di una frazione di tempo da un ciclo di clock al successivo).

Un sistema a clock variabile avrà un ciclo di clock compreso tra 2 ns e 8 ns; il valore medio si può ricavare dalle informazioni appena calcolate e dalla distribuzione di frequenza delle istruzioni.

In definitiva, la durata media delle istruzioni per il sistema a clock variabile è

$$\text{periodo del clock} = 8 \times 24\% + 7 \times 12\% + 6 \times 44\% + 5 \times 18\% + 2 \times 2\% = 6{,}3 \text{ ns}$$

Dato che l'implementazione a clock variabile presenta un ciclo di clock di durata media inferiore, sarà necessariamente più veloce. Il rapporto tra le prestazioni, indicando con NI il numero di istruzioni, è:

$$\frac{\text{prestazioni della CPU}_{\text{clock variabile}}}{\text{prestazioni della CPU}_{\text{clock fisso}}} = \frac{\text{tempo di esecuzione della CPU}_{\text{clock fisso}}}{\text{tempo di esecuzione della CPU}_{\text{clock variabile}}} = \frac{\text{NI} \times \text{periodo di clock della CPU}_{\text{clock fisso}}}{\text{NI} \times \text{periodo di clock della CPU}_{\text{clock variabile}}} = \frac{\text{periodo di clock della CPU}_{\text{clock fisso}}}{\text{periodo di clock della CPU}_{\text{clock variabile}}} = \frac{8}{6{,}3} = 1{,}27$$

L'implementazione a clock variabile sarebbe quindi 1,27 volte più veloce. Sfortunatamente l'implementazione di un clock a frequenza variabile per le diverse classi di istruzioni è estremamente difficile ed il costo di un tale approccio potrebbe essere maggiore del beneficio; nel prossimo paragrafo verrà illustrato, in alternativa, come utilizzare un periodo di clock più breve, in cui viene compiuto un minor numero di operazioni e viene variato il numero di cicli di clock nelle diverse classi di istruzioni.

Il prezzo da pagare nei progetti a clock singolo è significativo, ma potrebbe essere considerato accettabile per set di istruzioni limitati come quello analizzato. Se si cercasse invece di implementare l'unità di elaborazione in virgola mobile o un set di istruzioni comprendente istruzioni più complesse, un progetto a singolo clock sarebbe del tutto inaccettabile, come illustra il seguente esempio relativo alle istruzioni in virgola mobile.

Esempio

Prestazioni di CPU a singolo ciclo con istruzioni in virgola mobile

Si supponga di disporre di un'unità in virgola mobile che richieda 8 ns per una somma in virgola mobile e 16 ns per una moltiplicazione. I tempi di tutte le altre unità funzionali siano identici a quelli dell'esempio precedente, e l'esecuzione di un'istruzione in virgola mobile proceda come quella delle istruzioni logico-aritmetiche, tranne per il fatto che viene utilizzata la ALU in virgola mobile anziché quella principale. Utilizzando la distribuzione delle istruzioni del programma spice vista nel Capitolo 4 (figura 4.54), si calcoli il rapporto di prestazioni tra un'implementazione in cui il ciclo di clock abbia una durata variabile nelle diverse classi di istruzioni ed una in cui tale durata sia fissa. Si assuma inoltre che:

- Tutte le load impieghino lo stesso tempo ed ammontino al 31% delle istruzioni.
- Tutte le store impieghino lo stesso tempo ed ammontino al 21% delle istruzioni.
- Le istruzioni di formato-R ammontino al 27% del totale.
- I salti condizionati contribuiscano per il 5%, mentre i salti incondizionati per il 2%.
- Somma e sottrazione in virgola mobile abbiano la stessa durata e, complessivamente, ammontino al 7% del numero di istruzioni.
- Moltiplicazione e divisione in virgola mobile abbiano la stessa durata e, complessivamente, ammontino al 7% del numero di istruzioni.

Soluzione

Dall'esempio precedente è noto che

$$\frac{\text{prestazioni della CPU}_{\text{clock variabile}}}{\text{prestazioni della CPU}_{\text{clock fisso}}} = \frac{\text{periodo di clock della CPU}_{\text{clock fisso}}}{\text{periodo di clock della CPU}_{\text{clock variabile}}}$$

Il periodo di clock per il sistema a clock fisso sarà pari alla durata dell'istruzione più lenta, ossia la moltiplicazione in virgola mobile: tale durata, e quindi il periodo di clock, è pari a 2 + 1 + 16 + 1 = 20 ns.

Si consideri un sistema in cui le diverse istruzioni abbiano diversi periodi di clock: la durata di un'addizione in virgola mobile è pari a 2+1+8+1=12 ns. Il prodotto delle durate delle istruzioni per le relative frequenze fornisce la lunghezza media del periodo di clock:

$$\text{periodo del clock} = 8 \times 31\% + 7 \times 21\% + 6 \times 27\% + 5 \times 5\% + 2 \times 2\% + 20 \times 7\% + 12 \times 7\% = 7{,}0 \text{ ns}$$

Il miglioramento di prestazioni è quindi pari a

$$\frac{\text{prestazioni della CPU}_{\text{clock variabile}}}{\text{prestazioni della CPU}_{\text{clock fisso}}} = \frac{\text{periodo di clock della CPU}_{\text{clock fisso}}}{\text{periodo di clock della CPU}_{\text{clock variabile}}} = \frac{20}{7} = 2{,}9$$

In definitiva, un sistema a clock variabile permetterebbe di migliorare le prestazioni di un fattore pari a 2,9 volte.

In modo del tutto analogo, se si disponesse di un calcolatore con istruzioni e modi di indirizzamento più potenti, la durata delle istruzioni potrebbe variare da un valore pari al ritardo di tre o quattro unità funzionali sino a quello corrispondente a decine o centinaia di unità. Inoltre, visto che la durata del ciclo di clock deve essere pari al valore peggiore tra quelli richiesti dalle diverse istruzioni, non si possono sfruttare tecniche implementative che riducano il ritardo dei casi più comuni ma che non riducano il tempo di ciclo nel caso peggiore: un'implementazione ad un solo ciclo viola dunque il principio di progetto fondamentale che prevede di rendere veloce l'evento più frequente.

Come se non bastasse, nelle implementazioni a singolo ciclo ciascuna unità funzionale può essere utilizzata solamente una volta per ciclo di clock: alcune unità dovranno necessariamente essere duplicate, contribuendo così ad un maggior costo dell'implementazione. Un progetto a singolo ciclo è inefficiente sia dal punto di vista delle prestazioni che da quello del costo dell'hardware!

Si possono aggirare questi ostacoli utilizzando tecniche implementative con un clock di durata inferiore (determinato dai ritardi delle unità funzionali elementari) e che richiedano diversi cicli di clock per istruzione; il prossimo paragrafo esaminerà questo schema implementativo alternativo. Il Capitolo 6 tratterà una tecnica diversa, basata sulle pipeline, con un'unità di elaborazione molto simile a quella a ciclo singolo, ma molto più efficiente: le pipeline incrementano l'efficienza grazie alla sovrapposizione temporale dell'esecuzione di diverse istruzioni, aumentando il fattore di utilizzo dell'hardware e migliorando le prestazioni.

5.4 Un'implementazione multi-ciclo

Nell'esempio del paragrafo precedente, ciascuna istruzione era stata decomposta in una serie di passi corrispondenti alle operazioni richieste dalle diverse unità funzionali: tali passi possono essere utilizzati per creare un'*implementazione a più cicli* (o *multi-ciclo*). In un'implementazione multi-ciclo, ciascun *passo* richiede un ciclo di clock. In questo caso la stessa unità funzionale può essere utilizzata più di una volta per istruzione, purché in cicli di clock diversi: la condivisione delle unità funzionali permette di ridurre la quantità di hardware richiesto. La possibilità di avere istruzioni con un diverso numero di cicli di clock e la possibilità di condividere unità funzionali per la stessa istruzione sono i vantaggi principali dei progetti multi-ciclo. Una versione astratta di un'unità operativa multi-ciclo è riportata in figura 5.30, e può essere confrontata con l'unità operativa a ciclo singolo della figura 5.13. Le principali caratteristiche della versione multi-ciclo sono:

- È sufficiente una sola unità di memoria per istruzioni e dati.
- È presente una sola ALU, anziché una ALU e due sommatori.
- Uno o più registri sono aggiunti a ciascuna unità funzionale per memorizzarne l'uscita unità fino a quando, in un ciclo di clock successivo, il valore verrà utilizzato.

Al termine di ogni ciclo di clock, tutti i dati utilizzati nei cicli di clock successivi dovranno essere memorizzati in un elemento di stato. I dati utilizzati da *istruzioni successive* sono memorizzati negli elementi di stato visibili al programmatore (ossia il register file, PC o la memoria), mentre i dati utilizzati nella *stessa istruzione* ma in un ciclo di clock successivo sono memorizzati in uno dei registri aggiuntivi.

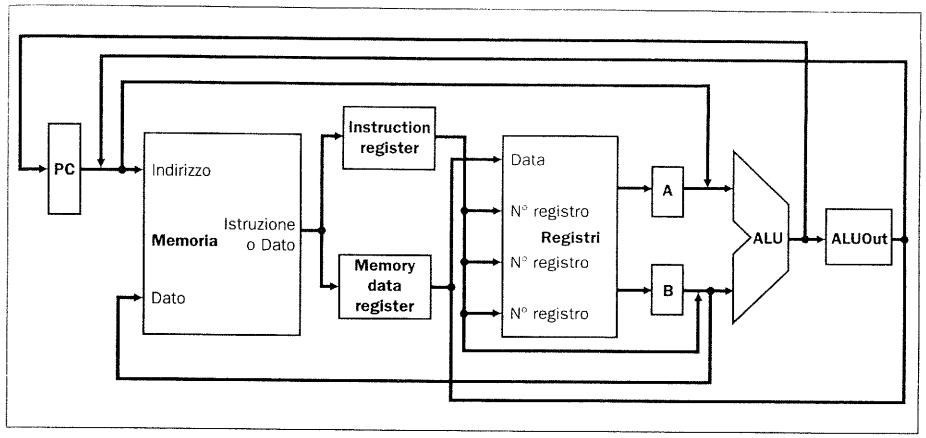


Figura 5.30 Una vista ad alto livello dell'unità di elaborazione multi-ciclo. La figura mostra gli elementi principali dell'unità di elaborazione: un'unità di memoria condivisa, una sola ALU condivisa dalle istruzioni, le connessioni tra tali unità condivise. L'adozione di unità funzionali condivise richiede l'aggiunta o l'espansione dei multiplexer e l'utilizzo di nuovi registri temporanei che memorizzino i dati tra i cicli di clock della stessa istruzione: i registri addizionali sono l'Instruction Register (IR, registro dell'istruzione), il Memory Data Register (MDR, letteralmente: registro per i dati provenienti dalla memoria), A, B e ALUOut.

La posizione dei registri aggiuntivi è quindi determinata da due fattori: quali unità funzionali riescono a lavorare in un solo ciclo di clock e quali dati saranno richiesti nei cicli successivi per implementare le istruzioni. Nel progetto multi-ciclo si assumerà che il ciclo di clock possa contenere al più una delle seguenti operazioni: accesso in memoria, accesso al register file (due letture o una scrittura), operazione della ALU; tutti i dati prodotti da una di queste tre unità funzionali (memoria, register file, ALU) dovranno essere salvati in un registro temporaneo per essere utilizzati nei cicli di clock successivi.

Per soddisfare i requisiti citati sono quindi stati aggiunti i seguenti registri:

- Il registro dell'istruzione (o Instruction Register, IR) ed il registro dei dati di memoria (o Memory Data Register, MDR) memorizzano l'uscita della memoria rispettivamente durante le operazioni di lettura di un'istruzione e di un dato. Sono utilizzati due registri separati in quanto, come emergerà tra breve, i due valori saranno necessari nello stesso ciclo di clock.
- I registri A e B memorizzano i valori dei registri letti dal register file.
- Il registro ALUOut memorizza l'uscita della ALU.

Tutti i registri, fatta eccezione per IR, memorizzano i dati solamente tra un ciclo di clock e quello immediatamente successivo e pertanto non richiedono un segnale di controllo della scrittura. Il registro IR, invece, deve memorizzare l'istruzione fino alla fine della sua esecuzione, richiedendo quindi un segnale di controllo per la scrittura; questa distinzione sarà più chiara quando si esamineranno i diversi cicli di clock per le diverse istruzioni.

Per poterle utilizzare a scopi diversi, alcune unità funzionali sono state condivise: sarà quindi necessario aggiungere dei multiplexer e, in alcuni casi, espandere quelli esistenti. Ad esempio, poiché si utilizza una sola memoria per istruzioni e dati, è

necessario un multiplexer per selezionare la provenienza degli indirizzi della memoria tra le due sorgenti possibili: PC (per leggere istruzioni) e ALUOut (per i dati).

Avendo sostituito le tre ALU dell'unità di elaborazione a singolo ciclo con una sola ALU, quest'ultima dovrà ricevere tutti gli ingressi che erano convogliati alle diverse unità. Per trattare questi ingressi addizionali, l'unità di elaborazione è stata modificata come segue:

1. Si è aggiunto un nuovo multiplexer sul primo ingresso della ALU, che seleziona il registro A oppure PC.
2. Il multiplexer a due vie presente sul secondo ingresso della ALU è sostituito da uno a quattro vie: i due ingressi aggiuntivi sono la costante 4 (per incrementare PC) ed il valore del campo offset, esteso e scalato (per il calcolo dell'indirizzo di salto).

La figura 5.31 riporta i dettagli dell'unità di elaborazione completa di questi multiplexer aggiuntivi: aggiungendo pochi registri e multiplexer si è potuto ridurre il numero di unità di memoria da due ad una, ed eliminare due sommatori. Essendo i registri ed i multiplexer relativamente piccoli, la modifica può ridurre in modo non indifferente il costo dell'hardware.

Poiché l'unità di elaborazione di figura 5.31 richiede diversi cicli di clock per istruzione, i segnali di controllo saranno necessariamente differenti: gli elementi di stato visibili al programmatore (il Program Counter, la memoria ed i registri) ed il registro IR necessitano di segnali di controllo per la scrittura, mentre la memoria richiede anche un segnale di controllo per la lettura. Risulta altresì possibile riutilizzare l'unità di controllo della ALU progettata per l'unità di elaborazione a singolo ciclo (si vedano la figura 5.15 e l'Appendice C). Infine, ciascun multiplexer a due vie richiede un singolo segnale di controllo, mentre il multiplexer a quattro vie ne richiede due: la figura 5.32 riporta l'unità di elaborazione di figura 5.31, dopo l'aggiunta dei segnali di controllo.

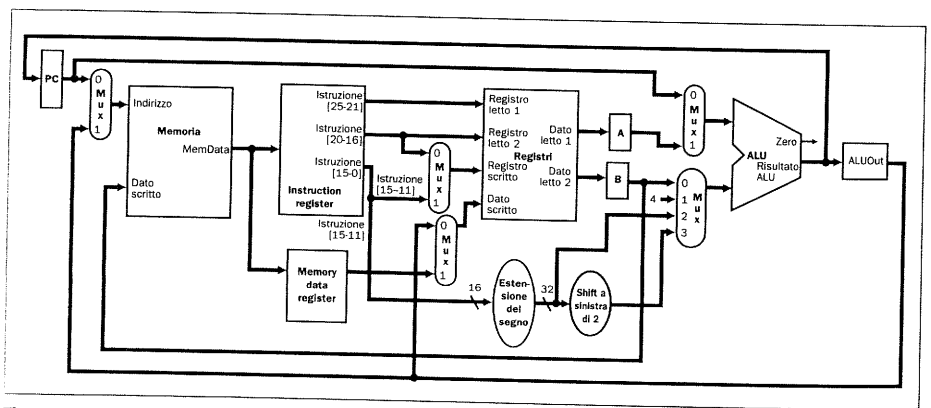


Figura 5.31 L'unità di elaborazione multi-ciclo che gestisce le principali istruzioni MIPS. Mentre questa unità di elaborazione supporta il normale incremento di PC, saranno richieste alcune interconnessioni aggiuntive ed un multiplexer per implementare i salti condizionati e incondizionati, esaminati tra breve. Rispetto all'unità di elaborazione a singolo ciclo, le aggiunte comprendono diversi registri (IR, MDR, A, B, ALUOut), un multiplexer per l'indirizzo di memoria, un multiplexer per l'ingresso superiore della ALU e la trasformazione del multiplexer sull'ingresso inferiore della ALU in uno a quattro vie; queste piccole aggiunte permettono di eliminare due sommatori e un'unità di memoria.

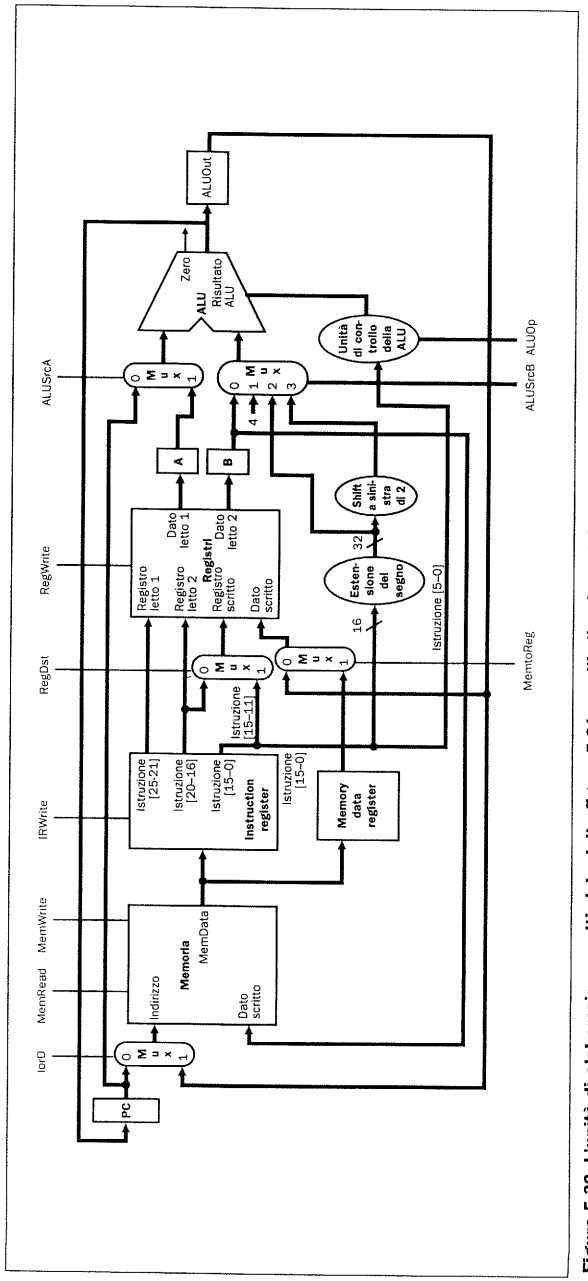


Figura 5.32 L'unità di elaborazione multi-ciclo della figura 5.31 con l'indicazione dei segnali di controllo. I segnali ALUOp e ALUSrcB sono segnali di controllo a 2 bit, mentre tutti gli altri sono segnali di 1 bit; i registri A e B non richiedono un segnale di scrittura in quanto il loro contenuto viene letto solo nel ciclo immediatamente successivo a quello in cui è scritto. Il Memory Data Register è stato aggiunto per parcheggiare il dato letto dalla memoria durante una load: il dato non può essere scritto direttamente nel register file perché il ciclo di clock non è sufficiente a contenere sia il tempo richiesto dall'accesso in memoria sia quello della scrittura nel register file. Il segnale MemRead è stato spostato verso l'alto per semplificare la figura; la circuiteria richiesta per i salti sarà aggiunta tra breve.

L'unità di elaborazione multi-ciclo necessita ancora delle aggiunte necessarie a supportare i salti condizionati e incondizionati; in seguito si vedrà come le istruzioni vengono sequenzializzate e come generare la logica di controllo.

Aggiungendo le istruzioni branch e jump, vi sono tre possibili sorgenti per il valore da scrivere in PC:

1. L'uscita della ALU, che vale PC+4 durante il reperimento dell'istruzione; questo valore viene scritto direttamente dentro il registro PC.
2. Il registro ALUOut, che a calcolo effettuato conterrà l'indirizzo del salto condizionato.
3. I 26 bit meno significativi dell'Instruction Register (IR), scalati di due posizioni a sinistra e concatenati con i 4 bit superiori di PC incrementato, da utilizzare quando l'istruzione è una jump.

Come già osservato a proposito dell'unità di elaborazione a singolo ciclo, PC viene scritto sia in modo incondizionato che condizionato: durante i normali incrementi e le jump PC è scritto incondizionatamente, ma se l'istruzione è un salto condizionato, PC incrementato è sostituito dal valore presente in ALUOut solamente se i due registri selezionati sono uguali. Questo problema verrà risolto grazie a due segnali di controllo per la scrittura di PC, chiamati PCWrite e PCWriteCond.

Questi due segnali devono essere collegati alla logica di controllo per la scrittura di PC: come già fatto nell'implementazione a singolo ciclo, si utilizzeranno alcune porte logiche per ricavare il segnale di controllo per la scrittura del registro PC a partire da PCWrite, PCWriteCond e dall'uscita Zero della ALU, utilizzata per stabilire se i due registri, operandi della `beq`, sono uguali. Nel caso dei salti condizionati, per determinare se PC debba essere scritto, si collegano con una porta AND il segnale Zero della ALU e PCWriteCond; l'uscita della AND è a sua volta messa in OR con PCWrite, che è il segnale di controllo per la scrittura incondizionata: l'uscita della porta OR viene collegata al segnale di controllo della scrittura del registro PC.

La figura 5.33 riporta l'unità di elaborazione multi-ciclo completa e la relativa unità di controllo, compresi i segnali di controllo aggiuntivi ed il multiplexer per implementare l'aggiornamento di PC.

Prima di esaminare in dettaglio i passi necessari per eseguire ciascuna istruzione, è conveniente esaminare informalmente gli effetti dei segnali di controllo (come già fu fatto per il progetto a singolo ciclo in figura 5.18): la figura 5.34 mostra l'operazione specificata da ciascun segnale di controllo quando affermato e quando non affermato.

Approfondimento: Per ridurre il numero di segnali che connettono le diverse unità funzionali, i progettisti spesso utilizzano dei *bus condivisi*: un bus condiviso è un insieme di linee che connettono diverse unità; nella maggior parte dei casi, comprendono le diverse sorgenti che possono porre dei dati sul bus e le diverse destinazioni che possono leggere tali valori. In modo analogo alla riduzione del numero di unità funzionali nell'unità di elaborazione, è possibile ridurre il numero di bus che connettono le unità, condividendoli: ad esempio, vi sono sei sorgenti di dati che raggiungono la ALU, ma solo due di esse sono utilizzate in ogni istante, per cui due bus sono sufficienti per trasportare i valori che devono raggiungere la ALU. Anziché porre un grande multiplexer di fronte alla ALU, un progettista potrebbe utilizzare un bus condiviso, a patto di garantire che in ogni istante di tempo solo una delle sorgenti stia inviando dei dati verso il bus. Questa tecnica può permettere di risparmiare delle linee di dato, ma sarà necessario un pari numero di linee di controllo per determinare quale valore venga scritto sul bus. La controindicazione principale di questa tecnica è una possibile riduzione delle prestazioni, poiché difficilmente un bus sarà veloce quanto una connessione da punto a punto.

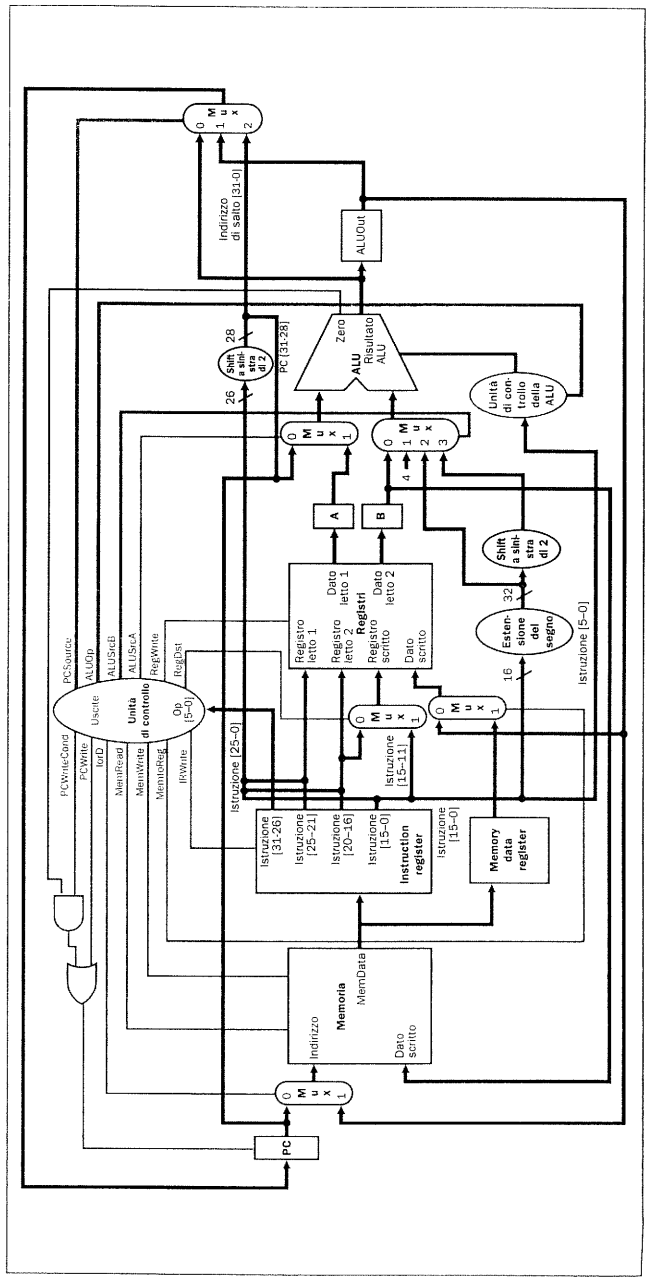


Figura 5.33 L'unità di elaborazione completa ed i relativi segnali di controllo nell'implementazione multi-ciclo. I segnali di controllo della figura 5.32 sono connessi all'unità di controllo e sono stati inclusi tutti gli elementi necessari per la modifica di PC. Le aggiunte principali rispetto alla figura 5.32 sono il multiplexer utilizzato per selezionare la fonte del nuovo valore di PC (in alto a destra), due porte logiche utilizzate per combinare i segnali di scrittura di PC (in alto a sinistra) ed i segnali di controllo PCSource, PCWrite e PCWriteCond. Il segnale PCWriteCond è posto in AND con l'uscita Zero della ALU per decidere se il salto condizionato debba essere eseguito; il risultato è posto in OR con il segnale di controllo PCWrite per generare l'effettivo segnale di controllo per la scrittura di PC. Inoltre, l'uscita di IR è manipolata in modo da inviare i 26 bit inferiori (l'indirizzo del salto incondizionato) alla logica utilizzata per la scelta del nuovo PC: i 26 bit sono scalati a sinistra di due posizioni, aggiungendo due bit pari a 0 nella parte bassa ed i 28 bit risultanti sono concatenati con i 4 bit più significativi di PC già incrementato.

Azioni dei segnali di controllo di 1 bit

Nome del segnale	Effetto quando non affermato	Effetto quando affermato
RegDst	Il numero del registro destinazione per Registro scritto proviene dal campo rt.	Il numero del registro destinazione per Registro scritto proviene dal campo rd.
RegWrite	Nessuno.	Nel registro specificato sull'ingresso Registro scritto è scritto il valore presente sull'ingresso Dato scritto.
ALUSrcA	Il primo operando della ALU è PC.	Il primo operando della ALU è il registro A.
MemRead	Nessuno.	Il contenuto della cella di memoria determinata dall'ingresso Indirizzo è posto sull'uscita MemData.
MemWrite	Nessuno.	Il contenuto della cella di memoria determinata dall'ingresso Indirizzo è sostituito dal valore presente sull'ingresso Dato scritto.
MemtoReg	Il valore inviato all'ingresso Dato scritto dei registri proviene da ALUOut.	Il valore inviato all'ingresso Dato scritto dei registri proviene da MDR.
IorD	L'indirizzo fornito alla memoria proviene da PC.	L'indirizzo fornito alla memoria proviene da ALUOut.
IRWrite	Nessuno.	L'uscita della memoria viene scritta in IR.
PCWrite	Nessuno.	PC viene scritto; la provenienza del valore è controllata da PCSource.
PCWriteCond	Nessuno.	PC viene scritto se anche l'uscita Zero della ALU è attiva.

Azioni dei segnali di controllo di 2 bit

Nome del segnale	Valore	Effetto
ALUOp	00	La ALU calcola una somma.
	01	La ALU calcola una sottrazione.
	10	La ALU calcola l'operazione determinata dal campo funct dell'istruzione.
ALUSrcB	00	Il secondo ingresso della ALU proviene dal registro B.
	01	Il secondo ingresso della ALU è la costante 4.
	10	Il secondo ingresso della ALU è il valore dei 16 bit meno significativi di IR, estesi a 32 bit con segno.
	11	Il secondo ingresso della ALU è il valore dei 16 bit meno significativi di IR, estesi a 32 bit con segno e scalati a sinistra di 2 bit.
PCSource	00	In PC viene scritta l'uscita della ALU (PC+4).
	01	In PC viene scritto il contenuto di ALUOut (l'indirizzo di destinazione del salto condizionato).
	10	In PC viene scritto l'indirizzo di destinazione del salto incondizionato (IR[25-0] scalato a sinistra di 2 bit e concatenato con PC+4[31-28]).

Figura 5.34 Azioni causate dai valori dei segnali di controllo della figura 5.33. La tabella superiore descrive i segnali di controllo di 1 bit, mentre quella inferiore riporta i segnali di controllo di 2 bit; solamente i segnali di controllo che pilotano i multiplexer possono avere effetto quando non affermati. Queste informazioni sono simili a quelle di figura 5.18 per l'unità di elaborazione a singolo ciclo, ma vi sono parecchi nuovi segnali (IRWrite, PCWrite, PCWriteCond, ALUSrcB e PCSource), mentre altri sono stati eliminati o sostituiti (PCSrc, Branch e Jump).

Suddividere l'esecuzione delle istruzioni in cicli di clock

Data l'unità di elaborazione della figura 5.33, per determinare quali nuovi segnali di controllo dovranno essere introdotti ed indicare il valore che i segnali di controllo devono assumere, si deve esaminare che cosa deve accadere in ciascun ciclo di clock dell'esecuzione multi-ciclo. L'obiettivo nel decomporre l'esecuzione delle operazioni in cicli di clock dovrebbe essere quello di bilanciare la quantità di lavoro svolta in ciascun ciclo, in modo da minimizzare il periodo di clock. Si può iniziare suddividendo l'esecuzione di ciascuna istruzione in una serie di passi, ciascuno corrispondente ad un ciclo di clock, che siano pressappoco della stessa lunghezza: ad

esempio, si potrebbe vincolare ciascun passo a contenere al più un'operazione con la ALU, un accesso al register file o un accesso alla memoria. Con questo vincolo il ciclo di clock dovrebbe risultare non più lungo della più lenta di queste operazioni.

Si ricordi che alla fine di ciascun ciclo di clock tutti i valori richiesti nel ciclo successivo dovranno essere memorizzati in un registro, che può essere uno degli elementi di stato principali (ad esempio PC, il register file o la memoria), un registro temporaneo scritto ad ogni ciclo di clock (ad esempio A, B, MDR o ALUOut), oppure un registro temporaneo dotato di segnale di controllo della scrittura (ad esempio IR). Si ricordi anche che, essendo il circuito sensibile ai fronti, si leggerà sempre il valore corrente dei registri, in quanto il nuovo valore apparirà solo al ciclo di clock successivo.

Nel caso dell'unità di elaborazione a singolo ciclo, durante l'esecuzione ciascuna istruzione utilizza un insieme di elementi, di cui molti lavorano in serie, utilizzando come ingresso l'uscita di un altro elemento; altri, invece, operano in parallelo: per esempio, l'incremento di PC e la lettura dell'istruzione sono contemporanee. Una situazione analoga si verifica anche nell'unità di elaborazione multi-ciclo: tutte le operazioni attribuite ad un certo passo vengono eseguite in parallelo in un ciclo di clock, mentre i passi successivi vengono eseguiti in serie in diversi cicli di clock. La limitazione ad una sola operazione di tipo ALU, accesso alla memoria o accesso al register file determina ciò che può essere contenuto nello stesso passo.

Si noti che verranno distinte la lettura e la scrittura di PC o degli altri registri individuali rispetto alle letture e alle scritture nel register file: mentre nel primo caso la lettura o la scrittura è interna al ciclo di clock, per leggere o scrivere un dato nel register file è necessario un ciclo di clock aggiuntivo. Il motivo di questa distinzione risiede nel fatto che il register file ha una logica di controllo ed un overhead di accesso superiori rispetto ai singoli registri, e quindi l'obiettivo di ridurre il periodo del clock richiede di impiegare cicli di clock aggiuntivi per gli accessi al register file.

Nel seguito vengono riportati dei possibili passi di esecuzione con le relative azioni; ciascuna istruzione richiede da tre a cinque passi:

1. Passo di prelievo dell'istruzione

Si carica l'istruzione dalla memoria e si calcola l'indirizzo dell'istruzione successiva in ordine sequenziale:

```
IR = Memoria[PC] ;
PC = PC + 4 ;
```

Operazioni: Si utilizza PC come indirizzo di memoria, si esegue una lettura e si scrive l'istruzione nell'Instruction Register (IR), dove risiederà per i passi successivi; si incrementa altresì PC di 4. Per implementare questo passo dovranno essere affermati i segnali di controllo MemRead e IRWrite e dovrà essere posto IorD a 0 per selezionare PC come sorgente dell'indirizzo. L'incremento di PC, nello stesso passo, richiede di porre il segnale ALUSrcA a 0 (inviando PC alla ALU) e ALUSrcB a 01 (inviando 4 alla ALU), con ALUOp pari a 00 (per forzare una somma); infine occorrerà memorizzare nuovamente in PC l'indirizzo incrementato, affermando PCWrite. L'incremento di PC e l'accesso alla memoria possono avvenire in parallelo ed il nuovo valore di PC non sarà visibile fino al prossimo ciclo di clock (il valore incrementato di PC viene anche memorizzato in ALUOut, ma tale azione non provoca disfunzioni).

2. Passo di decodifica dell'istruzione e caricamento dei registri

In questo passo e nel precedente non si conosce ancora quale sia l'istruzione, quindi si possono compiere solamente azioni applicabili a tutte le istruzioni (come prelevare l'istruzione nel passo 1) o che non siano dannose, nel caso in cui l'istruzione non sia quella ipotizzata. Quindi in questo passo si possono leggere i due registri indicati dai campi rs e rt dell'istruzione, poiché non è dannoso leggerli anche nel caso in cui non siano richiesti: i valori letti dal register file potrebbero servire in passi successivi, quindi vengono letti dal register file e memorizzati nei registri temporanei A e B.

Viene anche calcolato l'indirizzo di destinazione dei salti condizionati utilizzando la ALU, il che non è dannoso dal momento che si può ignorare tale valore nel caso in cui l'istruzione non sia branch. Il potenziale indirizzo di destinazione viene memorizzato in ALUOut.

Anticipare "ottimisticamente" l'esecuzione di queste operazioni ha il beneficio di ridurre il numero di cicli di clock richiesti per eseguire un'istruzione; esse possono essere eseguite così presto grazie alla regolarità del formato delle istruzioni: ad esempio, se l'istruzione ha due registri in ingresso, questi sono sempre nei campi rs e rt, mentre se l'istruzione è branch, l'offset è sempre nei 16 bit meno significativi:

```
A = Reg[IR[25-21]] ;
B = Reg[IR[20-16]] ;
ALUOut = PC + (sign-extend(IR[15-0]) << 2) ;
```

Operazioni: Si accede al register file per leggere i registri rs e rt e memorizzare il risultato nei registri A e B; siccome A e B sono sovrascritti ad ogni ciclo di clock, il register file deve essere letto ad ogni ciclo. In questo passo si calcola anche l'indirizzo di destinazione dei salti condizionati e lo si memorizza in ALUOut, da dove verrà prelevato nel ciclo di clock successivo se l'istruzione è branch: ciò richiede di portare ALUSrcA a 0 (inviando PC alla ALU), ALUSrcB ad 11 (inviando il campo offset, scalato ed esteso, alla ALU) ed ALUOp a 00 (specificando una somma). L'accesso al register file ed il calcolo dell'indirizzo del salto avvengono in parallelo.

Terminato questo ciclo di clock, la prossima azione dipende dal valore dell'istruzione.

3. Esecuzione, calcolo dell'indirizzo di memoria o completamento del salto

Questo è il primo passo in cui l'operazione compiuta dall'unità di elaborazione è determinata dalla classe di istruzione: la ALU elabora gli operandi preparati nei passi precedenti svolgendo una funzione tra le tre possibili, a seconda della classe di istruzione. Si possono descrivere le azioni svolte per le diverse classi di istruzioni come segue:

Accesso alla memoria:

```
ALUOut = A + sign-extend(IR[15-0]) ;
```

Operazioni: La ALU somma gli operandi in modo da ottenere l'indirizzo della memoria: ciò richiede di porre ALUSrcA ad 1 (inviando il registro A al primo ingresso della ALU) e ALUSrcB a 10 (inviando sul secondo ingresso della ALU l'uscita dell'unità di estensione del segno); il segnale ALUOp viene portato a 00 (facendo sì che la ALU sommi).

Istruzione logico-aritmetica (tipo-R):

```
ALUOut = A op B ;
```

Operazioni: La ALU esegue l'operazione specificata dal codice di funzione sui due valori letti dal register file nel ciclo precedente: ciò richiede i valori ALUSrcA=1 e ALUSrcB=00 (la cui combinazione invia i registri A e B agli ingressi della ALU), mentre il segnale ALUOp viene portato a 10 (in modo che i segnali di controllo della ALU vengano determinati a partire dal campo funct).

Salto condizionato (branch):

```
if (A==B) PC = ALUOut ;
```

Operazioni: Si utilizza la ALU per determinare l'uguaglianza tra i due registri letti nel passo precedente: il segnale Zero della ALU verrà utilizzato per determinare se si debba eseguire il salto. In questo caso si pone ALUSrcA=1 e ALUSrcB=00 (mandando i valori provenienti dal register file alla ALU), con il segnale ALUOp posto a 01 (facendo sì che la ALU sottragga) per controllare l'eguaglianza. Il segnale PCCondWrite deve essere affermato per aggiornare PC nel caso in cui l'uscita Zero della ALU sia affermata; portando PCSource a 01, il valore scritto in PC proviene da ALUOut, che contiene l'indirizzo di destinazione del salto calcolato nel ciclo precedente. Per i salti condizionati che vengono eseguiti, in pratica PC è scritto due volte: la prima a partire dall'uscita della ALU (nel passo di decodifica dell'istruzione e caricamento dei registri) ed una a partire da ALUOut (durante il passo di completamento del salto): il valore scritto per ultimo è quello che verrà utilizzato nella successiva fase di reperimento dell'istruzione.

Salto incondizionato (jump):

```
PC = PC[31-28] || (IR[25-0]<<2) ;
```

Operazioni: Il valore di PC è sostituito dall'indirizzo del salto: PCSource è pilotato in modo da inviare l'indirizzo del salto a PC, e PCWrite viene affermato per scrivere l'indirizzo in PC.

4. Passo di accesso alla memoria o completamento dell'istruzione di tipo-R

In questo passo le istruzioni load o store accedono alla memoria e le istruzioni logico-aritmetiche scrivono il risultato. Se viene letto un valore dalla memoria, questo è posto nel Memory Data Register (MDR), dal quale sarà utilizzato nel ciclo successivo.

Accesso alla memoria:

```
MDR = Memoria [ALUOut] ;
```

oppure

```
Memoria [ALUOut] = B ;
```

Operazioni: Se l'istruzione è una load, si legge una parola di dato dalla memoria e la si scrive nel MDR; se è una store, il dato viene scritto in memoria. In entrambi i casi si utilizza l'indirizzo calcolato nel passo precedente e memorizzato in ALUOut. Per una store, l'operando sorgente è memorizzato in B (in realtà B è stato letto due volte, una nel passo 2 e l'altra nel passo 3; fortunatamente, nei due casi è stato letto lo stesso valore, dal momento che il numero di registro, memorizzato in IR, non è cambiato). Si afferma il segnale MemRead per una load e MemWrite per una store;

inoltre, per le load, il segnale IorD è posto ad 1 in modo da usare l'indirizzo di memoria proveniente dalla ALU anziché quello proveniente da PC. Siccome MDR viene comunque scritto ad ogni ciclo, non richiede alcun segnale di controllo.

Istruzione logico-aritmetica (tipo-R):

```
Reg[IR[15-11]] = ALUOut ;
```

Operazioni: Si pone il contenuto di ALUOut, corrispondente al risultato dell'operazione della ALU del ciclo precedente, nel registro rd. Il segnale RegDst deve essere posto ad 1 (forzando l'utilizzo del campo rd (bit 15-11) nella selezione della cella del register file in cui scrivere), e RegWrite deve essere affermato, mentre MemtoReg è posto a 0 (in modo da scrivere l'uscita della ALU anziché i dati provenienti dalla memoria).

5. Passo di completamento della lettura da memoria

In questo passo vengono completate le istruzioni load scrivendo il valore proveniente dalla memoria.

Load:

```
Reg[IR[20-16]] = MDR ;
```

Operazioni: Si scrive il dato letto, stato memorizzato in MDR nel ciclo precedente, all'interno del register file: a tale scopo, si pone MemtoReg=1 (per scrivere il dato proveniente dalla memoria), si afferma RegWrite (causando una scrittura) e si pone RegDst=0 per scegliere il campo rt (bit 20-16) come numero di registro.

La sequenza di cinque passi è riassunta in figura 5.35; da tale sequenza è possibile dedurre le azioni dell'unità di controllo in ciascun ciclo di clock.

Definizione dell'unità di controllo

A questo punto, avendo determinato quali siano i segnali di controllo e quando debbano assumere valori affermati, si può procedere all'implementazione dell'unità di controllo. Nel progettare l'unità di controllo per l'unità operativa a singolo ciclo si era utilizzato un insieme di tabelle di verità che specificavano i valori dei segnali di controllo in funzione della classe di istruzioni; l'unità di controllo dell'unità operativa multi-ciclo è più complessa, poiché ciascuna istruzione viene eseguita in più passi: l'unità di controllo deve dunque specificare sia i segnali da affermare in ciascuno dei passi sia il passo successivo della sequenza.

In questo paragrafo e nel successivo Paragrafo 5.5 si analizzeranno due tecniche di specifica dell'unità di controllo: la prima tecnica è basata su macchine a stati finiti che vengono solitamente rappresentate in modo grafico; la seconda tecnica, detta *microprogrammazione*, utilizza per l'unità di controllo una rappresentazione simile ad un programma. Entrambe le tecniche rappresentano l'unità di controllo in una forma che permette la sintesi automatica dell'implementazione dettagliata (sotto forma di porte logiche, ROM o PLA) da parte dei sistemi CAD. In questo capitolo ci si concentrerà sul progetto dell'unità di controllo e sulla sua rappresentazione nelle due forme; per il lettore interessato alle modalità di trasformazione di queste specifiche nell'hardware effettivo, l'Appendice C continua lo sviluppo di questo capitolo traducendo l'unità di controllo multi-ciclo in un'implementazione

Nome del passo	Azione per istruzioni di tipo-R	Azione per istruzioni di accesso alla memoria	Azione per salti condizionati	Azione per salti incondizionati
Prelievo dell'istruzione	IR = Memoria[PC] PC = PC+4			
Decodifica dell'istruzione/ caricamento dei registri	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUOut = PC+(sign-extend(IR[15-0])<<2)			
Esecuzione, calcolo dell'indirizzo, completamento dei salti	ALUOut = A op B	ALUOut = A+sign-extend (IR[15-0])	if (A == B) then PC = ALUOut	PC = PC[31-28] ll (IR[25-0]<<2)
Accesso alla memoria o completamento dell'istruzione di tipo-R	Reg [IR[15-11]] = ALUOut	Load: MDR = Memoria[ALUOut] oppure Store: Memoria[ALUOut] = B		
Completamento della lettura da memoria		Load: Reg[IR[20-16]] = MDR		

Figura 5.35 Schema riassuntivo dei passi intrapresi durante l'esecuzione delle diverse classi di istruzioni. Le istruzioni possono richiedere da tre a cinque passi di esecuzione: i primi due passi sono indipendenti dalla classe di istruzioni, mentre in quelli successivi l'istruzione richiede da uno a tre cicli aggiuntivi per essere completata, a seconda della classe. Le caselle vuote nei passi di accesso alla memoria e di completamento della lettura indicano che le classi di istruzioni corrispondenti richiedono un numero minore di cicli: in un'implementazione multi-ciclo, si inizia l'esecuzione di una nuova istruzione non appena l'istruzione corrente viene completata, quindi tali cicli non sono di attesa e non vengono sprecati. Come già citato, il register file di fatto viene letto ad ogni ciclo, ma finché IR non cambia i valori letti sono sempre uguali; in particolare, il valore letto nel registro B durante lo stadio di decodifica dell'istruzione per le istruzioni branch o di tipo-R è uguale al valore che B assume durante il passo di esecuzione e che viene usato nell'accesso alla memoria per l'istruzione store word.

hardware dettagliata. Le idee principali sull'unità di controllo si possono intuire da questo capitolo senza esaminare il materiale dell'Appendice C, ma per coloro che vogliono "spaccare il bit", l'Appendice C spiega come fare!

Il primo dei metodi utilizzati per specificare l'unità di controllo multi-ciclo fa uso di *macchine a stati finiti* (*finite state machine*, FSM): esse consistono di un insieme di stati e di istruzioni su come cambiare stato. Le istruzioni sono definite tramite una *funzione di stato futuro*, che mette in corrispondenza lo stato presente e gli ingressi con lo stato futuro. Nell'utilizzo delle macchine a stati finiti per il progetto delle unità di controllo, ciascuno stato specifica inoltre l'insieme delle uscite che vengono affermate quanto la macchina si trova in quello stato; nell'implementazione della macchina si assume normalmente che le restanti uscite siano non affermate: il corretto funzionamento dell'unità di elaborazione dipende dal fatto che i segnali non esplicitamente affermati siano effettivamente non affermati, anziché indifferenti. Ad esempio, il segnale RegWrite deve essere affermato solo quando si vuole scrivere un elemento del register file: quando non esplicitamente affermato, tale segnale deve essere pari a zero per evitare scritture indesiderate.

I segnali di controllo dei multiplexer sono trattati diversamente, perché selezionano uno degli ingressi sia che valgano 0 sia che valgano 1: nella macchina a stati finiti, quindi, verranno sempre specificati i valori di tutti gli ingressi di controllo dei multiplexer che non abbiano valore indifferente. In alcuni tipi di implementazione dell'unità di controllo, la forzatura di un segnale a 0 potrebbe essere un'azione automatica e non richiedere alcuna porta logica: un esempio di questo tipo di macchine a stati finiti è riportato nell'Appendice B e si invitano i lettori che non abbiano sufficiente familiarità con il concetto di macchina a stati finiti a consultare l'Appendice B prima di proseguire nella lettura.

L'unità di controllo a stati finiti corrisponde essenzialmente ai cinque passi di esecuzione mostrati nelle pagine precedenti, dove ciascun passo della macchina a stati

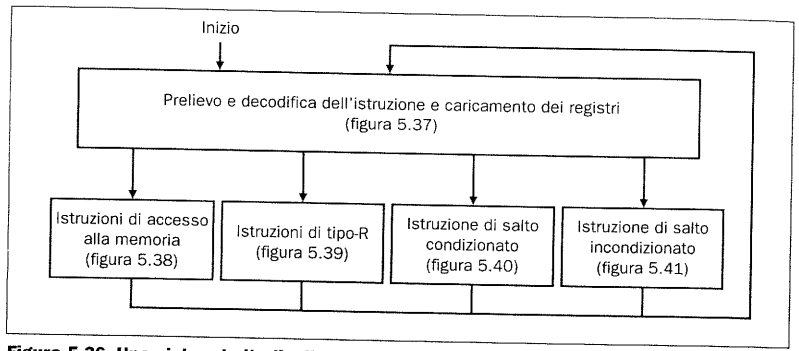


Figura 5.36 Una vista ad alto livello dell'unità di controllo come macchina a stati finiti. I primi passi sono indipendenti dalla classe dell'istruzione, in seguito il completamento delle istruzioni delle diverse classi richiede delle sequenze dipendenti dal codice operativo; dopo aver terminato le azioni richieste per le diverse classi di istruzioni, l'unità di controllo inizia a prelevare una nuova istruzione. Ciascun rettangolo della figura rappresenta uno o più stati; l'arco etichettato con *Inizio* indica lo stato in cui inizia l'esecuzione al momento di prelevare la prima istruzione.

finiti corrisponde ad un ciclo di clock. La macchina a stati finiti consterà di diverse parti: dato che i primi due passi di esecuzione sono identici in tutte le istruzioni, i primi due stati della macchina a stati finiti saranno comuni a tutte le istruzioni; i passi da 3 a 5 dipendono invece dal codice operativo. Al termine dell'esecuzione dell'ultimo passo di una particolare classe di istruzioni, la macchina a stati finiti dovrà ritornare allo stato iniziale per iniziare a prelevare l'istruzione successiva.

La figura 5.36 riporta una rappresentazione astratta di tale macchina a stati finiti; per completarne i dettagli, si espanderà per prima la parte relativa al prelievo dell'istruzione e alla decodifica, mentre in seguito si esamineranno gli stati e le azioni relativi alle diverse classi di istruzioni.

I primi due stati della macchina a stati finiti sono riportati in figura 5.37 utilizzando la tradizionale rappresentazione grafica: gli stati sono numerati per semplificare l'esposizione, sebbene i numeri siano arbitrari; lo stato 0, corrispondente al passo 1, è lo stato iniziale della macchina.

I segnali affermati in ciascuno stato sono indicati all'interno del cerchio che lo rappresenta; gli archi tra gli stati definiscono lo stato futuro e, nel caso in cui ne siano possibili molti, sono etichettati con le condizioni che selezionano uno specifico stato futuro. Dopo lo stato 1, i segnali affermati dipendono dalla classe dell'istruzione, pertanto la macchina a stati finiti ha quattro archi uscenti dallo stato 1, corrispondenti alle quattro classi: accesso alla memoria, tipo-R, branch on equal e jump. L'azione di portarsi in stati diversi in funzione del tipo di istruzione è detto *decodifica*, in quanto la scelta dello stato futuro, e quindi le azioni che ne seguiranno, dipende dalla classe dell'istruzione.

La figura 5.38 riporta la porzione di macchina a stati finiti utilizzata nell'implementazione delle istruzioni di accesso alla memoria, in cui nel primo stato, dopo il prelievo dell'istruzione e dei registri, viene calcolato l'indirizzo di memoria (stato 2). Per calcolare tale indirizzo occorrono dei valori sui multiplexer in ingresso alla ALU tali da far sì che sul primo ingresso vada il registro A e che il secondo ingresso sia ottenuto tramite estensione del segno dal campo di spiazzamento; il risultato è scritto nel registro ALUOut. Dopo il calcolo dell'indirizzo di memoria, quest'ultima deve essere letta o scritta, il che richiede due stati: se il codice operati-

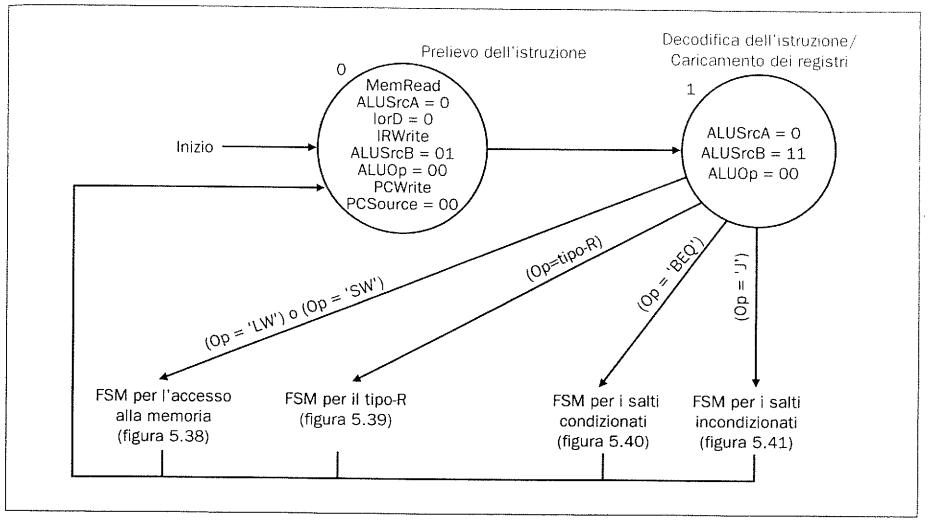


Figura 5.37 La porzione di prelievo e decodifica dell'istruzione è uguale per tutte le istruzioni. Gli stati riportati in questa figura corrispondono al rettangolo superiore della macchina a stati finiti astratta della figura 5.36. Nel primo stato si affermano due segnali per leggere un'istruzione dalla memoria e scriverla nell'Instruction Register (MemRead e IRWrite) e si pone IorD a 0 per scegliere PC come sorgente dell'indirizzo. I valori dei segnali ALUSrcA, ALUSrcB, ALUOp, PCWrite e PCSource fanno sì che PC+4 venga calcolato e memorizzato in PC (oltre a memorizzarlo in ALUOut, da cui però non verrà mai letto). Nello stato successivo si calcola l'indirizzo di destinazione del salto forzando ALUSrcB ad 11 (in modo che i 16 bit meno significativi, estesi e scalati, vengano inviati alla ALU), ALUSrcA a 0 ed ALUOp a 00: il risultato viene memorizzato nel registro ALUOut, il quale viene comunque scritto ad ogni ciclo di clock. Vi sono poi quattro stati futuri possibili, in funzione della classe di istruzione, informazione nota in questo stato: il valore di un ingresso dell'unità di controllo, detto Op, determina quale arco debba essere seguito.

vo dell'istruzione è `lw`, allora lo stato 3 (corrispondente al passo di accesso alla memoria) effettua la lettura (viene affermato MemRead); l'uscita della memoria è sempre scritta in MDR. Se l'opcode è invece `sw`, lo stato 5 effettua una scrittura in memoria (affermando MemWrite). Negli stati 3 e 5, il segnale IorD è posto ad 1 per forzare la provenienza dell'indirizzo di memoria dalla ALU (ciò non è richiesto per le istruzioni store, poiché l'indirizzo di scrittura utilizza un ingresso separato della memoria). Dopo l'esecuzione della scrittura, l'istruzione `sw` è completa e lo stato futuro è lo stato 0. Se l'istruzione è invece una load, è necessario un altro stato (stato 4) per scrivere il risultato proveniente dalla memoria nel register file: lo si ottiene forzando i segnali di controllo dei multiplexer come MemtoReg=1 e RegDst=0, in modo da inviare il valore di MDR al register file, utilizzando rt come numero di registro. Dopo tale stato, corrispondente al passo di completamento della lettura da memoria, lo stato futuro sarà lo stato 0.

L'implementazione delle istruzioni di tipo-R richiede due stati, corrispondenti ai passi 3 (esecuzione) e 4 (completamento dell'istruzione di tipo-R): la figura 5.39 riporta questa parte, composta da due stati, della macchina a stati finiti. Lo stato 6 afferma il segnale ALUSrcA e pone ALUSrcB a 00, facendo sì che i due registri letti dal register file vengano utilizzati come ingressi della ALU; viene altresì forzato ALUOp a 10, in modo da istruire l'unità di controllo della ALU ad utilizzare il campo funzione nella determinazione dei segnali di controllo della ALU. Nello

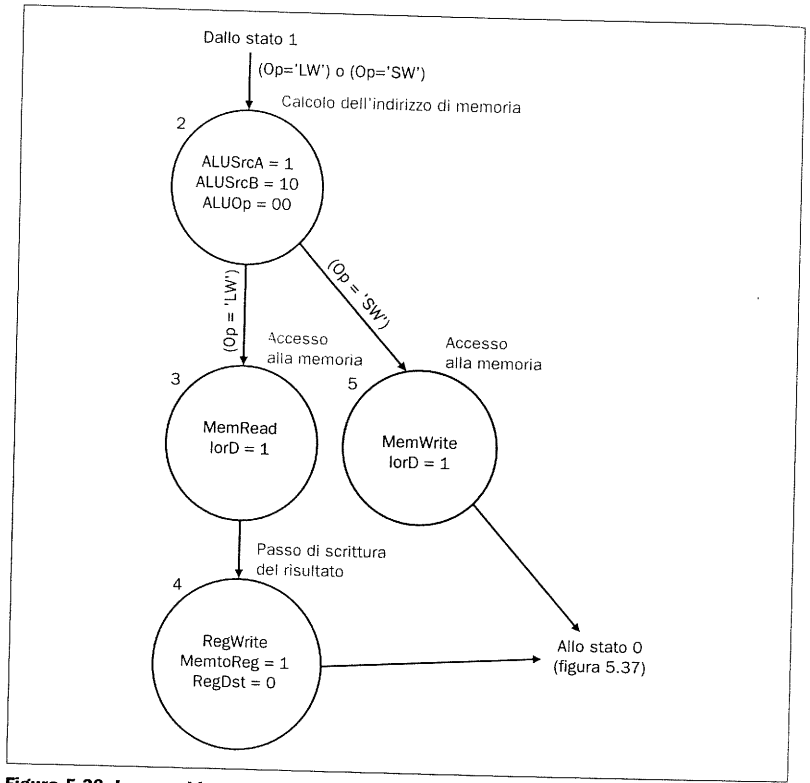


Figura 5.38 La macchina a stati finiti per il controllo delle istruzioni di accesso alla memoria ha quattro stati. Essi corrispondono al rettangolo denominato "Istruzioni di accesso alla memoria" della figura 5.36. Dopo il calcolo dell'indirizzo di memoria, che avviene nello stato 2 grazie ai segnali di controllo ALUSrcA, ALUSrcB e ALUOp, sono richieste sequenze diverse per le istruzioni load e store: le load richiedono uno stato aggiuntivo per copiare il risultato da MDR (in cui è stato scritto nello stato 3) al register file.

stato 7 viene affermato il segnale RegWrite per forzare una scrittura nel register file, viene affermato RegDst in modo da utilizzare il campo rd come numero di registro destinazione, mentre MemtoReg è non affermato, in modo da selezionare ALUOut come sorgente del valore che verrà scritto nel register file.

Nel caso dei salti condizionati, è necessario un solo stato aggiuntivo, poiché questi terminano l'esecuzione già al terzo passo: in questo stato (mostrato in figura 5.40), occorre forzare i segnali di controllo in modo da causare il confronto dei registri A e B e la scrittura condizionale di PC con l'indirizzo presente nel registro ALUOut. Il confronto richiede di affermare ALUSrcA e porre ALUSrcB a 00, ponendo ALUOp a 01 per forzare la sottrazione (verrà utilizzata solo l'uscita Zero della ALU e si scarterà il risultato della sottrazione); la scrittura di PC richiede invece di affermare PCWriteCond e di porre PCSource = 01, causando la scrittura del valore del registro ALUOut (che contiene l'indirizzo del salto calcolato nello stato 1, come da figura 5.37) in PC, a patto che l'uscita Zero della ALU sia affermata.

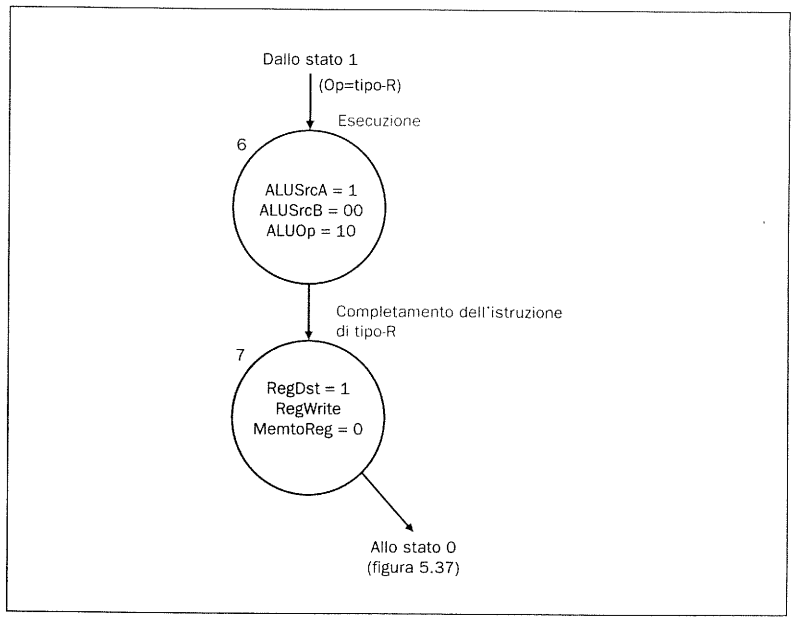


Figura 5.39 Le istruzioni di tipo-R si implementano con una semplice macchina con due stati. Essi corrispondono al rettangolo denominato "Istruzioni di tipo-R" della figura 5.36. L'operazione logico-aritmetica viene eseguita nel primo stato, mentre il secondo causa la scrittura del risultato (che si trova in ALUOut) nel register file: nello stato 7 i tre segnali di controllo affermati determinano la scrittura del contenuto di ALUOut nel registro del register file specificato dal campo rd dell'Instruction Register.

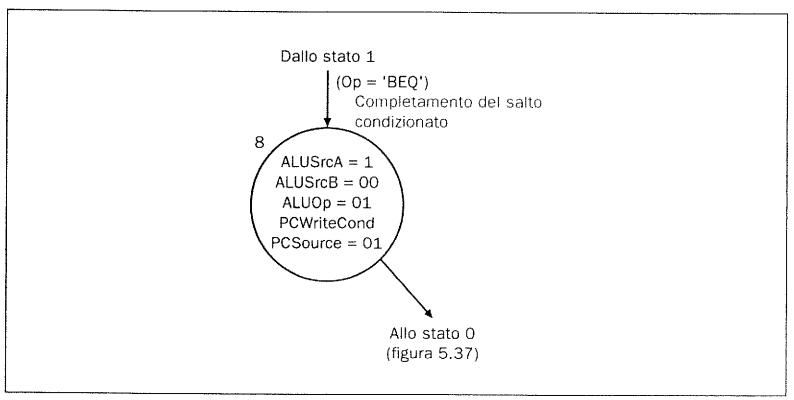


Figura 5.40 L'istruzione branch richiede un solo stato. Le prime tre uscite che vengono affermate in tale stato (ALUSrcA, ALUSrcB e ALUOp) specificano alla ALU di eseguire il confronto, mentre i segnali PCSource e PCWriteCond determinano la scrittura condizionale nel caso in cui la condizione di salto sia verificata. Si noti che non viene utilizzato il valore memorizzato nel registro ALUOut, ma direttamente l'uscita Zero della ALU; l'indirizzo di destinazione del salto è invece letto da ALUOut, dove era stato memorizzato al termine dello stato 1.

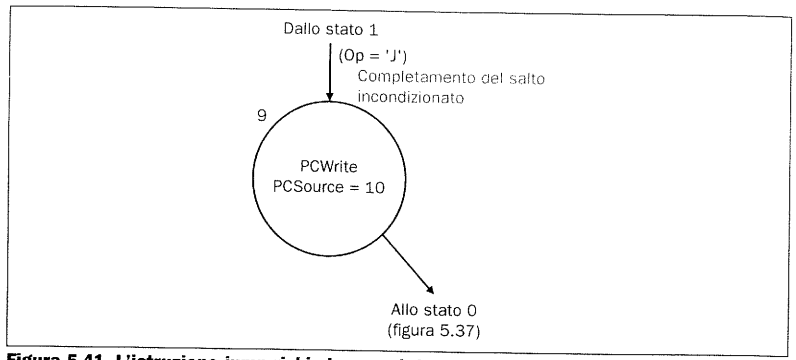


Figura 5.41 L'istruzione jump richiede uno stato per affermare i due segnali di controllo che scrivono in PC i 26 bit meno significativi dell'Instruction Register scalati a sinistra di 2 posizioni e concatenati ai 4 bit più significativi del PC dell'istruzione.

L'ultima classe di istruzioni è costituita dalla jump: come la branch, richiede un solo stato per il proprio completamento (figura 5.41), nel quale si afferma il segnale PCWrite per forzare la scrittura di PC e si pone PCSource a 10 in modo che il valore scritto sia costituito dai 26 bit dell'Instruction Register, a cui si concatenano 00_{due} come bit meno significativi ed i 4 bit più significativi di PC.

A questo punto è possibile riunire le parti della macchina a stati finiti in modo da fornire la specifica dell'unità di controllo, come riportato in figura 5.42. In ciascuno stato sono riportati i segnali affermati, mentre lo stato futuro dipende dai bit del codice operativo dell'istruzione, per cui gli archi sono etichettati con operazioni di confronto con i relativi opcode.

A partire da questa implementazione, e sapendo che ciascuno stato richiede 1 ciclo di clock, è possibile calcolare il valore di CPI per una combinazione di istruzioni tipica.

Calcolo di CPI nelle CPU multi-ciclo

Esempio Utilizzando l'unità di controllo riportata in figura 5.42 e la combinazione di istruzioni per il programma gcc riportata nell'esempio del Paragrafo 5.3, qual è il valore di CPI se si assume che ciascuno stato richieda un ciclo di clock?

Soluzione La combinazione è del 22% di load, 11% di store, 49% di istruzioni di formato-R, 16% di branch e 2% di jump; dalla figura 5.42 si deduce il numero di cicli di clock per ciascuna classe di istruzioni come segue:

- Load: 5
- Store: 5
- Istruzioni di formato-R: 4
- Salti condizionati: 3
- Salti incondizionati: 3

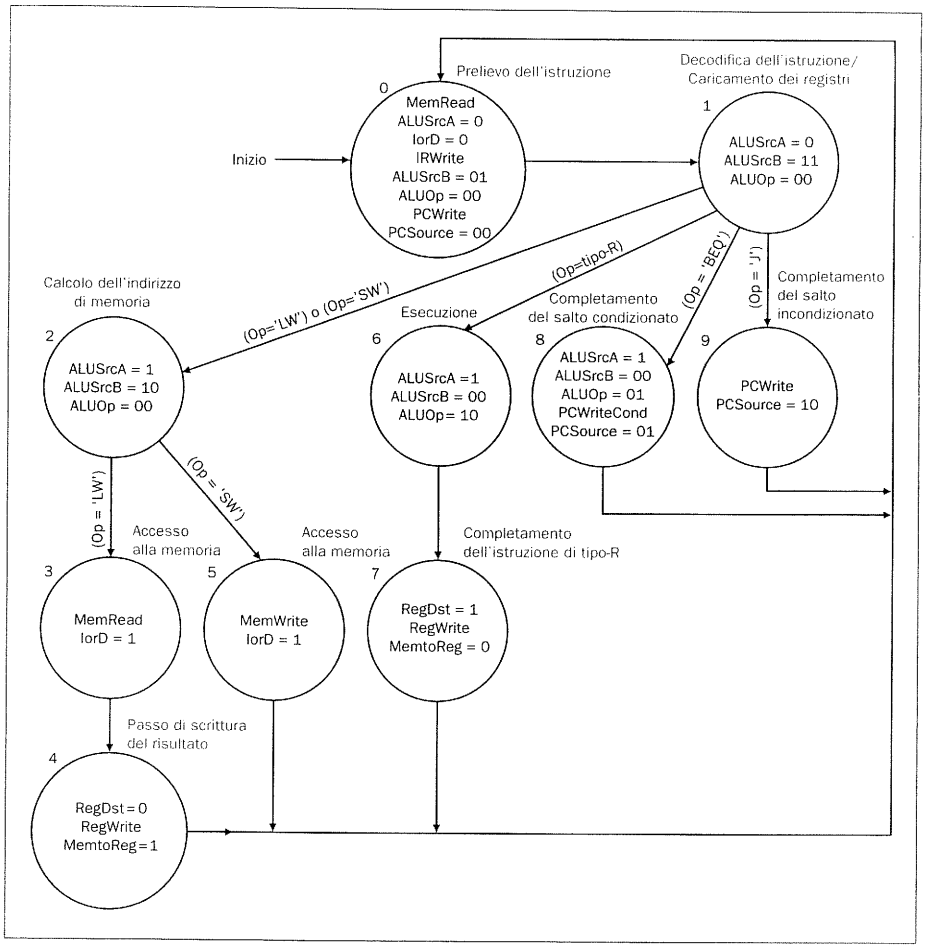


Figura 5.42 La macchina a stati finiti completa dell'unità di controllo relativa all'unità di elaborazione di figura 5.33. Le etichette sugli archi sono le condizioni da verificare nella determinazione dello stato futuro; nel caso in cui tale stato futuro non dipenda da una condizione, l'etichetta è assente. Le etichette interne ai nodi indicano invece i segnali di uscita dell'unità di controllo affermati durante lo stato corrispondente; gli ingressi dei multiplexer vengono sempre specificati quando la loro uscita è utilizzata, per cui in alcuni stati si troveranno ingressi di multiplexer posti a 0. Nell'Appendice C si vedrà come tradurre questa macchina a stati finiti in equazioni logiche e come implementare tali equazioni.

Il valore di CPI è dato dalle espressioni seguenti:

$$\text{CPI} = \frac{\text{cicli di clock della CPU}}{\text{numero di istruzioni}} = \frac{\sum \text{numero di istruzioni}_i \times \text{CPI}_i}{\text{numero di istruzioni}} =$$

$$= \sum \frac{\text{numero di istruzioni}_i}{\text{numero di istruzioni}} \times \text{CPI}_i$$

Il rapporto

$$\frac{\text{numero di istruzioni}_i}{\text{numero di istruzioni}}$$

non è altro che la frequenza delle istruzioni della classe i. Si possono quindi sostituire i valori ed ottenere

$$\begin{aligned} CPI &= 0,22 \times 5 + 0,11 \times 4 + 0,49 \times 4 + 0,16 \times 3 + 0,02 \times 3 = \\ &= 1,1 + 0,44 + 1,96 + 0,48 + 0,06 = 4,04 \end{aligned}$$

Questo valore di CPI è migliore di quello che si sarebbe ottenuto nel caso peggiore, cioè se tutte le istruzioni avessero richiesto lo stesso numero di cicli di clock (ossia 5).

Una macchina a stati finiti può essere implementata mediante un registro temporaneo che memorizza lo stato corrente ed un blocco di logica combinatoria che determina sia i segnali che devono essere affermati nell'unità di elaborazione, sia lo stato futuro; una rappresentazione di tale struttura è riportata in figura 5.43. L'Appendice C descrive in dettaglio come si possa implementare una macchina a stati finiti con questa struttura: nel Paragrafo C.3 viene implementata la logica combinatoria di controllo per la macchina a stati finiti di figura 5.42, sia tramite una memoria a sola lettura (ROM, Read-Only Memory), sia mediante una matrice logica programmabile (PLA, Programmable Logic Array). Tali elementi sono descritti in Appendice B. Il prossimo paragrafo propone invece una metodologia alternativa per rappresentare l'unità di controllo; le due tecniche esaminate sono semplicemente rappresentazioni alternative delle stesse informazioni di controllo.

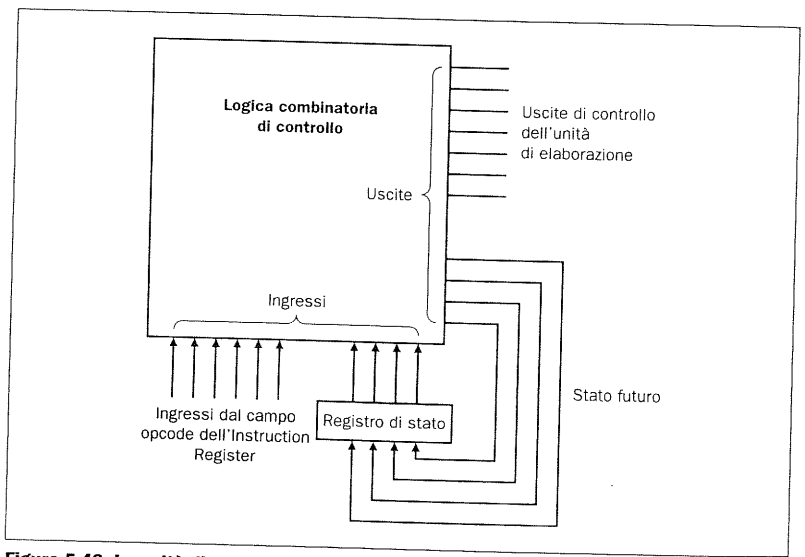


Figura 5.43 Le unità di controllo realizzate come macchine a stati finiti sono solitamente implementate utilizzando un blocco di logica combinatoria ed un registro per memorizzare lo stato corrente. Le uscite della logica combinatoria sono il numero del prossimo stato ed i segnali di controllo da affermare durante il ciclo corrente, mentre gli ingressi sono lo stato presente e altri ingressi necessari a determinare lo stato futuro, in questo caso i bit del campo opcode. Si noti che nelle macchine a stati finiti utilizzate in questo capitolo le uscite dipendono soltanto dallo stato presente e non dagli ingressi, come meglio spiegato dall'approfondimento che segue.

Approfondimento: Il tipo di macchina a stati finiti riportato in figura 5.43 è detto macchina di Moore, dal nome di Edward Moore, e la sua caratteristica saliente è che le uscite dipendono solamente dallo stato presente. In una macchina di Moore il rettangolo denominato logica combinatoria di controllo può essere diviso in due parti: la prima contiene le uscite di controllo e riceve in ingresso solo lo stato, mentre la seconda ha una sola uscita, quella di stato futuro.

Un tipo alternativo di macchina a stati finiti è la macchina di Mealy, dal nome di George Mealy: essa permette di utilizzare sia lo stato presente che il valore degli ingressi per determinare le uscite. Le macchine di Moore presentano dei vantaggi teorici di implementazione in termini di velocità e dimensioni dell'unità di controllo: la velocità si giustifica notando che le uscite di controllo, che devono essere disponibili il più presto possibile nel ciclo di clock, non dipendono dagli ingressi ma solo dallo stato presente; il vantaggio nelle dimensioni sarà evidente nell'Appendice C, quando si procederà nell'implementazione fino ad ottenere le porte logiche. Un possibile svantaggio delle macchine di Moore è che possono richiedere un numero maggiore di stati: per esempio, in situazioni in cui vi sia la differenza in un solo stato tra due sequenze di stati, in una macchina di Mealy si possono fondere tali stati rendendo le uscite dipendenti dagli ingressi.

5.5 La microprogrammazione: semplificare il progetto delle unità di controllo

Per l'unità di controllo del semplice sottoinsieme MIPS esaminato una rappresentazione grafica della macchina a stati finiti come quella di figura 5.42 è sicuramente adeguata: il diagramma si può disegnare in una sola pagina e la traduzione in equazioni (si veda l'Appendice C) può procedere senza troppi errori. Si consideri invece un'implementazione dell'intero set di istruzioni MIPS, che contiene oltre 100 istruzioni (si veda l'Appendice A): in un'implementazione tipica le istruzioni richiedono da 1 ad oltre 20 cicli di clock, per cui le funzioni dell'unità di controllo saranno necessariamente molto più complesse. Si può anche considerare un set di istruzioni con un maggior numero di istruzioni, appartenenti a classi molti diverse: l'unità di controllo potrebbe richiedere migliaia di stati con centinaia di sequenze distinte; per esempio, il set di istruzioni Intel 80x86 ha molte più combinazioni di modi di indirizzamento, ed un insieme molto più vasto di codici operativi.

In tali casi la specifica grafica dell'unità di controllo diviene difficilmente gestibile, essendo la macchina a stati finiti costituita da centinaia o migliaia di stati, e da un numero ancora maggiore di archi! La rappresentazione grafica, sebbene utile per piccole macchine a stati, non sarà più limitata ad una pagina, e al crescere delle sue dimensioni diverrà sempre meno comprensibile. I programmatori conoscono bene questo fenomeno: via via che i programmi divengono più grandi, sono necessarie tecniche di strutturazione aggiuntive (come procedure e moduli) per mantenerli comprensibili. Ovviamente, specificare complesse funzioni di controllo direttamente tramite equazioni, senza rischiare di commettere errori, diviene praticamente impossibile.

È forse possibile utilizzare alcune idee sorte nel campo della programmazione per aiutare la creazione di un metodo che permetta di specificare l'unità di controllo in modo maggiormente comprensibile e più facilmente progettabile? Si ipotizzi

di trattare l'insieme dei segnali di controllo che devono essere affermati in uno stato come un'istruzione che dovrà essere eseguita dall'unità di elaborazione; per evitare confusione tra le istruzioni del set di istruzioni MIPS e le istruzioni di controllo di basso livello, queste ultime saranno chiamate *microistruzioni*. Ciascuna microistruzione definisce l'insieme dei segnali di controllo dell'unità di elaborazione che devono essere affermati in un determinato stato: eseguire una microistruzione ha l'effetto di affermare i segnali di controllo da essa specificati.

Oltre a definire quali segnali di controllo affermare, si dovrà anche specificarne la sequenza temporale, ossia quale microistruzione dovrà seguire quella corrente. Facendo riferimento alla figura 5.42, il prossimo stato viene determinato in due modi possibili: talvolta vi è un solo stato futuro, che segue incondizionatamente quello corrente (per esempio, lo stato 1 segue sempre lo stato 0, e l'unico modo per raggiungere lo stato 1 è attraverso lo stato 0); in altri casi la scelta del prossimo stato dipende dagli ingressi (ciò si verifica nello stato 1, che ha quattro diversi stati futuri).

Nella programmazione si verifica una situazione analoga: talvolta un gruppo di istruzioni deve essere eseguito sequenzialmente, talvolta serve un salto; il comportamento normale è quello sequenziale, mentre i salti vanno indicati esplicitamente. Nel descrivere l'unità di controllo come un programma, si assumerà quindi che le microistruzioni scritte in sequenza vengano anche eseguite in sequenza, mentre i salti debbano essere indicati in modo esplicito. Il meccanismo di sequenzializzazione può ancora essere implementato da una struttura simile a quella di figura 5.43, ma è spesso più efficiente ricorrere ad un contatore; si vedrà un'implementazione di questo tipo al termine del paragrafo.

Progettare l'unità di controllo come un programma che implementa le istruzioni macchina facendo uso di microistruzioni più semplici prende il nome di *microprogrammazione*: l'idea fondamentale è quella di rappresentare simbolicamente i valori affermati sulle linee di controllo, in modo tale che il microprogramma sia una rappresentazione delle microistruzioni, così come il linguaggio assembler è una rappresentazione delle istruzioni macchina. Nella definizione della sintassi per il linguaggio assembler si rappresentano solitamente le istruzioni come una serie di campi (opcode, registri, campo offset o immediato); analogamente, si rappresenterà sintatticamente una microistruzione come una sequenza di campi correlati tra loro.

Definizione di un formato di microistruzioni

Il microprogramma è una rappresentazione simbolica dell'unità di controllo che verrà tradotta in porte logiche da un apposito programma: in tal modo, si potranno scegliere il numero di campi di una microistruzione ed i segnali di controllo influenzati da ciascun campo. Il formato delle microistruzioni dovrà essere scelto in modo da semplificarne la rappresentazione, rendendo più agevole la scrittura e la comprensione del microprogramma: per esempio, può risultare utile disporre di un campo che controlla la ALU ed un insieme di tre campi che determinano le due sorgenti per l'operazione della ALU e la destinazione del risultato. Oltre all'obiettivo di leggibilità, è importante che il formato del microprogramma renda difficile o impossibile la scrittura di microistruzioni incoerenti: una microistruzione è detta incoerente se richiede che un segnale di controllo venga posto contemporaneamente a due valori diversi; si vedrà un esempio tra breve.

Per evitare la possibilità che un formato permetta microistruzioni incoerenti, si può fare in modo che i diversi campi siano responsabili della specifica di insiemi diversi e disgiunti di segnali di controllo. Per scegliere come compiere la ripartizione dei segnali di controllo tra i campi delle microistruzioni per l'implementazione dell'esempio finora seguito, è utile far riferimento a due figure precedenti:

- La figura 5.33, che mostra tutti i segnali di controllo ed il loro effetto sull'unità di elaborazione.
- La figura 5.34, che riporta la funzione di ciascun segnale di controllo dell'unità di elaborazione.

Nome del campo	Funzione del campo
ALU control	Specifica l'operazione compiuta dalla ALU in questo ciclo di clock; il risultato è sempre scritto in ALUOut.
SRC1	Specifica la provenienza del primo operando della ALU.
SRC2	Specifica la provenienza del secondo operando della ALU.
Register control	Specifica la lettura o la scrittura del register file e la provenienza del valore nel caso della scrittura.
Memory	Specifica la lettura o la scrittura per la memoria e la provenienza dei dati; nel caso di una lettura specifica il registro destinazione.
PCWrite control	Specifica la modalità di aggiornamento di PC.
Sequencing	Specifica la modalità di scelta della microistruzione successiva.

Figura 5.44 Ciascuna microistruzione contiene sette campi. I valori permessi per ciascun campo sono riportati in figura 5.45.

I segnali che non sono mai affermati contemporaneamente possono condividere lo stesso campo. La figura 5.44 mostra una possibile decomposizione delle microistruzioni in sette campi e descrive la funzione generale di ciascuno di essi; i primi sei campi controllano l'unità di elaborazione, mentre il campo Sequencing (il settimo) specifica la modalità di selezione della microistruzione successiva.

Le microistruzioni vengono normalmente memorizzate in una ROM o in una PLA (strutture descritte nell'Appendice B ed utilizzate per implementare l'unità di controllo nell'Appendice C), per cui si può assegnare un indirizzo a ciascuna microistruzione: gli indirizzi sono solitamente sequenziali, così come si erano scelti numeri sequenziali per numerare gli stati nella macchina a stati finiti. La scelta della microistruzione successiva può essere effettuata mediante una delle seguenti possibilità:

1. Incrementare l'indirizzo della microistruzione corrente per ottenere l'indirizzo della successiva: questo comportamento sequenziale sarà indicato dal valore `Seq` nel campo Sequencing; essendo l'esecuzione sequenziale molto frequente, in molti sistemi di microprogrammazione questo è il valore assegnato in modo implicito (in mancanza di specifica contraria).
2. Saltare alla microistruzione che inizia l'esecuzione della prossima istruzione MIPS: la prima microistruzione (corrispondente allo stato 0) è etichettata con `Fetch` (essendo responsabile del prelievo dell'istruzione, appunto detto in inglese *fetch*) e si porrà l'indicatore `Fetch` nel campo Sequencing per indicare tale azione.
3. Scegliere la microistruzione successiva sulla base degli ingressi dell'unità di controllo: quest'operazione di scelta viene denominata *smistamento* (in inglese *dispatch*, termine che ricorda le spedizioni postali). Le operazioni di smistamento vengono solitamente implementate ricorrendo a tabelle contenenti gli indirizzi delle microistruzioni destinazione: tali tabelle sono indirizzate dagli ingressi dell'unità di controllo e possono essere implementate come ROM o PLA. Spesso vi sono diverse tabelle di smistamento; in questa implementa-

zione se ne renderanno necessarie due, una per uscire dallo stato 1 e l'altra per lo stato 2. Si indicherà che la microistruzione successiva dovrà essere determinata da un'operazione di smistamento con il valore `Dispatch i` nel campo Sequencing, dove `i` è il numero della tabella di smistamento.

La figura 5.45 descrive i valori permessi per ciascun campo della microistruzione e l'effetto dei diversi valori dei campi; si ricordi che il microprogramma è una rappresentazione simbolica, per cui il formato di microistruzione illustrato è solamente uno dei tanti possibili.

Nome del campo	Valore del campo	Funzione del campo con tale valore
Label	Una stringa qualunque	Questo campo è utilizzato per specificare le etichette utilizzate per controllare la sequenza delle microistruzioni. Le etichette che terminano con 1 o 2 sono utilizzate durante gli smistamenti tramite tabelle di salto indicizzate sulla base del codice operativo; le altre etichette sono utilizzate come destinazioni dirette per i salti tra microistruzioni. Le etichette non generano direttamente segnali di controllo, ma sono utilizzate per definire il contenuto delle tabelle di smistamento e per generare i segnali di controllo relativi al campo Sequencing.
ALU control	`Add`	Indica alla ALU di sommare.
	`Subt`	Indica alla ALU di sottrarre; è utilizzato nell'implementazione del confronto per i salti condizionati.
	`Func code`	Utilizza il campo funct dell'istruzione per determinare i segnali di controllo della ALU.
SRC1	`PC`	PC è il primo ingresso della ALU.
	`A`	Il registro A è il primo ingresso della ALU.
SRC2	`B`	Il registro B è il secondo ingresso della ALU.
	`4`	Il valore 4 va sul secondo ingresso della ALU.
	`Extend`	L'uscita dell'unità di estensione del segno va sul secondo ingresso della ALU.
	`Extshft`	L'uscita dell'unità di scalamento va sul secondo ingresso della ALU.
Register control	`Read`	Legge due registri utilizzando i campi rs e rt di IR come numeri di registro, ponendo i dati letti nei registri A e B.
	`Write ALU`	Scrive nel register file utilizzando il campo rt di IR come numero di registro ed il contenuto di ALUOut come dato.
	`Write MDR`	Scrive nel register file utilizzando il campo rt di IR come numero di registro ed il contenuto di MDR come dato.
Memory	`Read PC`	Legge la memoria utilizzando PC come indirizzo; il risultato è scritto in IR (e nel MDR).
	`Read ALU`	Legge la memoria utilizzando ALUOut come indirizzo; il risultato è scritto in MDR.
	`Write ALU`	Scrive nella memoria utilizzando ALUOut come indirizzo; il dato è contenuto in B.
PCWrite control	`ALU`	Scrive l'uscita della ALU in PC
	`ALUOut-cond`	Se l'uscita Zero della ALU è attiva, scrive in PC il contenuto del registro ALUOut.
	`Jump address`	Scrive in PC l'indirizzo di salto ricavato dall'istruzione.
Sequencing	`Seq`	Seleziona sequenzialmente la microistruzione successiva.
	`Fetch`	Torna alla prima microistruzione, che inizia l'esecuzione di una nuova istruzione.
	`Dispatch i`	Esegue uno smistamento utilizzando la ROM specificata da `i` (1 o 2).

Figura 5.45 Ciascun campo della microistruzione può assumere un certo numero di valori distinti. La seconda colonna fornisce i valori permessi per ciascun campo, mentre la terza definisce l'effetto di tali valori. Ciascun valore dei campi, eccezion fatta per l'etichetta, corrisponde ad un particolare insieme di valori dei segnali di controllo dell'unità operativa; tale corrispondenza è descritta nell'Appendice C, Paragrafo C.5, dove è anche mostrato come utilizzare il campo etichetta per generare le tabelle di smistamento. Come si vedrà, l'implementazione con microcodice differirà leggermente da quella ottenuta come macchina a stati finiti, ma solamente per dettagli che non influiscono sulla semantica delle istruzioni.

Approfondimento: Il semplice formato di microistruzioni indicato potrebbe permettere combinazioni non supportate dall'unità di elaborazione; tipicamente, un microassemblatore dovrà fare dei controlli sui campi delle microistruzioni per garantire che tali incoerenze siano segnalate come errori e corrette. Un'alternativa sarebbe ristrutturare il formato delle microistruzioni per evitare il problema, ma ciò potrebbe peggiorare la leggibilità: la maggior parte dei sistemi di microprogrammazione scelgono la leggibilità e richiedono all'assemblatore del microcodice di controllare le incoerenze.

La creazione del microprogramma

È ora possibile creare il microprogramma per l'unità di controllo; le istruzioni verranno etichettate con nomi simbolici, che potranno essere utilizzati per specificare il contenuto delle tabelle di smistamento (si veda nel Paragrafo C.5 dell'Appendice C la definizione di tali tabelle e la loro costruzione). Nello scrivere il microprogramma ci possono essere delle situazioni in cui è necessario lasciare vuoto uno dei campi: quando un campo che controlla un'unità funzionale o che causa la scrittura di un elemento di stato (come i campi Memory e Register control) è vuoto, non si dovrà affermare nessun segnale di controllo. Inoltre, quando un campo specifica *esclusivamente* il segnale di controllo di un multiplexer che a sua volta determina l'ingresso di un'unità funzionale, come il campo SRC1, lasciarlo vuoto significa che non interessa il valore di ingresso dell'unità funzionale (che coincide con l'uscita del multiplexer).

Il modo più semplice di comprendere il microprogramma è di suddividerlo in segmenti che gestiscono le diverse fasi dell'esecuzione delle istruzioni, esattamente come è stato fatto per il progetto della macchina a stati finiti.

La prima fase dell'esecuzione di ciascuna istruzione è il reperimento dell'istruzione, la sua decodifica ed il calcolo del valore di PC successivo in ordine sequenziale e di quello di destinazione nel caso di un eventuale salto. Queste azioni corrispondono direttamente ai primi due passi dell'esecuzione descritta pagine addietro. Le due microistruzioni necessarie per i due passi sono mostrate nel seguito:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
`Fetch`	`Add`	`PC`	`4`		`Read PC`	`ALU`	`Seq`
	`Add`	`PC`	`Extshft`	`Read`			`Dispatch 1`

Per comprendere l'effetto di ciascuna microistruzione è più semplice esaminare i campi a gruppi. Nella prima microistruzione i campi affermati ed il loro effetto sono i seguenti:

Campi	Effetto
ALU control, SRC1, SRC2	Calcola PC+4 (il valore è anche scritto in ALUOut, ma non verrà mai letto da tale registro).
Memory	Legge l'istruzione e la pone in IR.
PCWrite control	Scrive l'uscita della ALU in PC.
Sequencing	Va alla microistruzione successiva.

Il campo Label, che contiene l'etichetta `Fetch`, sarà utilizzato nel campo Sequencing ogni volta che il microprogramma debba passare all'esecuzione dell'istruzione successiva.

Nella seconda microistruzione si controllano le seguenti operazioni:

Campi	Effetto
ALU control, SRC1, SRC2	Memorizza PC + estensione con segno di (IR[15-0]) << 2 in ALUOut.
Register control	Usa i campi rt e rs per leggere i registri e porre i dati in A e B.
Sequencing	Usa la tabella di smistamento 1 per scegliere l'indirizzo della microistruzione successiva.

Si può pensare all'operatore di smistamento come ad un'istruzione di tipo *case* o *switch*, dove in codice operativo e la tabella di smistamento 1 sono utilizzati per scegliere una tra quattro sequenze di microistruzioni poste ad una delle seguenti etichette (si noti che terminano tutte con "1"):

- `Mem1` per le istruzioni di accesso alla memoria
- `Rformat1` per le istruzioni di tipo-R
- `BEQ1` per l'istruzione branch equal
- `JUMP1` per l'istruzione jump.

Il microprogramma per le istruzioni di accesso alla memoria è composto di quattro microistruzioni, come mostrato nel seguito. La prima calcola l'indirizzo di memoria; per completare una load è necessaria una sequenza di due microistruzioni, mentre una sola è sufficiente per una store:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch

I campi della prima microistruzione della sequenza hanno i seguenti effetti:

Campi	Effetto
ALU control, SRC1, SRC2	Calcola l'indirizzo di memoria: registro (rs) + sign-extend(IR[15-0]) e memorizza il risultato in ALUOut.
Sequencing	Usa la seconda tabella di smistamento per saltare alla microistruzione con etichetta `LW2` o `SW2`.

La prima microistruzione dedicata all'istruzione `lw` è etichettata `LW2`, dato che viene raggiunta tramite un'operazione di smistamento usando la seconda tabella; essa si comporta come segue:

Campi	Effetto
Memory	Legge dalla memoria utilizzando l'uscita della ALU come indirizzo e scrivendo il risultato in MDR.
Sequencing	Va alla microistruzione successiva.

La microistruzione successiva completa l'esecuzione nel modo seguente:

Campi	Effetto
Register control	Scrive il contenuto di MDR nell'elemento del register file specificato da rt.
Sequencing	Va alla microistruzione con etichetta `Fetch`.

La microistruzione per le store, etichettata `SW2`, si comporta in modo simile a quella con etichetta `LW2`:

Campi	Effetto
Memory	Scrive nella memoria utilizzando ALUOut come indirizzo e il contenuto di B come valore.
Sequencing	Va alla microistruzione con etichetta `Fetch`.

La sequenza di microprogramma per le istruzioni di tipo-R consiste di due microistruzioni: la prima effettua l'operazione con la ALU (ed è etichettata `Rformat1` per poter essere raggiunta dagli smistamenti), mentre la seconda scrive il risultato nel register file:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch

Poiché i campi delle due microistruzioni non sono in conflitto (in particolare, usano campi diversi), si potrebbe pensare alla possibilità di combinarle in una sola microistruzione; in effetti gli ottimizzatori di microcodice effettuano operazioni di questo tipo durante la compilazione del microcodice. Tuttavia in questo caso il risultato dell'operazione della ALU viene scritto nel registro ALUOut, ed il valore scritto non può essere letto fino al prossimo ciclo di clock: non si possono fondere le due operazioni in una sola microistruzione (se lo si facesse, si scriverebbe il valore sbagliato nel register file!). Per permettere la fusione delle microistruzioni si potrebbe eliminare il registro ALUOut, ma ciò porterebbe ad un allungamento del ciclo di clock per far sì che la scrittura nel register file possa avvenire nello stesso ciclo di clock dell'operazione fatta dalla ALU.

La prima microistruzione avvia l'operazione nella ALU:

Campi	Effetto
ALU control, SRC1, SRC2	La ALU elabora il contenuto dei registri A e B con l'operazione determinata tramite il campo funzione.
Sequencing	Va alla microistruzione successiva.

La seconda microistruzione causa la scrittura dell'uscita della ALU nel register file:

Campi	Effetto
Register control	Scrive il contenuto di ALUOut nell'elemento del register file specificato dal campo rd.
Sequencing	Va alla microistruzione con etichetta `Fetch`.

La microistruzione appena eseguita aveva calcolato l'indirizzo di destinazione del salto, quindi la sequenza di microprogramma per l'istruzione branch è composta da una sola microistruzione, etichettata con BEQ1:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
BEQ1	Subt	A	B			ALUOut-cond	Fetch

I campi affermati hanno il seguente significato:

Campi	Effetto
ALU control, SRC1, SRC2	Calcola la differenza tra A e B con la ALU per calcolare l'uscita Zero.
PCWrite control	Causa la scrittura di PC con il valore presente in ALUOut, a patto che l'uscita Zero della ALU sia affermata.
Sequencing	Va alla microistruzione con etichetta Fetch.

Anche la sequenza di microprogramma per l'istruzione jump consta di una sola microistruzione:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
JUMP1						Jump address	Fetch

I campi affermati sono i due seguenti:

Campi	Effetto
PCWrite control	Causa la scrittura di PC con l'indirizzo di destinazione del salto.
Sequencing	Va alla microistruzione con etichetta Fetch.

Il microprogramma è riportato per intero nella figura 5.46: esso è composto dalle 10 microistruzioni appena esaminate e corrisponde alla macchina a stati finiti di 10 stati progettata nel paragrafo precedente, essendo stato anch'esso derivato dalla stessa sequenza di esecuzione delle istruzioni in cinque passi. In processori più

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2				Read ALU			Seq
			Write MDR				Fetch
SW2				Write ALU			Fetch
Rformat1	Func code	A	B				Seq
			Write ALU			Fetch	
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Figura 5.46 Il microprogramma dell'unità di controllo. Si ricordi che le etichette sono utilizzate nella determinazione delle operazioni di smistamento: Dispatch 1 effettua un salto basandosi sull'IR verso un'etichetta che termina con 1, mentre Dispatch 2 utilizza l'IR per effettuare un salto ad un'etichetta che termina con 2.

complessi si sceglierebbe di rappresentare l'unità di controllo con il microprogramma, dal momento che si potrebbero raggiungere centinaia o migliaia di microistruzioni. Le unità di elaborazione dei processori complessi richiedono tipicamente dei registri temporanei addizionali per memorizzare i risultati intermedi durante l'implementazione di istruzioni complesse multi-ciclo. I registri A e B sono registri temporanei di questo tipo, ma un'unità di elaborazione per un set di istruzioni più complesso ne avrebbe un numero decisamente superiore e questi sarebbero fortemente interconnessi agli altri elementi: i registri temporanei sono a disposizione del microprogrammatore e rendono ancora più forte l'analogia tra il progetto dell'unità di controllo e l'attività di programmazione classica.

L'implementazione del microprogramma

La traduzione del microprogramma in hardware coinvolge due aspetti: decidere come implementare la funzione di sequenzializzazione e scegliere un metodo per memorizzare la funzione di generazione dei segnali di controllo. Il microprogramma può essere visto come una rappresentazione testuale di una macchina a stati finiti e può essere implementato esattamente con la stessa metodologia di implementazione delle macchine a stati: utilizzando una PLA per codificare sia la funzione di sequenzializzazione che la funzione di controllo (si faccia riferimento alla figura 5.43). In realtà, spesso l'implementazione delle funzioni di sequenzializzazione e di controllo procede in modo diverso, soprattutto per microprogrammi di grandi dimensioni.

La forma di implementazione alternativa consiste nel memorizzare la funzione di controllo in una memoria a sola lettura (ROM) e nell'implementare a parte la funzione di sequenzializzazione: in figura 5.47 si può notare l'utilizzo di un incrementatore per selezionare l'istruzione di controllo successiva. Con questo tipo di implementazione la memoria del microcodice determina il valore delle linee di controllo dell'unità di elaborazione, oltre alla *modalità di selezione* del prossimo stato (anziché *specificare* il prossimo stato, come avverrebbe nell'implementazione con macchina a stati finiti): la logica di selezione dell'indirizzo contiene le tabelle di smistamento, implementate come ROM o PLA, e determina la successiva microistruzione da eseguire sotto il controllo della logica di selezione dell'indirizzo. Il vantaggio di questa implementazione della funzione di sequenzializzazione è di eliminare la logica per la normale esecuzione in sequenza delle microistruzioni, demandando tale operazione ad un contatore: in casi in cui vi siano delle lunghe sequenze di microistruzioni, la sequenzializzazione separata potrebbe dunque portare ad una riduzione della logica nell'unità di controllo.

Nella figura 5.47 la funzione di controllo può essere implementata da una ROM anziché da una PLA: in tal caso il microprogramma viene assemblato, memorizzato nella memoria del microcodice ed indirizzato dal contatore di microprogramma, nello stesso modo in cui un normale programma è memorizzato nella memoria di programma e l'istruzione successiva viene determinata dal contatore di programma. Questa analogia con la programmazione è alla base della terminologia adottata (microcodice, microprogrammazione, ecc.), ed è uno dei primi metodi usati per implementare i microprogrammi (come si vedrà nel Paragrafo 5.10).

Sebbene il tipo di sequenzializzatore illustrato in figura 5.47 sia tipicamente utilizzato per implementare una specifica dell'unità di controllo data sotto forma di microprogramma (il processo di traduzione è descritto nel Paragrafo C.5 dell'Appendice C), potrebbe anche essere utilizzato per implementare una specifica come

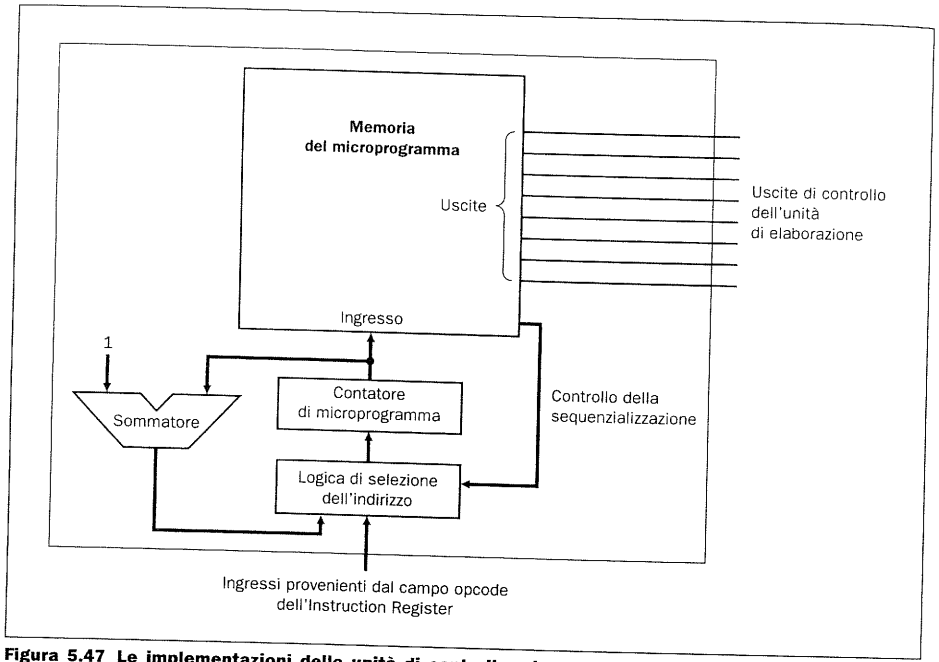


Figura 5.47 Le implementazioni delle unità di controllo microprogrammate utilizzano tipicamente un incrementatore esplicito per determinare lo stato seguente in modo sequenziale e memorizzano il microcodice in una ROM. Le microistruzioni, che vengono utilizzate per pilotare i segnali di controllo dell'unità di elaborazione, sono assemblate direttamente a partire dal microprogramma; il contatore di microprogramma, che sostituisce il registro di stato delle unità di controllo implementate con macchina a stati finiti, determina la modalità di selezione della microistruzione successiva. La logica di selezione dell'indirizzo contiene le tabelle di smistamento e la logica per selezionare tra i possibili stati futuri: la selezione della microistruzione successiva dipende dalle uscite di controllo della sequenzializzazione provenienti dalla logica di controllo. La combinazione del contatore di microprogramma, dell'incrementatore, delle tabelle di smistamento e della logica di selezione dell'indirizzo realizza un sequenzializzatore per la scelta della microistruzione successiva. La memoria per il microcodice può consistere di una memoria a sola lettura (ROM), o essere implementata tramite una PLA: le PLA sono solitamente più efficienti nelle implementazioni VLSI, mentre le ROM sono modificabili con maggior facilità (per una discussione più approfondita dei vantaggi relativi delle due alternative si rimanda al Paragrafo 5.9 e all'Appendice C).

macchina a stati finiti, come è descritto in dettaglio nel Paragrafo C.4. L'Appendice C mostra come passare dalle rappresentazioni simboliche delle macchine a stati finiti o dei microprogrammi viste in questo capitolo ai bit di memoria o alle righe delle PLA: il lettore interessato all'implementazione dettagliata o al processo di traduzione è invitato a consultare tale appendice.

Le scelte per la rappresentazione dell'unità di controllo (diagramma a stati finiti o microprogramma) e per l'implementazione della stessa (PLA o ROM, stato codificato o sequenzializzatore esplicito) sono decisioni indipendenti, influenzate sia dalla struttura della funzione di controllo sia dalla tecnologia utilizzata per implementare l'unità di controllo. Tali aspetti saranno brevemente accennati nel Paragrafo 5.9, ma prima è conveniente esaminare uno degli aspetti più complessi nell'ambito delle unità di controllo: le eccezioni.

5.6 Le eccezioni

La definizione dell'unità di controllo è una delle fasi più impegnative nel progetto dei processori. L'unità di controllo è la parte più difficile da progettare correttamente ed anche la parte più difficile da rendere veloce. Uno degli aspetti più complessi è l'implementazione delle *eccezioni* e delle *interruzioni* (o *interrupt*), ossia eventi che, come i salti, cambiano il normale flusso di esecuzione delle istruzioni. Un'eccezione è un evento inatteso proveniente dall'interno del processore: un esempio di eccezione è l'overflow aritmetico; un'interruzione è anch'essa un evento che causa una variazione inattesa nel flusso di controllo, ma proviene dall'esterno del processore: le interruzioni sono utilizzate dai dispositivi di I/O per comunicare con il processore, come si vedrà nel Capitolo 8.

Molte architetture e molti autori non distinguono tra interruzioni ed eccezioni, utilizzando spesso il più vecchio termine *interrupt* per indicare i due tipi di eventi. In questo testo si seguirà la convenzione MIPS, utilizzando il termine *eccezione* in riferimento a *tutti* i cambiamenti inattesi del flusso di controllo senza distinguere tra cause interne ed esterne; si utilizzerà il termine *interrupt* solo quando l'evento è causato esternamente. L'architettura Intel 80x86 usa il termine *interrupt* per tutti questi eventi, mentre l'architettura PowerPC utilizza *eccezione* per indicare il verificarsi di un evento inconsueto ed *interrupt* per indicare una variazione nel flusso di controllo.

Le interruzioni furono create per gestire eventi inattesi come overflow aritmetici e richieste di servizio da parte di dispositivi di I/O; lo stesso meccanismo fu esteso a trattare le eccezioni generate internamente. La tabella seguente riporta alcuni esempi e indica se la situazione è causata internamente al processore o generata esternamente:

Tipo di evento	Provenienza	Terminologia MIPS
Richiesta di un dispositivo di I/O	Esterna	Interruzione
Chiamata al sistema operativo da parte di un programma utente	Interna	Eccezione
Overflow aritmetico	Interna	Eccezione
Uso di un'istruzione indefinita	Interna	Eccezione
Malfunzionamento dell'hardware	Entrambe	Eccezione o Interruzione

Molti dei requisiti per il supporto delle eccezioni provengono dal contesto specifico che ne causa l'occorrenza, quindi si ritornerà su questo argomento nei Capitoli 7 ed 8, dove si discuteranno rispettivamente le gerarchie di memoria e le unità di ingresso/uscita, e si comprenderà più a fondo la necessità di funzionalità aggiuntive nel meccanismo di eccezione. In questo paragrafo si tratterà l'implementazione dell'unità di controllo per rilevare i due tipi di eccezioni che si possono verificare con la parte scelta del set di istruzioni e con l'implementazione presentata nei paragrafi precedenti.

Il rilevamento di condizioni eccezionali e la scelta di un'opportuna azione da intraprendere si trovano spesso sul cammino critico di una macchina, che determina il periodo del clock e quindi le prestazioni. Se non si pone particolare attenzione alle eccezioni nel progetto dell'unità di controllo, i tentativi di aggiungere la gestione delle eccezioni ad un'implementazione complessa potrebbero ridurre notevol-

mente le prestazioni, oltre a rendere più difficile la realizzazione di un progetto corretto.

La gestione delle eccezioni

I due tipi di eccezioni generabili dall'implementazione dell'esempio sono l'esecuzione di un'istruzione non valida e l'overflow aritmetico; l'operazione fondamentale che il processore deve compiere al verificarsi di un'eccezione consiste nel salvare l'indirizzo dell'istruzione responsabile dell'eccezione nell'*Exception Program Counter* (EPC) e trasferire il controllo al sistema operativo, ad un qualche indirizzo specifico.

Il sistema operativo potrà quindi prendere i provvedimenti più adeguati, che potrebbero consistere nel fornire qualche servizio al programma utente, eseguire qualche azione predeterminata in risposta all'overflow o terminare l'esecuzione del programma segnalando un errore. Dopo aver compiuto le azioni necessarie in risposta all'eccezione, il sistema operativo può terminare il programma o riprenderne l'esecuzione, utilizzando EPC per determinare il punto da cui ripartire. Nel Capitolo 7 si esaminerà in maggior dettaglio l'operazione di ripresa dell'esecuzione.

Per poter gestire correttamente l'eccezione, il sistema operativo deve conoscerne il motivo, oltre all'istruzione che l'ha causata. Vi sono due metodi per rendere disponibile il motivo di un'eccezione: il metodo utilizzato nell'architettura MIPS è di includere un registro di stato (detto registro *Causa*) contenente un campo che indica il motivo dell'eccezione.

Un metodo alternativo prevede di adottare le *interruzioni vettorizzate*: nel corso di un'interruzione vettorizzata, l'indirizzo a cui trasferire il controllo è determinato dalla causa dell'eccezione. Ad esempio, per gestire i due tipi di eccezioni sopra elencati, si potrebbero definire i seguenti indirizzi:

Tipo di eccezione	Indirizzo del vettore di eccezione (esa)
Istruzione indefinita	$C0\ 00\ 00\ 00_{esa}$
Overflow aritmetico	$C0\ 00\ 00\ 20_{esa}$

Il sistema operativo ricava il motivo dell'eccezione dall'indirizzo a cui viene attivato; gli indirizzi sono separati da 32 byte, equivalenti ad 8 istruzioni, ed in questa breve sequenza il sistema operativo deve registrare il motivo dell'eccezione ed eseguire qualche semplice elaborazione. Quando l'eccezione non è vettorizzata, tutte le eccezioni utilizzano lo stesso punto iniziale ed il sistema operativo deve decodificare il registro di stato per conoscere la causa.

Si possono implementare le operazioni richieste dalla gestione delle eccezioni aggiungendo alcuni registri e alcuni segnali di controllo all'implementazione già vista, ed estendendo lievemente la macchina a stati finiti. Si assuma di voler implementare il sistema di eccezioni utilizzato nell'architettura MIPS (l'implementazione delle eccezioni vettorizzate non porrebbe maggiori difficoltà). Occorrerà aggiungere due registri all'unità di elaborazione:

- *EPC*: un registro di 32 bit utilizzato per memorizzare l'indirizzo dell'istruzione coinvolta (tale registro sarebbe necessario anche nel caso di eccezioni vettorizzate).
- *Causa*: un registro utilizzato per memorizzare la causa dell'eccezione; nell'architettura MIPS ha 32 bit, sebbene alcuni di essi siano inutilizzati. Si assuma che il bit meno significativo di tale registro codifichi le due possibili fonti di eccezioni sopra citate: istruzione indefinita = 0 e overflow aritmetico = 1.

Saranno necessari due segnali di controllo aggiuntivi per pilotare la scrittura dei registri EPC e Causa, detti *EPCWrite* e *CausaWrite*; inoltre, servirà un segnale di controllo di 1 bit per fornire il valore corretto al bit meno significativo del registro Causa, chiamato *CausaInt*. Infine, occorrerà poter scrivere nel registro PC l'indirizzo iniziale della procedura di gestione delle eccezioni, per esempio $C0000000_{esa}$. Attualmente PC è alimentato dall'uscita di un multiplexer a tre vie, controllato dal segnale PCSource (figura 5.33): lo si può trasformare in un multiplexer a quattro vie, con l'ingresso aggiuntivo collegato al valore costante $C0000000_{esa}$, ed un valore di PCSource pari ad 11_{due} selezionerà tale valore.

Siccome PC viene incrementato nel primo ciclo di ciascuna istruzione, non si può semplicemente scrivere il valore di PC in EPC: tale valore è pari all'indirizzo voluto più quattro. Si dovrà invece utilizzare la ALU per sottrarre quattro da PC e scrivere il risultato in EPC: ciò non richiede nessun segnale di controllo aggiuntivo né alcuna connessione, poiché la ALU può essere programmata per sottrarre e la costante 4 è già uno dei suoi possibili ingressi. La porta di ingresso di EPC è dunque connessa all'uscita della ALU; la figura 5.48 riporta l'unità di elaborazione multi-ciclo completa delle aggiunte necessarie alla gestione delle eccezioni.

Con l'unità di elaborazione della figura 5.48 si potranno gestire i diversi tipi di eccezioni con uno stato ciascuno: in ciascun caso, lo stato fornisce il valore al registro Causa, ricalcola il valore originario di PC salvandolo in EPC e scrive in PC l'indirizzo della procedura di gestione delle eccezioni. Per aggiungere le due eccezioni considerate saranno quindi sufficienti i due stati riportati in figura 5.49.

Per integrare questa macchina a stati finiti in quella dell'unità di controllo occorre determinare la modalità di rilevamento delle eccezioni ed aggiungere degli archi che trasferiscano il controllo dalla sequenza principale di esecuzione alla porzione di macchina a stati finiti per la gestione delle eccezioni.

Il rilevamento delle eccezioni nell'unità di controllo

È ora necessario individuare un metodo per rilevare il verificarsi delle eccezioni e per trasferire il controllo allo stato corretto tra i due riportati in figura 5.49; le due eccezioni vengono rilevate in modo diverso:

- *Istruzione indefinita*: viene rilevata quando, a causa del valore di op, non esiste uno stato futuro definito in uscita dallo stato 1. L'eccezione viene gestita definendo che lo stato futuro per tutti i valori di op eccettuato `lw`, `sw`, 0 (tipo-R), `j` e `beq` sia lo stato 10. Tale condizione è simbolicamente indicata con l'etichetta *altro* per indicare che il campo opcode non corrisponde a nessuno di quelli che compaiono nelle altre etichette degli archi uscenti dallo stato 1; il diagramma a stati finiti è riportato in figura 5.50.
- *Overflow aritmetico*: nel Capitolo 4 si era vista la logica interna alla ALU per il rilevamento dell'overflow e si era usata un'uscita della ALU detta *Overflow* per indicare tale condizione. Questo segnale viene utilizzato nella macchina a stati finiti modificata per specificare un ulteriore stato futuro a partire dallo stato 7, come mostrato in figura 5.50.

La figura 5.50 rappresenta la specifica completa dell'unità di controllo del sottoinsieme MIPS con i due tipi di eccezioni; si ricordi che la sfida progettuale nelle unità di controllo per i processori reali sta nel riuscire a gestire la varietà di interazioni tra le istruzioni e gli altri eventi che possono causare eccezioni in modo da mantenere la logica di controllo piccola e veloce. Le complesse interazioni che pos-

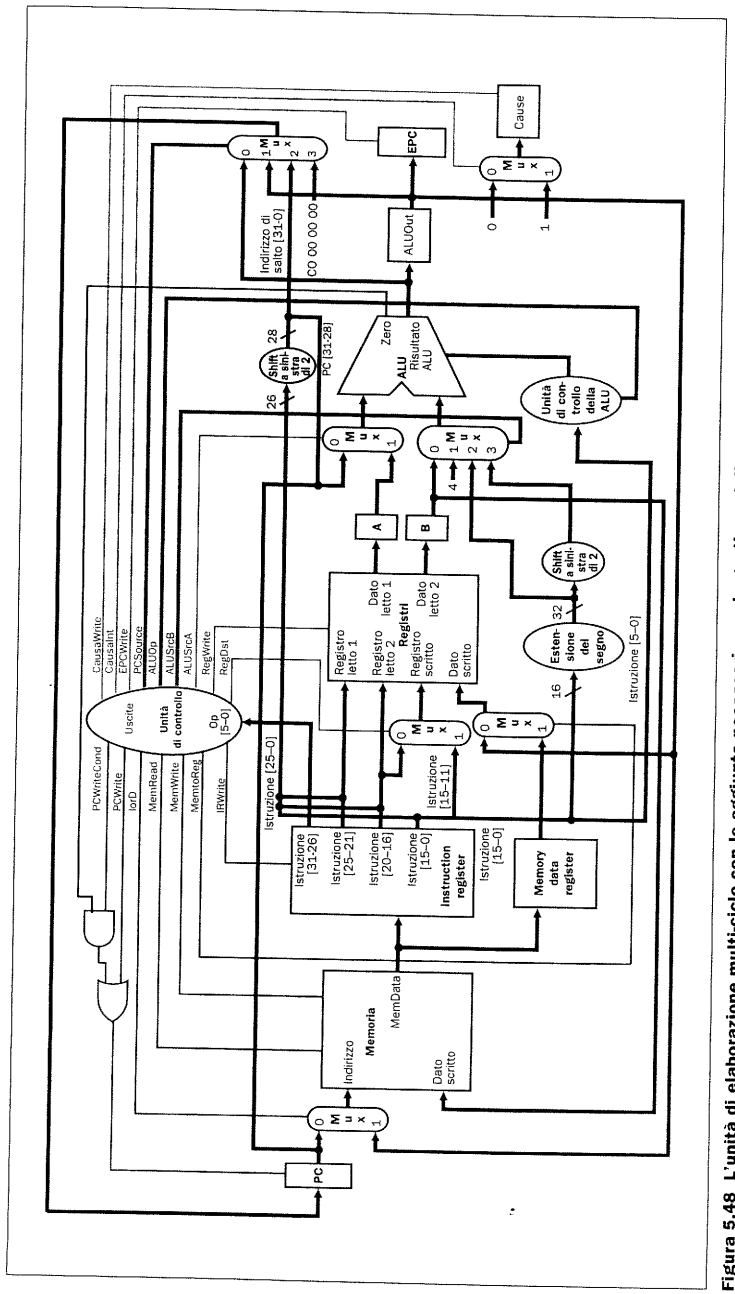


Figura 5.48 L'unità di elaborazione multi-ciclo con le aggiunte necessarie per la gestione delle eccezioni. Le aggiunte comprendono in particolare i registri Causa ed EPC, un multiplexer per determinare il valore scritto nel registro Causa, un'espansione del multiplexer che determina il valore da scrivere in PC ed i segnali di controllo per tali registri e multiplexer.

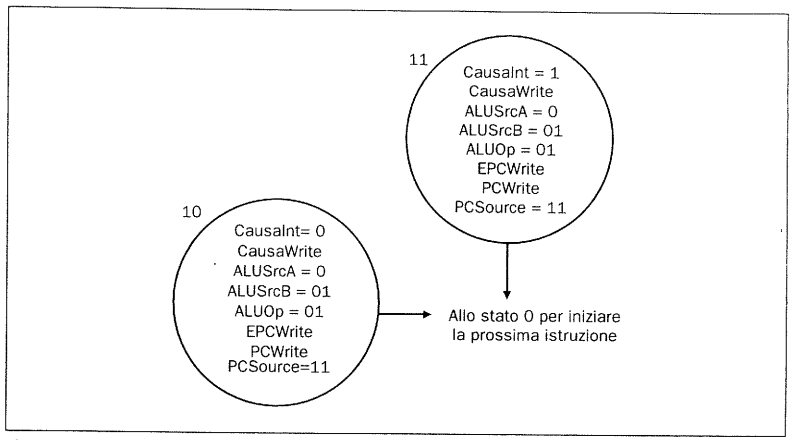


Figura 5.49 Questa coppia di stati gestisce le azioni necessarie per le due eccezioni considerate. Ciascuno stato fornisce il controllo di tre azioni: scrittura di un valore nel registro Causa, memorizzazione dell'indirizzo dell'istruzione responsabile dell'eccezione in EPC e caricamento di PC con l'indirizzo di gestione delle eccezioni. Gli stati 10 e 11 rappresentano l'inizio della gestione delle eccezioni: quando si verifica un'eccezione si trasferisce il controllo ad uno di questi stati. Dopo il completamento dello stato 10 o 11 si restituisce il controllo allo stato 0 iniziando il prelievo di una nuova istruzione.

sono verificarsi sono il fenomeno che rende l'unità di controllo la parte più difficile del progetto hardware.

Approfondimento: Ad un esame accurato della macchina a stati finiti di figura 5.50 si possono trovare alcuni problemi causati dalla modalità scelta per la gestione delle eccezioni. Per esempio, nel caso di un overflow aritmetico l'istruzione che ha causato l'overflow completa la scrittura del risultato poiché la diramazione per il caso di overflow è nello stesso stato in cui avviene la scrittura. È tuttavia possibile che la definizione dell'architettura preveda che l'istruzione non deve avere effetto se causa un'eccezione, come nel caso dell'architettura MIPS. Nel Capitolo 7 si vedrà che alcune classi di eccezioni richiedono di evitare che l'istruzione modifichi lo stato del processore: questo aspetto della gestione delle eccezioni diviene complesso e potenzialmente limita le prestazioni.

5.7 Un caso reale: l'implementazione del Pentium Pro

Le tecniche per la costruzione dell'unità di elaborazione e dell'unità di controllo descritte in questo capitolo costituiscono il cuore di ogni calcolatore; tuttavia, tutti gli elaboratori moderni vanno oltre le tecniche di questo capitolo ed utilizzano la tecnica delle *pipeline*. Tale tecnica, trattata nel prossimo capitolo, migliora le prestazioni rendendo contemporanea l'esecuzione di diverse istruzioni e raggiunge un throughput molto vicino ad un istruzione per ciclo di clock (come un'implementazione a singolo ciclo), con il periodo di clock determinato dal ritardo delle singole unità funzionali (come in un'implementazione multi-ciclo) anziché dall'intero percorso di esecuzione delle istruzioni. L'ultimo processore della famiglia Intel 80x86

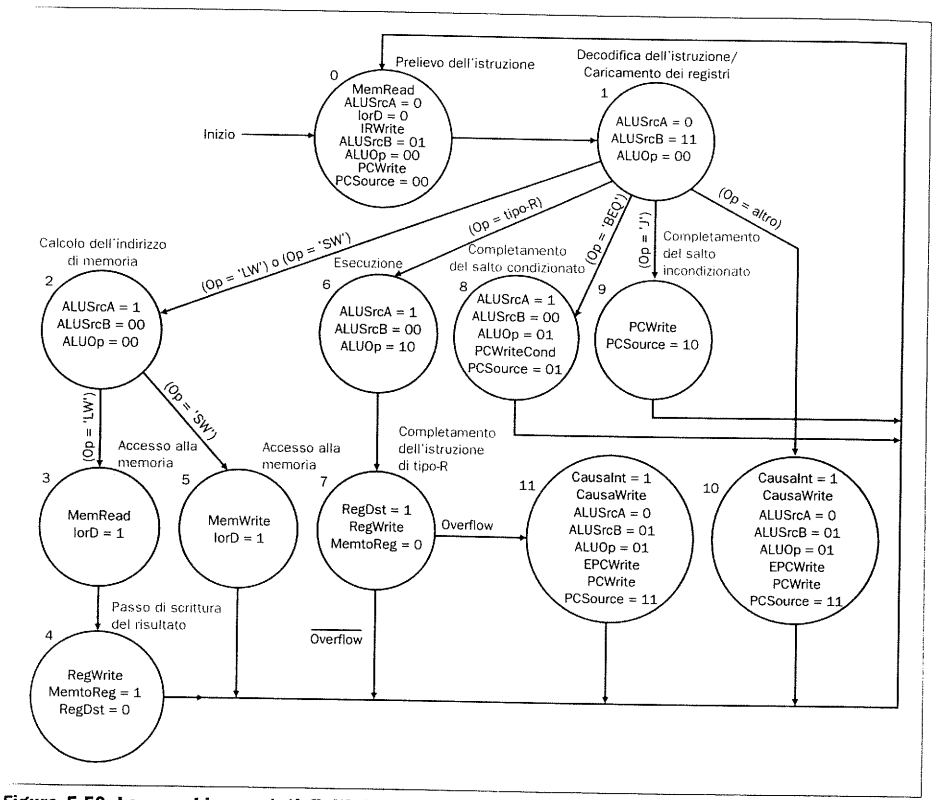


Figura 5.50 La macchina a stati finiti dell'unità di controllo con le aggiunte per il rilevamento e la gestione delle eccezioni. Gli stati 10 ed 11 provengono dalla figura 5.49; l'arco di uscita dello stato 1 etichettato con (*Op=altro*) indica lo stato futuro quando l'ingresso non corrisponde a nessun opcode tra `lw`, `sw`, 0 (tipo-R), `j` e `beq`; l'arco uscente dallo stato 7 etichettato *Overflow* indica l'azione quando la ALU segnala un overflow.

non dotato di pipeline è stato l'80386, introdotto nel 1985; il primo processore della serie MIPS, l'R2000, anch'esso introdotto nel 1985, aveva la pipeline.

I più recenti processori Intel 80x86 (l'80486, il Pentium ed il Pentium Pro) adottano tecniche di realizzazione delle pipeline sempre più sofisticate; il loro progetto si trova però sempre di fronte alla sfida di implementare un'unità di controllo per il complesso set di istruzioni 80x86 descritto nel Capitolo 3. Le unità funzionali fondamentali e le unità di elaborazione dei processori moderni, sebbene decisamente più complesse di quelle descritte in questo capitolo, condividono la stessa funzionalità di base ed hanno segnali di controllo analoghi: il progetto dell'unità di controllo si fonda quindi sugli stessi principi visti in questo capitolo.

Le sfide nell'implementazione di architetture più complesse

A differenza di quella del MIPS, l'architettura 80x86 contiene istruzioni molto complesse che possono richiedere decine, se non centinaia, di cicli di clock per essere

eseguite: per esempio, l'istruzione di copia di stringhe (`MOVS`) prevede il calcolo e l'aggiornamento di due diversi indirizzi di memoria, oltre a leggere e scrivere un byte della stringa. La maggior quantità e complessità di modi di indirizzamento nell'architettura 80x86 complica anche l'implementazione delle istruzioni più semplici, simili a quelle del MIPS. Fortunatamente, un'unità di elaborazione multi-ciclo ha una struttura che permette di adattarsi alle differenze nella quantità di lavoro richiesta per le diverse istruzioni, tipica delle istruzioni 80x86; tale adattabilità è consentita da due caratteristiche:

1. Un'unità di elaborazione multi-ciclo permette alle istruzioni di impiegare un numero di cicli diverso: le istruzioni 80x86 più semplici, simili a quelle dell'architettura MIPS, possono essere eseguite in tre o quattro cicli, mentre le istruzioni più complesse possono richiederne qualche decina.
2. Un'unità di elaborazione multi-ciclo può utilizzare lo stesso componente più di una volta per istruzione: ciò è critico sia nell'implementazione dei modi di indirizzamento più complessi, sia nelle operazioni più laboriose, entrambi aspetti salienti dell'architettura 80x86. Se ciò non fosse possibile, l'unità di elaborazione dovrebbe avere un'ampiezza tale da poter soddisfare le richieste delle istruzioni più complesse senza la possibilità di riusare alcun componente, portando ad un risultato del tutto inaccettabile. Per esempio, un'unità di elaborazione a singolo ciclo per l'80x86, non potendo riutilizzare dei componenti, richiederebbe parecchie memorie dati ed un gran numero di ALU.

L'uso di un'unità di elaborazione multi-ciclo e di un'unità di controllo microprogrammata aprono la strada ad una metodologia di implementazione del set di istruzioni 80x86; il compito difficile rimane quello di creare un'implementazione ad alte prestazioni, il che richiede di saper gestire la diversità di requisiti delle diverse istruzioni. In poche parole, un'implementazione ad alte prestazioni deve garantire che le istruzioni semplici vengano eseguite velocemente e che l'effetto della complessità del set di istruzioni sia di penalizzare prevalentemente le istruzioni più complesse e usate meno frequentemente.

Per raggiungere tale obiettivo, tutte le implementazioni Intel dell'architettura 80x86 a partire dal 486 hanno utilizzato una combinazione di unità di controllo cablata (in inglese *hardwired*) per le istruzioni più semplici e di unità di controllo microprogrammata per le istruzioni di maggior complessità. Per le istruzioni che si possono eseguire con un solo passo dell'unità operativa (cioè quelle con complessità simile ad un'istruzione MIPS), l'unità di controllo cablata genera le informazioni di controllo ed esegue l'istruzione in un passo, richiedendo pochi cicli di clock. Le istruzioni che richiedono diversi passi dell'unità operativa ed una sequenzializzazione complessa sono gestite dall'unità di controllo microprogrammata, che impiega un numero di cicli superiore e pilota i diversi passi necessari a completare l'esecuzione dell'istruzione. Il beneficio di questo approccio è di permettere al progettista di raggiungere un basso numero di cicli per le istruzioni più semplici senza dover costruire manualmente l'enorme unità di elaborazione che sarebbe necessaria per la gestione delle istruzioni più complesse nel caso generale.

La struttura dell'implementazione del Pentium Pro

Sia il processore Pentium che il Pentium Pro sono in grado di eseguire più di un'istruzione per ciclo di clock sfruttando una tecnica avanzata basata su pipeline, detta *superscalare*, che verrà descritta nel prossimo capitolo. L'aspetto più importan-

te che interessa in questo capitolo è che l'esecuzione di più di un'istruzione per ciclo di clock implica il dover duplicare alcune risorse dell'unità di elaborazione. Il modo più semplice di vedere il problema è immaginare che il processore abbia diverse unità di elaborazione, ciascuna adatta a gestire una classe di istruzioni, per esempio load e store, operazioni della ALU, salti. In tal modo il processore può eseguire una load o store nello stesso ciclo di clock in cui esegue una branch ed un'operazione della ALU; il Pentium permette l'esecuzione di due istruzioni per ciclo di clock ed il Pentium Pro quattro.

Le unità di elaborazione del Pentium Pro in effetti eseguono delle semplici microistruzioni (o microoperazioni, secondo la terminologia Intel) simili ad istruzioni MIPS: tali microistruzioni sono operazioni del tutto autonome che hanno inizialmente un'ampiezza di 72 bit. L'unità di controllo per l'implementazione di tali microistruzioni è completamente cablata: l'unità di controllo a questo livello espande insieme fino a quattro microistruzioni di 72 bit in 120 segnali di controllo per le unità di elaborazione intere e in 285 segnali di controllo per l'unità di elaborazione in virgola mobile. Quest'ultimo passo di espansione della microistruzione in segnali di controllo è molto simile alla generazione dei segnali di controllo per l'unità di elaborazione a singolo ciclo o per la ALU.

Le microistruzioni sono generate a partire dalle istruzioni 80x86 per mezzo dell'unità di controllo cablata o di quella microprogrammata: le istruzioni 80x86 che richiedono meno di quattro microistruzioni sono decodificate direttamente da un insieme di PLA in un numero variabile di microistruzioni (tra una e quattro). Per le istruzioni che richiedono più di quattro microistruzioni, l'unità di controllo effettua un'operazione di smistamento nella memoria del microcodice ed utilizza un normale sequenzializzatore di microcodice per generare cinque o più microistruzioni. La ROM del microcodice contiene circa 8000 microistruzioni, ed alcune sequenze sono condivise da più istruzioni 80x86.

L'utilizzo di una semplice unità di controllo cablata al livello più basso, e l'adozione di unità di elaborazione semplici per l'esecuzione delle microistruzioni, permettono al Pentium Pro di lavorare a frequenze di clock notevoli, pari a quelle dei processori che implementano architetture del set di istruzioni più semplici. Inoltre, il processo di traduzione, che combina il controllo diretto delle istruzioni semplici ed il controllo microprogrammato di quelle complesse, permette al Pentium Pro di eseguire velocemente le istruzioni semplici, più frequenti, fornendo un valore basso e decisamente competitivo di CPI nelle istruzioni di tipo intero.

5.8 Errori e trabocchetti

Trabocchetto: l'implementazione di un'istruzione complessa mediante microcodice potrebbe non essere più veloce di una sequenza di istruzioni più semplici.

La maggior parte dei calcolatori che hanno un set di istruzioni ampio e complesso sono implementate, almeno in parte, ricorrendo al microcodice memorizzato in ROM. Stranamente, su tali sistemi, alcune sequenze di istruzioni semplici possono essere altrettanto veloci, o addirittura più veloci, della sequenza predeterminata del microcodice per una certa istruzione.

Come si giustifica tale fenomeno? Tempo addietro il microcodice aveva il vantaggio di essere accessibile da una memoria molto più veloce di quella che contenevа le istruzioni del programma; da quando, nel 1968, furono introdotte le cache, tale vantaggio non è più così rilevante. Tuttavia, il microcodice continua ad avere il vantaggio di utilizzare i registri temporanei durante i calcoli, il che può risultare utile in sistemi con un numero limitato di registri general-purpose. Lo svantaggio del microcodice è che gli algoritmi vengono scelti prima di annunciare il processore e non possono cambiare fino al modello successivo della stessa architettura: le istruzioni di un programma, invece, possono beneficiare di un miglioramento negli algoritmi in ogni momento del ciclo di vita della macchina. Per lo stesso motivo la sequenza di microcodice probabilmente non sarà ottimale per tutte le possibili combinazioni di operandi.

Un esempio che ricade in questa categoria è l'istruzione di copia di stringa (`MOVS`) utilizzata con il prefisso di ripetizione discusso nel Capitolo 3: tale istruzione è più lenta di un ciclo in cui si copia una parola di memoria ad ogni iterazione, come già visto nel relativo paragrafo di errori e trabocchetti.

Un altro esempio è costituito dall'istruzione `LOOP`, che decrementa un registro ed effettua un salto all'etichetta specificata se il registro decrementato è diverso da zero; essa è simile all'istruzione PowerPC "branch conditional to count register" (`bcctr`), già accennata nel Capitolo 3. Tali istruzioni sono progettate per essere impiegate nel salto al termine dei cicli che hanno un numero fisso di iterazioni (come molti dei cicli *for*): l'istruzione, oltre a compiere una quantità di lavoro superiore alla media, ha il beneficio di ridurre le potenziali perdite di efficienza dovute ai salti nei sistemi a pipeline (come si vedrà nella discussioni sui salti del prossimo capitolo).

Sfortunatamente, in tutte le implementazioni recenti dell'Intel 80x86, l'istruzione `LOOP` è sempre più lenta della sequenza di codice costituita da istruzioni più semplici (se si assume che la piccola differenza nelle dimensioni del codice sia trascurabile). Di conseguenza i compilatori ottimizzanti non generano mai l'istruzione `LOOP`; a sua volta ciò riduce la spinta a rendere la `LOOP` più veloce nelle future implementazioni, in quanto è usata così poco!

Errore: se vi è spazio nella memoria del microcodice, aggiungere nuove istruzioni non costa nulla.

Uno dei benefici di un approccio microprogrammato è che la memoria del microcodice implementata nella ROM non è molto costosa: con il crescere della densità di integrazione dei transistor aggiungere della ROM è praticamente gratis. Si può tracciare un'analogia con la costruzione di una casa, in cui si scopra, praticamente alla fine dei lavori, che vi sono terreno e materiali sufficienti per aggiungere una stanza: quella stanza però non sarebbe gratuita, perché inciderebbe sui costi della manodopera e della manutenzione per tutta la durata della casa. La tentazione di aggiungere istruzioni "gratis" può avvenire solo quando il set di istruzioni non è definitivo, come è probabile nel caso del primo modello di un calcolatore: dato che è altamente desiderabile la compatibilità a livello binario dei programmi, tutti i modelli futuri di quel calcolatore saranno obbligati ad includere le cosiddette istruzioni gratuite, anche se in seguito lo spazio dovesse incidere sui costi.

Nel progetto dell'80286 si aggiunsero molte istruzioni: la disponibilità di silicio e l'uso di un'implementazione microprogrammata fecero sembrare indolori tali aggiunte. L'aggiunta probabilmente maggiore fu un sofisticato meccanismo di protezione, in larga misura inutilizzato nelle nuove implementazioni, sebbene debba comunque essere implementato. L'aggiunta fu motivata da un apparente bisogno di quel tipo di meccanismo e dal desiderio di migliorare le architetture dei microprocessori per fornire funzionalità simili a quelle dei calcolatori più grandi.

Analogamente si aggiunsero diverse istruzioni per operare in aritmetica decimale sui byte: queste istruzioni sono oggi raramente utilizzate perché l'utilizzo dell'aritmetica binaria a 32 bit e le conversioni da e verso la rappresentazione decimale portano a risultati decisamente più veloci. Come per il meccanismo di protezione, le istruzioni decimali devono essere implementate anche nei processori nuovi, anche se utilizzate solo raramente.

5.9 Note conclusive

Come si è visto in questo capitolo, sia l'unità di elaborazione che l'unità di controllo di un processore possono essere progettate a partire dall'architettura del set di istruzioni e dalla conoscenza delle caratteristiche principali della tecnologia. Nel Paragrafo 5.2 si è visto come costruire l'unità di elaborazione per un processore MIPS basandosi sull'architettura, nel caso di un'implementazione a singolo ciclo. Ovviamente la tecnologia sottostante influenza molte delle decisioni progettuali, determinando quali componenti utilizzare nell'unità di elaborazione e se un'implementazione a singolo ciclo abbia senso. In un modo simile la prima parte del Paragrafo 5.4 ha esaminato come la decisione di suddividere il ciclo di clock in una serie di passi portasse ad una diversa unità di elaborazione di tipo multi-ciclo; in entrambi i casi molte caratteristiche del progetto erano dettate dall'organizzazione a livello superiore (macchina a singolo ciclo o multi-ciclo), oltre che dal set di istruzioni.

Analogamente, l'unità di controllo è definita in larga misura dall'architettura del set di istruzioni, dall'organizzazione e dal progetto dell'unità di elaborazione. Nell'organizzazione a singolo ciclo questi tre aspetti definiscono essenzialmente il modo in cui calcolare i segnali di controllo; nell'organizzazione multi-ciclo i requisiti di controllo sono anche definiti dall'esatta decomposizione dell'esecuzione dell'istruzione in cicli, a sua volta basata sull'architettura del set di istruzioni.

L'unità di controllo è uno degli aspetti più critici del progetto di calcolatori: uno dei motivi principali è che il suo progetto richiede la comprensione del funzionamento di tutte le componenti del processore. Per agevolare il superamento di questa sfida sono state esaminate due tecniche di specifica dell'unità di controllo: diagrammi a stati finiti e microprogrammazione; tali rappresentazioni permettono di astrarre la specifica dell'unità di controllo dai dettagli della sua implementazione. Utilizzare le astrazioni in questo modo è una delle metodologie più importanti a disposizione dei progettisti per trattare la complessità del progetto dei calcolatori.

Una volta specificata l'unità di controllo, la si può convertire in hardware: i dettagli dell'implementazione dipenderanno sia dalla struttura dell'unità di controllo, sia dalla tecnologia sottostante utilizzata. L'astrazione della specifica dell'unità di controllo ha anche il vantaggio di rendere trasparenti le decisioni sull'implementazione, che dipendono dalla tecnologia e che hanno elevata probabilità di cambiare al passare del tempo.

Compromessi negli approcci relativi all'unità di controllo

Molto è cambiato da quando Wilkes [1953] scrisse il primo articolo sulla microprogrammazione; i cambiamenti più importanti sono i seguenti:

- Le unità di controllo sono implementate come parte integrale del processore, spesso sullo stesso circuito integrato: non si possono più modificare indipendentemente dal resto del processore. Inoltre, data la disponibilità di opportuni strumenti di progettazione assistita da calcolatore, la difficoltà di implementazione di una ROM e di una PLA sono analoghe.
- Le ROM, che venivano solitamente usate per memorizzare le microistruzioni, non hanno più una velocità maggiore delle RAM, che contengono il programma in linguaggio macchina. Un'implementazione con PLA della funzione di controllo è solitamente molto più compatta della corrispondente ROM, che potrebbe avere righe duplicate o inutilizzate; se la PLA è più piccola, generalmente è anche più veloce.
- I set di istruzioni sono molto più semplici di quanto non fossero negli anni '60 e '70, portando ad una diminuzione di complessità dell'unità di controllo.
- Gli strumenti di progettazione assistita da calcolatore (CAD) sono migliorati così tanto da permettere di specificare in modo simbolico l'unità di controllo e, utilizzando calcolatori molto più veloci, di simularla approfonditamente prima di costruire l'hardware. Tale miglioramento rende plausibile la realizzazione della logica di controllo corretta senza richiedere correzioni in momenti successivi.

Questi cambiamenti hanno sfocato le distinzioni tra le diverse possibilità di implementazione; sicuramente utilizzare una rappresentazione astratta per l'unità di controllo è utile, ma il modo in cui questa viene implementata dipende dalle sue dimensioni, dalla tecnologia sottostante e dagli strumenti CAD disponibili.

Quadro d'insieme

L'unità di controllo può essere progettata utilizzando diverse possibili rappresentazioni: la scelta della modalità di controllo della sequenzializzazione e della modalità di rappresentazione della logica possono essere determinate indipendentemente, in un momento successivo, così come l'implementazione può procedere secondo diversi metodi utilizzando tecniche logiche strutturate. La figura 5.51 riporta la varietà di metodi per la specifica dell'unità di controllo e per la conversione della specifica in un'implementazione per mezzo di qualche forma di logica strutturata.

5.10 Inquadramento storico e approfondimenti

Maurice Wilkes imparò la progettazione dei calcolatori da Eckert e Mauchly durante un seminario estivo e procedette a costruire il primo calcolatore stored program completo e funzionante: l'EDSAC. Da tale esperienza egli si rese conto delle difficoltà poste dall'unità di controllo; pensò ad un controllo più centralizzato utilizzando una matrice di diodi e, dopo aver preso visione del calcolatore Whirlwind negli Stati Uniti, scrisse [Wilkes 1985]:

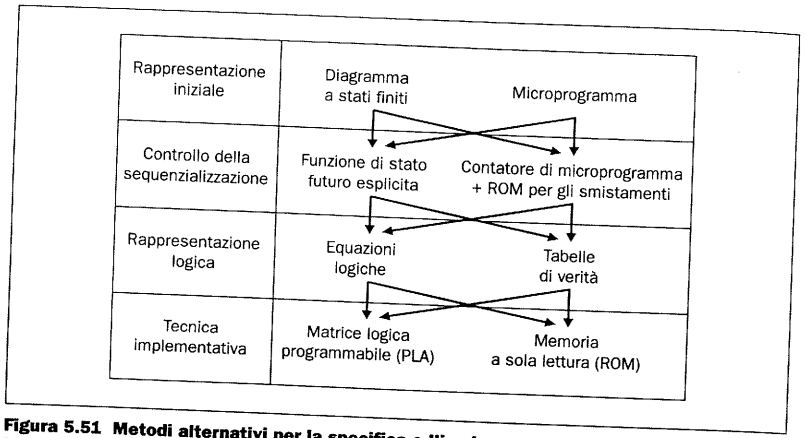


Figura 5.51 Metodi alternativi per la specifica e l'implementazione dell'unità di controllo. Le frecce indicano i possibili percorsi progettuali: tutti i cammini dalla rappresentazione iniziale alla tecnologia implementativa finale sono percorribili. Per tradizione, l'unità di controllo "cablata" indica che si utilizzano le tecniche riportate sulla sinistra, mentre con unità di controllo "microprogrammata" si fa riferimento all'adozione delle tecniche sulla destra.

Trovai che effettivamente aveva un controllo centralizzato basato su una matrice di diodi; era tuttavia solo capace di produrre una sequenza fissa di otto impulsi: una sequenza diversa per ciascuna istruzione, ma cionondimeno fissa per quanto riguarda una data istruzione. Credo che non fu prima di essere tornato a Cambridge che mi resi conto che la soluzione era di trasformare l'unità di controllo in un calcolatore in miniatura, aggiungendo una seconda matrice per determinare il flusso del controllo a livello microscopico e provvedendo al supporto delle microistruzioni condizionali.

Wilkes [1953] fu in anticipo sui tempi nell'individuare il problema; sfortunatamente, anche la sua soluzione fu in anticipo sui tempi: per costruire l'unità di controllo, la microprogrammazione si basa su una memoria veloce, non ancora disponibile negli anni '50. Quindi le idee di Wilkes rimasero principalmente una congettura accademica per una decina di anni, nonostante egli avesse costruito nel 1958 il calcolatore EDSAC 2 utilizzando un'unità di controllo microprogrammata con una ROM costituita da nuclei magnetici.

La IBM portò alla ribalta la microprogrammazione nel 1964 con la famiglia IBM 360. Prima di questo evento, l'IBM si considerava un gruppo di molte piccole aziende che vendevano macchine diverse, ciascuna con il suo prezzo ed il suo livello di prestazioni, ma anche con il suo set di istruzioni (si ricordi che si programmava molto poco in linguaggi ad alto livello, quindi i programmi scritti per un calcolatore IBM non potevano essere eseguiti su un altro). Gene Amdahl, uno dei capo architetti dell'IBM 360, disse un giorno che i manager delle diverse linee produttive furono d'accordo sulla famiglia di calcolatori 360 solamente perché erano convinti che la microprogrammazione ne avrebbe permesso la realizzazione. Per essere certo della praticabilità del progetto con microcodice, il vice presidente del ramo ingegneria dell'IBM fece persino una furtiva visita a Wilkes ed ebbe una discussione "teorica" sui vantaggi e gli svantaggi del microcodice. L'IBM riteneva l'idea così importante per i propri piani da dare una spinta al settore delle tecnologie di memorizzazione interno alla società per rendere fattibile la microprogrammazione.

Steward Tucker dell'IBM fu gravato della responsabilità di convertire al nuovo IBM 360 il software dell'IBM 7090; pensando alle possibilità del microcodice, suggerì di estendere la memoria dell'unità di controllo per includere dei simulatori, o degli interpreti, delle macchine esistenti. Tuckler [1967] coniò il termine *emulazione* per indicare una simulazione completa a livello di microprogrammazione; a volte l'emulazione sul 360 risultò addirittura più veloce dell'hardware originale.

Non appena il gigante del settore industriale (l'IBM copriva più di metà del settore informatico, in termini di ricavi) iniziò ad usare il microcodice, gli altri lo seguirono; una difficoltà nell'adozione del microcodice fu che la tecnologia di memorizzazione necessaria non era facilmente ottenibile, ma la soluzione arrivò presto con le ROM a semiconduttori, e poco dopo con le RAM. L'industria dei microprocessori seguì lo stesso percorso: le risorse limitate dei primi chip obbligavano ad un controllo cablato, ma, al crescere delle risorse, molti si persuasero ad utilizzare la microprogrammazione per via di una più semplice progettazione, una maggior facilità nei cambiamenti e la possibilità di utilizzare diverse tecniche implementative alternative.

Negli anni '60 e '70 la microprogrammazione era una delle tecniche più importanti nell'implementazione degli elaboratori. In gran parte di quel periodo, i sistemi erano implementati con componenti discreti o MSI (medium-scale integration, meno di 1000 porte logiche per chip), ed i progettisti dovevano decidere tra i due tipi di implementazione: *unità di controllo cablata* o *unità di controllo microprogrammata*. Il controllo cablato era caratterizzato da macchine a stati finiti che utilizzavano un calcolo esplicito dello stato futuro ed erano implementate principalmente con logica sparsa; in quei tempi, l'unità di controllo microprogrammata specificava le azioni di controllo mediante il microcodice, il quale veniva implementato con un sequenzializzatore (o contatore) di microprogramma ed una o più ROM. L'unità di controllo cablata ricevette questo nome perché la funzione di controllo era implementata in hardware e non era facilmente modificabile; il microprogramma memorizzato in ROM fu anche detto *firmware* (dall'inglese *firm*, stabile), perché poteva essere modificato un po' più facilmente dell'hardware, ma non così facilmente come il software.

La dipendenza da parti standard a bassa o media scala di integrazione rese radicalmente diversi questi due stili di progettazione: gli approcci microprogrammati erano interessanti perché era estremamente costoso implementare l'unità di controllo con un gran numero di porte logiche di bassa densità. Inoltre, la diffusione di set di istruzioni relativamente complessi richiedeva un'ampia unità di controllo, rendendo più efficiente un'implementazione basata su ROM. Le implementazioni cablate erano più veloci, ma troppo costose per la maggior parte dei sistemi; inoltre era molto difficile progettare correttamente l'unità di controllo e la sostituzione delle ROM era più semplice della correzione di un'unità di controllo a logica sparsa. Alla fine l'unità di controllo microprogrammata fu implementata in RAM, per permettere variazioni verso la fine del ciclo di progetto ed addirittura sul campo, dopo che la macchina era stata venduta.

Con la maggior diffusione della microprogrammazione comparvero set di istruzioni più sofisticati: anno dopo anno molte microarchitetture divennero sempre più dedicate a supportare un dato set di istruzioni, e quindi la riprogrammazione per un diverso set di istruzioni non offrì prestazioni soddisfacenti. Con il passare del tempo comparvero memorie di microcodice più ampie e divenne possibile prendere in considerazione dei sistemi della complessità del VAX, con più di 300 diversi codici operativi e oltre una decina di modi di indirizzamento della memoria. L'uso della

RAM per memorizzare il microcodice rese possibile la messa a punto del microcodice stesso e la correzione degli errori anche quando le macchine erano già sul campo. L'architettura VAX rappresenta il limite superiore per le architetture basate su implementazioni a microcodice: le tipiche implementazioni del set di istruzioni completo del VAX richiedevano da 400 a 500 Kbit di memoria di controllo.

L'architettura VAX è stata messa in pensione e sostituita dall'architettura Alpha: questa nuova architettura si basa sugli stessi principi progettuali utilizzati in altre architetture RISC, tra cui le architetture MIPS, SPARC, IBM PowerPC e HP Precision. Alla scomparsa del VAX la microprogrammazione di tipo tradizionale, in cui l'unità di controllo è implementata in un'unica grande memoria di controllo, scompare quasi del tutto dai progetti dei microprocessori: anche processori come gli Intel Pentium e Pentium Pro contengono ampie parti di controllo cablato, almeno per il loro nucleo centrale.

Ovviamente il progetto delle unità di controllo continuerà ad essere uno degli aspetti principali di tutti i calcolatori, ed il modo migliore di specificare ed implementare l'unità di controllo varierà, così come varieranno i calcolatori, da architetture RISC di tipo lineare con un'unità di controllo semplice, a processori special-purpose con potenzialmente molta logica di controllo più complessa e specializzata. Una recente mossa in questa direzione è l'annuncio da parte della Sun della costruzione di processori progettati per interpretare il linguaggio Java: la competitività di approccio rispetto a quello compilato, l'esistenza di un mercato significativo per processori più specializzati, e il ruolo che giocherà il microcodice sono domande alle quali verrà data una risposta nei prossimi anni.

Letture consigliate

Kidder, R. [1981]. *Soul of a New Machine*, Little, Brown and Co., New York.
Descrive il progetto della serie Eclipse della Data General, che sostituì i primi sistemi DG come il Nova. Kidder testimonia le strette interazioni tra gli architetti, i progettisti hardware, i programmatori di microcodice e la dirigenza del progetto.

Levy, H. M., e R. H. Eckhouse, Jr. [1989]. *Computer Programming and Architecture: The VAX*, second ed., Digital Press, Bedford, MA.
Buona descrizione dell'architettura VAX e di diverse implementazioni microprogrammate.

Patterson, D. A. [1983]. "Microprogramming," *Scientific American* 248:3 (Marzo) 36-43.
Presentazione generale dei concetti sulla microprogrammazione.

Tucker, S. G. [1967]. "Microprogram control for the System/360," *IBM Sytstems J.* 6:4, 222-41.
Descrive l'unità di controllo microprogrammata del 360, il primo sistema commerciale microprogrammato.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA..
Una biografia affascinante con molti aneddoti sui pionieri dell'industria e sui tentativi ed i successi nella costruzione dei primi elaboratori.

Wilkes, M. V. e J. BN. Stringer [1953]. "Microprogramming and the design of the control circuits in an electronic digital computer," *Proc. Cambridge Philosophical Society* 49:230-38. Ristampato anche in D. P. Siewiorek, C. G. Bell e A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 158-63, 1982 e in "The Genesis of Microprogramming," in *Annals of the History of Computing* 8:116.
Questi due articoli classici descrivono la proposta di Wilkes per l'uso del microcodice.

5.11 Parole chiave

Questo paragrafo elenca i nuovi termini fondamentali introdotti in questo capitolo, che spaziano dagli elementi dell'unità di elaborazione alle metodologie di temporizzazione, ai meccanismi di controllo, alle strutture logiche utilizzate nell'unità di controllo. I termini sono definiti nel Glossario.

eccezione o interrupt
elemento dell'unità di elaborazione
estensione del segno
firmware
implementazione a singolo ciclo
implementazione multi-ciclo o a molti cicli di clock
indirizzo di destinazione del salto
interrupt vettorizzato
macroistruzione
microcodice
microistruzione
microprogramma
salto eseguito
salto non eseguito
salto ritardato
segnale di controllo
smistamento o dispatch
superscalare
termine indifferente o *don't care*
unità di controllo cablata
unità di controllo microprogrammata

5.12 Esercizi

5.1 [5] <§5.3> Si descrivano gli effetti che un guasto singolo di tipo stuck-at-0 (in cui, cioè, indipendentemente dal valore che dovrebbe assumere, un segnale sia sempre bloccato a 0) avrebbe sui multiplexer nell'unità di elaborazione a singolo ciclo di figura 5.19. Quali istruzioni funzionerebbero ancora? Si considerino separatamente i seguenti guasti: RegDst=0, ALUSrc=0, MemtoReg=0, Zero=0.

5.2 [5] <§5.3> Si risolva un esercizio simile al 5.1, in cui i guasti considerati siano del tipo stuck-at-1 (cioè il segnale è bloccato ad 1).

5.3 [5] <§5.4> Si risolva un esercizio simile al 5.1, in cui si considerino gli effetti dei guasti stuck-at-0 sui multiplexer dell'unità di elaborazione multi-ciclo di figura 5.32. Si considerino i seguenti guasti: RegDst=0, MemtoReg=0, IorD=0 ALUSrcA=0.

5.4 [5] <§5.4> Si risolva un esercizio simile al 5.3, in cui i guasti considerati siano del tipo stuck-at-1 (cioè il segnale è bloccato ad 1).

5.5 15] <§5.3> Si desidera aggiungere l'istruzione `addi` (add immediate, somma con operando immediato) all'unità di elaborazione a singolo ciclo descritta in questo capitolo. Si aggiungano all'unità di elaborazione di figura 5.19 tutte le unità funzionali necessarie ed i relativi segnali di controllo, e si indichino le aggiunte necessarie alla tabella di figura 5.20. Si possono fotocopiare le figure o scaricarle dall'indirizzo *www.mkp.com/cod2e.htm* per facilitare l'indicazione delle aggiunte.

5.6 [15] <§5.3> Si risolva un esercizio simile al 5.5, dove però si voglia aggiungere l'istruzione `jal` (jump and link), descritta nel Capitolo 3. È conveniente partire dall'unità di elaborazione di figura 5.29.

5.7 [8] <§5.3> Si risolva un esercizio simile al 5.5, dove però si voglia aggiungere l'istruzione `bne` (branch if not equal), descritta nel Capitolo 3.

5.8 [15] <§5.3> Si risolva un esercizio simile al 5.5, dove però si voglia aggiungere una variante dell'istruzione `lw` (load word) che ottenga l'indirizzo del dato da leggere sommando due registri (si veda l'Esercizio 4.16) e che usi il formato-R.

5.9 [5] <§5.3> Si spieghi perché non è possibile modificare l'implementazione a singolo ciclo in modo da includere l'istruzione di scambio (`swap`) descritta nell'Esercizio 4.40, a meno di poter modificare il register file.

5.10 [5] <§§5.3, 5.4> Un amico propone di eliminare il segnale di controllo MemtoReg: il multiplexer che attualmente riceve in ingresso MemtoReg dovrà invece usare il segnale di controllo MemRead. La modifica proposta dall'amico funzionerà? Si considerino entrambe le unità di elaborazione.

5.11 [10] <§5.3> Si risolva un esercizio simile al 5.10, ma di natura più generale. Si determini se esistono dei segnali di controllo (diversi da MemtoReg) nell'implementazione a singolo ciclo che possano essere eliminati e sostituiti da un segnale di controllo esistente. Perché sì o perché no?

5.12 [15] <§5.3> Si consideri la seguente idea: si modifica l'architettura del set di istruzioni e si elimina la capacità di specificare uno spiazzamento per le istruzioni di accesso alla memoria. In particolare, tutte le istruzioni load e store con offset non nulli diverrebbero pseudoistruzioni e sarebbero implementate da due istruzioni; per esempio:

```
addi $at, $t1, 104  # somma l'offset ad un registro temporaneo
lw   $t0, $at       # nuovo modo di eseguire lw $t0, 104($t1)
```

Come si dovrebbe modificare l'unità di elaborazione a singolo ciclo e la relativa unità di controllo se si volesse usare tale architettura semplificata?

5.13 [10] <§5.3> {Es. 5.12} Se le modifiche descritte nell'Esercizio 5.12 venissero implementate, nascerebbero dei compromessi riguardanti le prestazioni; in particolare, il periodo del clock potrebbe ridursi e tutte le istruzioni load e store con offset non nullo richiederebbero un'ulteriore istruzione `addi` (un buon compilatore potrebbe trovare dei modi per ridurre il numero di `addi` richieste, ma si ignori questo fatto). Se vi fossero troppe istruzioni load e store con offset non nulli, probabilmente la modifica non migliorerebbe le prestazioni: assumendo i ritardi specificati nell'esempio del Paragrafo 5.3, quale sarebbe la massima percentuale di istruzioni di accesso alla memoria con offset che si può tollerare (ossia che permette alla modifica di avere un impatto positivo sulle prestazioni)?

5.14 [10] <§5.3> Nella stima delle prestazioni dell'implementazione a singolo ciclo si era ipotizzato che solo le unità funzionali principali introducessero dei ritardi (cioè si era considerato trascurabile il ritardo di multiplexer, unità di controllo, accesso a PC, unità di estensione del segno ed interconnessioni). Si assuma di modificare i ritardi utilizzati nel Paragrafo 5.3, supponendo di disporre di sommatori diversi per le somme più semplici:

- ALU: 2 ns
- sommatore per PC+4: X ns
- sommatore per il calcolo dell'indirizzo di salto: Y ns

a. Quale sarebbe il tempo di ciclo ponendo X=3 e Y=3?

b. Quale sarebbe il tempo di ciclo ponendo X=5 e Y=5?

c. Quale sarebbe il tempo di ciclo ponendo X=1 e Y=8?

5.15 [15] <§5.4> Si vuole aggiungere l'istruzione `addi` (add immediate) all'unità di elaborazione multi-ciclo descritta in questo capitolo; tale istruzione è stata descritta nel Capitolo 4. Si aggiungano le unità funzionali ed i segnali di controllo necessari all'unità di elaborazione multi-ciclo di figura 5.33 e si mostrino le modifiche necessarie alla macchina a stati finiti di figura 5.42. Potrebbe essere utile riesaminare i passi di esecuzione elencati nel Paragrafo 5.4 e considerare quali passi debbano essere eseguiti per eseguire la nuova istruzione. Si possono fotocopiare le figure o scaricarle dall'indirizzo *www.mkp.com/cod2e.htm* per facilitare l'indicazione delle aggiunte. Si cerchi di ottenere una soluzione che minimizzi il numero di cicli di clock richiesti per la nuova istruzione; si calcoli esplicitamente tale numero.

5.16 [5] <§§5.5, 5.8> {Es. 5.15} Si scriva la sequenza di microcodice per l'istruzione `addi`. Se occorresse modificare in qualche modo il formato delle microistruzioni o il contenuto dei campi, si indichi come determinare i segnali di controllo dal nuovo formato e dai nuovi campi.

5.17 [15] <§5.4> Si risolva un esercizio simile al 5.15, dove però si voglia aggiungere l'istruzione `jal` (jump and link), descritta nel Capitolo 3.

5.18 [15] <§5.4> Si risolva un esercizio simile al 5.15, dove però si voglia aggiungere l'istruzione di scambio `swap`, descritta nell'Esercizio 4.40. Non si modifichi il register file. Poiché non è ancora stato definito alcun formato per l'istruzione `swap`, lo si definisca nel modo più opportuno.

5.19 [15] <§5.4> Si risolva un esercizio simile al 5.15, dove però si voglia aggiungere una nuova istruzione, `wai` (where am I, letteralmente: dove mi trovo?), che ponga l'indirizzo dell'istruzione (il valore di PC al momento del reperimento dell'istruzione) in un registro specificato dal campo rt dell'istruzione in linguaggio macchina. Si assuma che l'unità di elaborazione non subisca modifiche e che, come al solito, il periodo del clock sia troppo breve per permettere un'operazione di ALU ed un accesso al register file nello stesso ciclo di clock, qualora il risultato di una delle due azioni influenzi l'altra.

5.20 [15] <§5.4> Si risolva un esercizio simile al 5.15, dove però si voglia aggiungere una nuova istruzione, `jm` (jump memory, salto indiretto attraverso la memoria): il formato di tale istruzione è simile a quello della load word, eccetto che il campo rt non è utilizzato perché i dati letti dalla memoria vengono posti in PC anziché nel registro destinazione.

5.21 [20] <§5.4> Si risolva un esercizio simile al 5.15, dove però si voglia aggiungere il supporto per istruzioni aritmetiche a quattro operandi, come ad esempio `add3`, che somma tra loro tre numeri anziché due:

```
add3   $t5, $t6, $t7, $t8     # $t5 = $t6 + $t7 + $t8
```

Si assuma che l'architettura del set di istruzioni venga modificata, aggiungendo un nuovo formato di istruzione, simile al formato-R, dove però i bit [0-4] siano usati per determinare il registro aggiuntivo (si continuano ad usare rs, rt e rd) ed ovviamente si usi un opcode diverso. La soluzione non deve richiedere porte di lettura aggiuntive nel register file, né si deve aggiungere una nuova ALU.

5.22 [10] <§5.4> Si mostri come si possa implementare l'istruzione jump register (descritta nei Paragrafi 3.5 e A.10) modificando semplicemente la macchina a stati finiti di figura 5.42 (può aiutare ricordare che `$0` = `$zero` = 0).

5.23 [15] <§5.4> Si consideri una modifica all'implementazione multi-ciclo che alteri il register file in modo che questo abbia una sola porta di lettura. Si descrivano (con un diagramma) le modifiche aggiuntive che saranno necessarie all'unità di elaborazione per supportare la modifica. Si modifichi la macchina a stati finiti indicando come funzioneranno le istruzioni, data la nuova unità di elaborazione.

5.24 [15] <§§5.1-5.4> In questo esercizio si usino i dati su gcc di figura 4.54; si ipotizzi che vi siano tre macchine:

- M1: L'unità di elaborazione multi-ciclo del Capitolo 5 con un clock di 500 MHz.
- M2: Una macchina simile all'unità di elaborazione multi-ciclo del Capitolo 5, dove però gli aggiornamenti dei registri vengono fatti nello stesso ciclo di clock di una lettura in memoria o di un'operazione con la ALU. Quindi nella figura 5.42 gli stati 6 e 7 e gli stati 3 e 4 sarebbero fusi insieme. La macchina ha un clock di 400 MHz, dal momento che l'aggiornamento dei registri allunga il cammino critico.
- M3: Una macchina simile ad M2, dove i calcoli degli indirizzi vengono effettuati nello stesso ciclo di clock dell'accesso alla memoria. Quindi gli stati 2, 3 e 4 si potranno fondere, così come 2 e 5 e come 6 e 7. La macchina ha un clock di 250 MHz per via del lungo periodo richiesto dalla combinazione del calcolo dell'indirizzo e dell'accesso alla memoria.

Si trovi quale di queste macchine è più veloce; vi sono combinazioni di istruzioni che renderebbero più veloce un'altra macchina? Se sì, quali?

5.25 [20] <§5.4> Alcuni amici che lavorano alla S^3 (Società Sistemi Sopraffini) hanno scoperto che il cammino critico che determina la durata del periodo di clock dell'unità di elaborazione multi-ciclo è l'accesso in memoria per le load e le store (ma *non* per la lettura delle istruzioni). Tale fatto ha fatto sì che la loro nuova implementazione del MIPS 3000 lavorasse ad una frequenza di clock di 500 MHz anziché ai previsti 750 MHz. Tuttavia, alla S^3 Chiara ha una soluzione: se tutti i cicli che accedono alla memoria venissero spezzati in due cicli di clock, la macchina potrebbe funzionare alla frequenza di clock prevista. Utilizzando le combinazioni di istruzioni del gcc riportate nel Capitolo 4 (figura 4.54), si determini di quanto sarebbe più veloce la macchina con un ciclo di accesso alla memoria di due cicli di clock rispetto alla macchina a 500 MHz con l'accesso alla memoria di un solo ciclo. Si assuma che tutti i salti condizionati ed incondizionati richiedano lo stesso numero di cicli e che le istruzioni aritmetiche immediate siano implementate come istruzioni di tipo-R.

5.26 [20] <§5.4> Si supponga l'esistenza di un'istruzione MIPS, detta `bcp` (block copy), che copia un blocco di parole da un indirizzo di memoria ad un altro. Si assuma che tale istruzione richieda che l'indirizzo di partenza del blocco sorgente sia nel registro `$t1`, l'indirizzo di destinazione in `$t2`, ed il numero di parole da copiare (sempre ≥ 0) in `$t3`. Si assuma inoltre che i valori di tali registri, come pure il valore di `$t4`, possano essere distrutti durante l'esecuzione dell'istruzione (in tal modo i registri potranno essere usati per memorizzare dei valori temporanei durante l'esecuzione).

Si scriva il programma in linguaggio assembler MIPS che implementi la copia del blocco: quante istruzioni verrebbero eseguite nella copia di un blocco di 100 parole? Utilizzando il valore di CPI delle istruzioni per l'implementazione multi-ciclo, quanti cicli sarebbero richiesti per la copia di 100 parole?

5.27 [30] <§5.5> {Es. 5.26} Il microcodice è stato utilizzato per aggiungere istruzioni più potenti al set di istruzioni: si esplorino i possibili benefici di questo approccio. Si identifichi una strategia per implementare l'istruzione `bcp` descritta nell'Esercizio 5.26, utilizzando le unità di elaborazione e di controllo multi-ciclo. Per implementare efficientemente l'istruzione, sarà probabilmente necessario apportare delle modifiche all'unità di elaborazione: si descrivano le modifiche proposte e si descriva il funzionamento dell'istruzione `bcp`. Vi sono dei vantaggi che si potrebbero ottenere dall'aggiunta di registri dedicati all'istruzione `bcp` nell'unità di elaborazione? Si stimi l'incremento di prestazioni ottenibile dall'implementazione hardware dell'istruzione rispetto alla soluzione software ottenuta nell'Esercizio 5.26 e si spieghi da dove deriva l'aumento delle prestazioni.

5.28 [30] <§5.5> {Es. 5.27} Utilizzando la strategia sviluppata nell'Esercizio 5.27, si modifichi il formato di microistruzione MIPS descritto nella figura 5.45 e si fornisca il microprogramma completo per l'istruzione `bcp`. Si descriva in dettaglio come estendere il microcodice in modo da supportare la creazione di strutture di controllo più complesse (come i cicli) a livello di microcodice. Il supporto all'istruzione `bcp` ha modificato la dimensione del microcodice? Il cambiamento del formato delle microistruzioni ha effetto su altre istruzioni?

5.29 [15] <§5.6> Si vuole aggiungere l'istruzione `rfe` (return from exception, ritorno da un'eccezione) all'unità di elaborazione multi-ciclo di questo capitolo; lo scopo principale dell'istruzione `rfe` è copiare il contenuto di EPC in PC (il meccanismo delle eccezioni richiederà molte funzionalità aggiuntive, che verranno discusse nel Capitolo 7). Si aggiungano le unità funzionali necessarie all'unità di elaborazione multi-ciclo di figura 5.48 e si indichino le modifiche necessarie alla macchina a stati finiti delle figure 5.49 e 5.50. Si possono fotocopiare le figure o scaricarle dall'indirizzo *www.mkp.com/cod2e.htm* per facilitare l'indicazione delle aggiunte.

5.30 [1 settimana] <§§5.2, 5.3> Utilizzando un linguaggio di simulazione dell'hardware come Verilog, si implementi un simulatore funzionale per la versione a singolo ciclo del progetto. Si scriva il simulatore utilizzando una libreria di parti standard, se disponibile. Se le parti contengono informazioni temporali, si determini il periodo di clock dell'implementazione ottenuta.

5.31 [1 settimana] <§§5.2, 5.4, 5.5> Utilizzando un linguaggio di simulazione dell'hardware come Verilog, si implementi un simulatore funzionale per la versione multi-ciclo del progetto. Si scriva il simulatore utilizzando una libreria di parti standard, se disponibile. Se le parti contengono informazioni temporali, si determini il periodo di clock dell'implementazione ottenuta.

5.32 [2-3 mesi] <§§5.1-5.3> Si costruisca, utilizzando parti standard, un sistema che implementi la macchina a singolo ciclo di questo capitolo.

5.33 [2-3 mesi] <§§5.1-5.8> Si costruisca, utilizzando parti standard, un sistema che implementi la macchina multi-ciclo di questo capitolo.

5.34 [Discussione] <§§5.5, 5.8, 5.9> Ipotesi: se la prima implementazione di un'architettura è microprogrammata, ciò ha effetti sul set di istruzioni. Perché dovrebbe essere vero? Si trovi, se esiste, un'architettura che probabilmente utilizzerà sempre il microcodice. Perché? Quali macchine non utilizzeranno mai il microcodice? Perché? Quale implementazione dell'unità di controllo aveva probabilmente in mente il progettista quando ha progettato l'architettura del set di istruzioni?

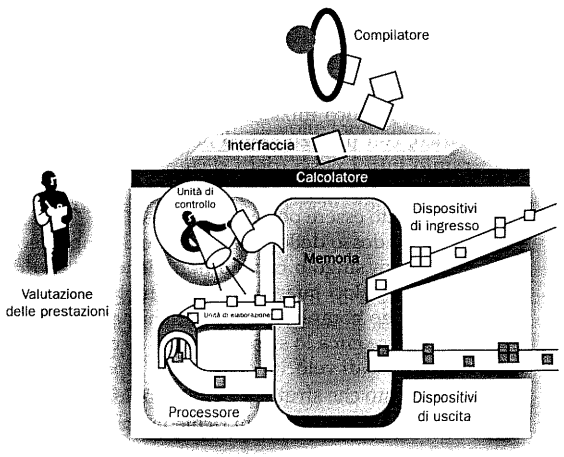
5.35 [Discussione] <§§5.5, 5.10> Wilkes inventò la microprogrammazione principalmente per semplificare la costruzione dell'unità di controllo. Dal 1980 in poi vi è stata un'esplosione di software di ausilio alla progettazione, dedicato alla semplificazione della costruzione delle unità di controllo, rendendone molto più semplice il progetto. Si cerchino delle prove, basandosi sui software CAD o su progetti reali, che supportino o confutino tale ipotesi.

5.36 [Discussione] <§5.10> Le istruzioni MIPS e le microistruzioni MIPS hanno molte somiglianze. Quali sarebbero le difficoltà, per un compilatore, a generare il microcodice MIPS anziché il linguaggio macchina? Quali modifiche alla microarchitettura renderebbero maggiormente utile questa soluzione?

6

Miglioramento delle prestazioni mediante l'uso di pipeline

Le cinque componenti classiche del calcolatore



Valutazione delle prestazioni

Il tempo scorre,
per ogni cosa
viene il suo turno;
nuove cose si susseguono,
mentre le altre invecchiano.

Robert Herrick
Hesperides: Ceremonies for Christmas Eve, 1648

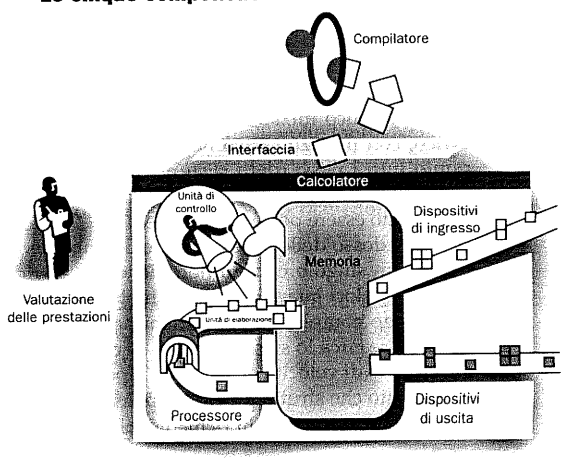
5.35 [Discussione] <§§5.5, 5.10> Wilkes inventò la microprogrammazione principalmente per semplificare la costruzione dell'unità di controllo. Dal 1980 in poi vi è stata un'esplosione di software di ausilio alla progettazione, dedicato alla semplificazione della costruzione delle unità di controllo, rendendone molto più semplice il progetto. Si cerchino delle prove, basandosi sui software CAD o su progetti reali, che supportino o confutino tale ipotesi.

5.36 [Discussione] <§5.10> Le istruzioni MIPS e le microistruzioni MIPS hanno molte somiglianze. Quali sarebbero le difficoltà, per un compilatore, a generare il microcodice MIPS anziché il linguaggio macchina? Quali modifiche alla microarchitettura renderebbero maggiormente utile questa soluzione?

6

Miglioramento delle prestazioni mediante l'uso di pipeline

Le cinque componenti classiche del calcolatore



Il tempo scorre,
per ogni cosa
viene il suo turno;
nuove cose si susseguono,
mentre le altre invecchiano.

Robert Herrick
Hesperides: Ceremonies for Christmas Eve, 1648

6.1 Vista d'insieme del progetto mediante pipeline

Il tempo non va sprecato.

Proverbio americano

La tecnica implementativa basata sull'uso di *pipeline* (letteralmente "oleodotto", termine usato anche per indicare le catene di montaggio) prevede la sovrapposizione temporale dell'esecuzione di diverse istruzioni; al giorno d'oggi le pipeline sono la chiave su cui si basa la velocità dei calcolatori.

Questo paragrafo utilizza pesantemente un'analogia al fine di fornire una vista d'insieme sui termini e sulle problematiche legate all'uso di pipeline. I lettori interessati solamente al quadro d'insieme possono concentrarsi su questo paragrafo e saltare direttamente al Paragrafo 6.8 per vedere il funzionamento delle pipeline e le relative implicazioni sulle prestazioni dei programmi; viceversa i lettori interessati all'anatomia dei calcolatori dotati di pipeline troveranno dei continui richiami a questo paragrafo nei Paragrafi dal 6.2 al 6.7.

Chiunque abbia avuto a che fare con il bucato ha già utilizzato, almeno in modo intuitivo, la tecnica delle pipeline. Un approccio al bucato di tipo classico, ossia *senza pipeline*, potrebbe essere il seguente:

1. Mettere un carico di abiti sporchi nella lavatrice.
2. Quando la lavatrice ha finito, mettere il carico bagnato nell'asciugatrice.
3. Quando l'asciugatrice ha finito, mettere il carico asciutto sull'asse da stiro e stirarlo.
4. Dopo aver finito di stirare, chiedere ad un amico di riporre gli abiti nell'armadio.

Appena l'amico avrà finito si potrà ricominciare con il prossimo carico di panni sporchi.

L'approccio con *pipeline* richiede molto meno tempo, come illustrato nella figura 6.1: appena la lavatrice ha terminato il primo carico e questo è stato trasferito nell'asciugatrice, si può riempire la lavatrice con il secondo carico di panni sporchi. Quando il primo carico è asciutto, lo si pone sull'asse da stiro per stirarlo, si sposta il carico bagnato nell'asciugatrice ed il successivo carico sporco nella lavatrice. In seguito l'amico inizierà a ritirare gli abiti mentre partirà la stiratura del secondo carico, con il terzo carico nell'asciugatrice ed il quarto nella lavatrice. A questo punto tutti i passi (detti *stadi* nel linguaggio delle pipeline) stanno lavorando in contemporanea: purché si abbiano risorse separate per ciascuno stadio, le operazioni possono essere poste in pipeline.

Il paradosso delle pipeline è che il tempo necessario a lavare, asciugare, stirare e riporre un singolo calzino non viene affatto ridotto; il motivo per cui le pipeline sono più veloci quando si hanno molti panni è che tutto lavora in parallelo, quindi si possono completare più carichi nella stessa quantità di tempo. Se ogni stadio richiedesse circa lo stesso tempo e se ci fosse abbastanza lavoro da fare, il fattore di incremento di velocità dovuto alla pipeline sarebbe pari al numero di stadi.

Il bucato che sfrutta pipeline è potenzialmente quattro volte più veloce del bucato tradizionale: 20 carichi richiederebbero circa 5 volte il tempo richiesto da un carico, mentre 20 carichi sequenziali impegnerebbero 20 volte tanto. Nella figura 6.1 la pipeline è solo 2,3 volte più veloce perché vi sono solamente 4 carichi.

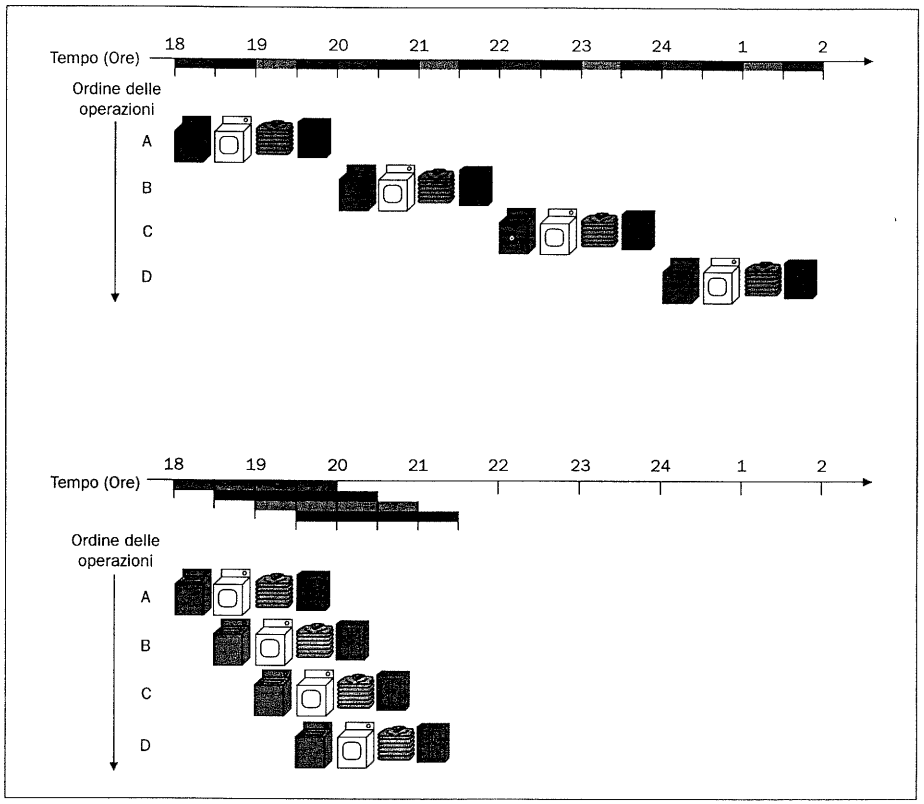


Figura 6.1 L'analogia del bucato per la pipeline. Anna, Bruno, Carla e Dario hanno panni sporchi da lavare, asciugare, stirare e riporre. Il lavaggio, l'asciugatura, la stiratura ed il riordino sono compiti che richiedono circa 30 minuti ciascuno. Il bucato sequenziale richiede 8 ore per quattro carichi di lavaggio, mentre il bucato con pipeline richiede solamente 3,5 ore. Lo stadio di avanzamento dei diversi carichi nella pipeline al passare del tempo è indicato dalle copie delle quattro risorse sulla tabella temporale bidimensionale, ma in realtà vi è una sola risorsa per tipo.

Gli stessi principi si applicano ai processori, dove ciò che si vuole porre in pipeline è l'esecuzione delle istruzioni; le istruzioni MIPS richiedono tipicamente cinque passi:

1. Prelievo dell'istruzione dalla memoria.
2. Lettura dei registri e decodifica dell'istruzione (il formato delle istruzioni MIPS permette la simultaneità tra la lettura e la decodifica).
3. Esecuzione dell'operazione o calcolo dell'indirizzo.
4. Accesso ad un operando nella memoria dati.
5. Scrittura del risultato in un registro.

Di conseguenza la pipeline del MIPS analizzata in questo capitolo avrà cinque stadi. L'esempio seguente mostra come il meccanismo delle pipeline velocizzi l'esecuzione delle istruzioni nello stesso modo in cui velocizza il bucato.

Confronto tra le prestazioni di un processore a singolo ciclo ed uno con pipeline

Esempio

Per concretizzare il discorso si consideri una pipeline ben precisa; in questo esempio, così come nel resto del capitolo, si limiterà l'attenzione ad otto istruzioni: load word (`lw`), store word (`sw`), add (`add`), subtract (`sub`), and (`and`), or (`or`), set-less-than (`slt`) e branch-on-equal (`beq`).

Si confronti il tempo medio richiesto da queste istruzioni in un'implementazione a singolo ciclo, in cui tutte le istruzioni richiedono 1 ciclo di clock, con un'implementazione con pipeline. La durata delle operazioni per le unità funzionali più importanti di questo esempio è di 2 ns per l'accesso alla memoria, 2 ns per le operazioni con la ALU e 1 ns per letture o scritture del register file (come detto nel Capitolo 5, nel modello a singolo ciclo ogni istruzione impiega esattamente un ciclo di clock, per cui il ciclo di clock va dilatato in modo da contenere l'istruzione più lenta).

Soluzione

Il tempo richiesto da ciascuna delle otto istruzioni è indicato in figura 6.2. Il progetto a singolo ciclo deve permettere l'esecuzione dell'istruzione più lenta (nella figura 6.2 questa è `lw`), per cui il tempo richiesto da ciascuna istruzione sarà 8 ns. In modo simile alla figura 6.1, la figura 6.3 confronta l'esecuzione con e senza pipeline di tre istruzioni load word: il tempo trascorso tra la prima e la quarta istruzione nel progetto senza pipeline è pari a 3×8 ns, ossia 24 ns.

Ogni stadio della pipeline richiede un ciclo di clock, per cui il periodo del clock deve essere abbastanza lungo da contenere l'operazione più lenta: così come il progetto a singolo ciclo deve usare il caso peggiore di 8 ns anche se alcune istruzioni potrebbero richiedere solo 5 ns, il ciclo di clock per la pipeline deve essere posto al valore del caso peggiore, cioè a 2 ns, anche se alcuni stadi richiederebbero solo 1 ns. La pipeline offre un miglioramento delle prestazioni di quattro volte: l'intervallo di tempo tra la prima e la quarta istruzione è di 3×2 ns, ossia 6 ns.

Classe dell'istruzione	Prelievo dell'istruzione	Lettura dei registri	Operazione della ALU	Accesso ai dati	Scrittura dei registri	Tempo totale
Load word (`lw`)	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store word (`sw`)	2 ns	1 ns	2 ns	2 ns		7 ns
Formato-R (`add, sub, and, or, slt`)	2 ns	1 ns	2 ns		1 ns	6 ns
Salto (`beq`)	2 ns	1 ns	2 ns			5 ns

Figura 6.2 Durata totale per le otto istruzioni, calcolata a partire dal ritardo di ciascun componente. Il calcolo assume che i multiplexer, l'unità di controllo, l'accesso al PC e il circuito di controllo del segno non abbiano ritardo.

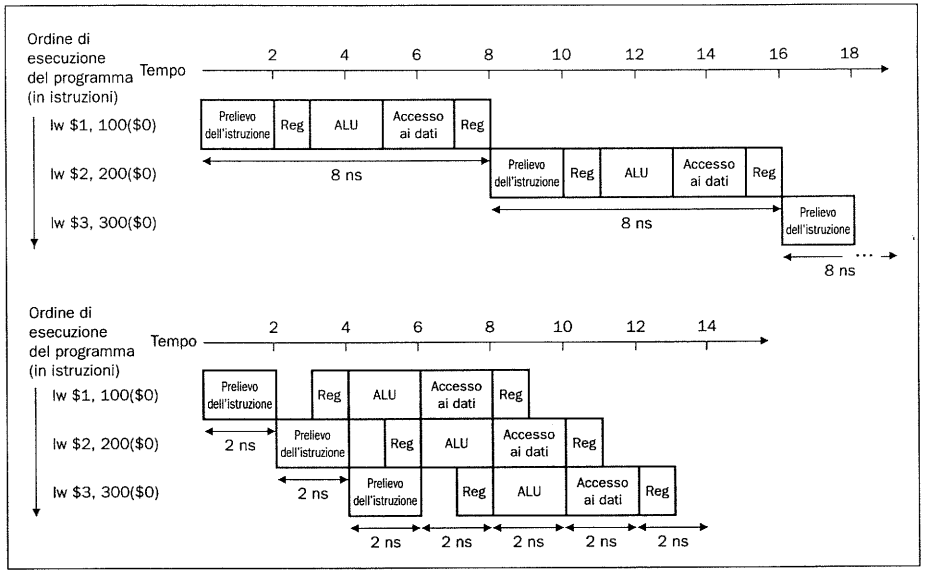


Figura 6.3 Esecuzione a singolo ciclo senza pipeline (in alto) confrontata con l'esecuzione con pipeline (in basso). In entrambi i casi si usano gli stessi componenti hardware, il cui ritardo è elencato in figura 6.2. In questo esempio si nota un incremento di velocità di un fattore quattro nel tempo medio richiesto dalle istruzioni, che scende da 8 ns a 2 ns. Si confronti questa figura con la 6.1: per il bucato si era ipotizzato che tutti gli stadi avessero pari durata. Se l'asciugatrice fosse stata più lenta, avrebbe determinato la durata di ciascuno stadio; anche nell'elaboratore la durata dello stadio è determinata dalla risorsa più lenta: l'operazione con la ALU o l'accesso alla memoria. Si assume che la scrittura nel register file avvenga nella prima metà del ciclo di clock e che la lettura dal register file abbia luogo nella seconda metà; tale assunzione sarà valida in tutto il capitolo.

È possibile ricavare una formula dalla precedente discussione sull'incremento di velocità: se gli stadi sono perfettamente bilanciati, il tempo tra le istruzioni in un sistema con pipeline (in condizioni ideali) è pari a

$$\text{Tempo tra le istruzioni}_{\text{con pipeline}} = \frac{\text{Tempo tra le istruzioni}_{\text{senza pipeline}}}{\text{Numero di stadi}}$$

In condizioni ideali l'incremento di velocità dovuto alla pipeline è pari al numero dei suoi stadi: una pipeline a cinque stadi sarebbe cinque volte più veloce.

La formula suggerisce che una pipeline a cinque stadi dovrebbe migliorare di cinque volte il periodo di 8 ns della versione senza pipeline, ossia avere un clock di 1,6 ns. L'esempio invece mostra che gli stadi potrebbero non essere perfettamente bilanciati; inoltre il meccanismo della pipeline introduce degli overhead. In definitiva il tempo per istruzione in un sistema con pipeline sarà maggiore del minimo possibile, per cui lo speedup sarà minore del numero di stadi della pipeline.

Inoltre, considerando l'esempio non si ottiene neppure un miglioramento del tempo totale di esecuzione delle tre istruzioni pari ad un fattore quattro: si sono infatti ottenuti 14 ns rispetto a 24 ns. Per capire perché il tempo totale di esecuzione non è una quantità così interessante, si consideri che cosa avverrebbe aumentando il numero di istruzioni. Si pensi di estendere i dati precedenti a 1003 istruzioni:

aggiungendo 1000 istruzioni all'esempio con pipeline, ogni istruzione contribuirebbe ad incrementare di 2 ns il tempo di esecuzione totale. Esso sarebbe pari a 1000 × 2 ns+14 ns, cioè 2014 ns. Nel caso senza pipeline, le 1000 istruzioni aggiunte richiederebbero 8 ns ciascuna, quindi il tempo di esecuzione totale sarebbe 1000 × 8 ns+24 ns, o 8024 ns. In queste condizioni ideali, per programmi di dimensioni normali il rapporto tra i tempi di esecuzione in sistemi senza pipeline rispetto a quelli con pipeline è vicino al rapporto tra le relative durate delle istruzioni:

$$\frac{8024\,\text{ns}}{2014\,\text{ns}} = 3,98 \approx \frac{8\,\text{ns}}{2\,\text{ns}}$$

La tecnica delle pipeline migliora le prestazioni *aumentando il throughput delle istruzioni, e non riducendo il tempo di esecuzione di ogni singola istruzione*. Fortunatamente il throughput delle istruzioni è la metrica più importante, dal momento che i programmi reali eseguono miliardi di istruzioni.

Progetto dei set di istruzioni orientati alle architetture con pipeline

Anche a partire dalla semplice descrizione del principio di funzionamento delle pipeline, è possibile intuire alcuni elementi sull'architettura del set di istruzioni del MIPS, che è stato progettato espressamente per l'esecuzione con processori dotati di pipeline.

Innanzitutto tutte le istruzioni MIPS hanno la stessa lunghezza: tale restrizione semplifica notevolmente il prelievo delle istruzioni nel primo stadio della pipeline e la loro decodifica nel secondo stadio. In un set di istruzioni come quello dell'80x86, in cui le istruzioni variano da 1 byte a 17 byte, il progetto della pipeline sarebbe notevolmente più impegnativo.

In secondo luogo il MIPS ha un numero ridotto di formati delle istruzioni ed in tutti i formati i registri sorgente sono sempre posti nelle stesse posizioni, qualunque sia l'istruzione. Tale simmetria permette al secondo stadio di iniziare a leggere il register file nello stesso tempo in cui il circuito determina il tipo dell'istruzione reperita: se il formato delle istruzioni MIPS non fosse stato simmetrico, sarebbe stato necessario spezzare lo stadio 2, portando la pipeline ad un totale di 6 stadi (si vedrà tra non molto lo svantaggio di avere pipeline più lunghe).

In terzo luogo nel MIPS gli operandi di tipo memoria possono comparire solo nelle istruzioni di load e di store: questo vincolo permette di usare lo stadio di esecuzione per calcolare l'indirizzo di memoria, accedendo alla memoria nello stadio successivo. Se si potessero effettuare delle operazioni sugli operandi in memoria, come avviene nell'80x86, gli stadi 3 e 4 dovrebbero essere espansi in uno stadio per l'indirizzo, uno per la memoria ed un terzo per l'esecuzione.

Un quarto fattore è l'allineamento obbligatorio degli operandi in memoria (si veda il capoverso Interfaccia Hardware/Software del Paragrafo 3.3): non è necessario considerare casi in cui una singola istruzione di trasferimento dati richieda due accessi alla memoria, per cui i dati possono essere trasferiti tra il processore e la memoria in un solo stadio della pipeline.

Criticità nelle pipeline

In un'architettura basata sull'uso di pipeline possono sorgere delle situazioni in cui l'istruzione successiva non può essere eseguita nel ciclo di clock immediatamente

seguente; tali eventi sono detti *criticità* (il termine inglese utilizzato è *hazard*, ossia rischio). Verranno esposti tre tipi di criticità, prima sfruttando l'analogia con il ciclo del bucato ed in seguito esponendo il corrispondente problema nei processori per delinearne la soluzione.

Criticità strutturali

La condizione denominata come *criticità strutturale* indica che le risorse hardware presenti non sono in grado di supportare la combinazione di istruzioni che si vorrebbe eseguire nello stesso ciclo di clock. Nel caso del bucato si verificherebbe una criticità strutturale se si utilizzasse un'unica macchina lavatrice/asciugatrice anziché due macchine separate, oppure se l'amico fosse impegnato in qualche altro compito e non potesse riporre gli abiti. In tali casi il piano di un bucato con pipeline andrebbe in fumo.

Come detto sopra, il set di istruzioni MIPS fu progettato appositamente per essere compatibile con le pipeline, facendo in modo che per i progettisti fosse semplice evitare le criticità strutturali durante il progetto della pipeline. Si supponga tuttavia di disporre di una sola memoria anziché di due: se nella pipeline di figura 6.3 vi fosse una quarta istruzione, nello stesso ciclo di clock la prima istruzione dovrebbe accedere ai dati nella memoria, mentre la quarta istruzione dovrebbe essere reperita dalla stessa memoria. Non disponendo di due memorie, l'architettura con pipeline presenterebbe una criticità strutturale.

Criticità sul controllo

Un secondo tipo di criticità sono quelle *sul controllo*, che sono determinate dalla necessità di dover compiere alcune scelte in funzione del risultato di un'istruzione mentre altre istruzioni sono in esecuzione.

Si supponga che al personale della lavanderia venga affidato l'infelice compito di lavare le divise di una squadra di calcio: a seconda di quanto queste siano sporche, occorre determinare la quantità di detersivo e la temperatura dell'acqua in modo da lavare in modo abbastanza energico da pulire le divise, ma non troppo energico, per evitare un'usura troppo rapida. Nella pipeline per il bucato è necessario attendere il completamento del secondo stadio, esaminando le divise asciutte, prima di poter decidere se sia necessario modificare i parametri della lavatrice. Cosa si può fare in questo caso?

Nel seguito si esamineranno due soluzioni alle criticità sul controllo nel caso del bucato, fornendone l'equivalente per gli elaboratori.

> *Stallo:* Lavorare in modo sequenziale fino a quando il primo carico è asciutto e se il risultato ottenuto non è quello desiderato ripetere il ciclo finché non si ottiene la combinazione corretta. Quest'opzione conservativa è certamente corretta, ma lenta.

L'equivalente di un'operazione di scelta in un elaboratore è l'istruzione di salto condizionato: se l'elaboratore dovesse entrare in stallo sui salti condizionati, dovrebbe sospendere l'esecuzione mediante pipeline procedendo in modo sequenziale fino a terminare l'istruzione di salto. Si ipotizzi di disporre di sufficienti risorse hardware in modo da poter confrontare i registri, calcolare l'indirizzo di destinazione del salto ed aggiornare il PC nel secondo stadio (per i dettagli si veda il Paragrafo 6.6): anche disponendo di tali circuiti aggiuntivi, in presenza di salti condizionati la pipeline si comporterebbe come quella di figura 6.4. L'istruzione `lw`,

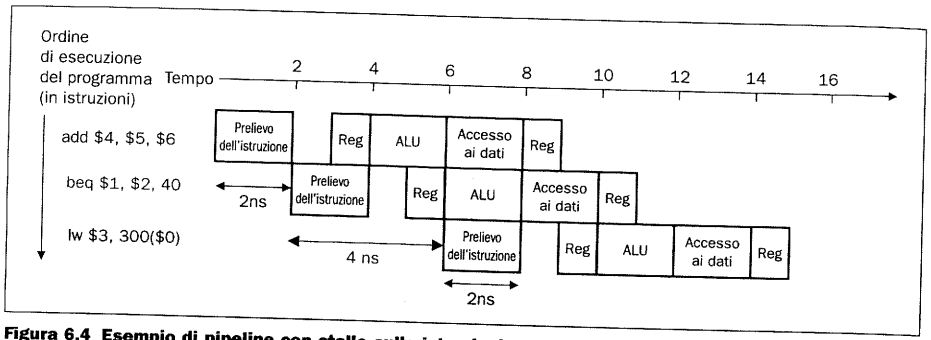


Figura 6.4 Esempio di pipeline con stallo sulle istruzioni di salto condizionato come possibile soluzione alle criticità sul controllo. Dopo l'istruzione di salto condizionato vi è uno stallo della pipeline, o buco, corrispondente ad uno stadio.

eseguita se il salto *non* viene effettuato, viene posta in stato di attesa (o di stallo) per un ciclo di clock aggiuntivo della durata di 2 ns, prima di iniziare la sua esecuzione. La figura mostra un concetto importante sull'uso delle pipeline, detto ufficialmente *stallo della pipeline*, ma spesso gergalmente indicato con *buco* (in inglese *bubble*, cioè bolla, ricordando un vuoto d'aria nell'oleodotto). In seguito si vedranno stalli in altre parti della pipeline.

Esempio

Influenza sulle prestazioni dello stallo dovuto ai salti condizionati

Si stimi l'impatto sul numero di cicli di clock per istruzione (CPI) dello stallo dovuto alle istruzioni di salto condizionato. Si assuma che tutte le altre operazioni abbiano un CPI pari ad 1.

Soluzione

La figura 3.38 del Capitolo 3 indica che i salti condizionati sono il 17% delle istruzioni eseguite nel programma gcc. Siccome le altre istruzioni eseguite hanno un CPI pari ad 1 ed il salti richiedono un ciclo di clock aggiuntivo dovuto allo stallo, si avrebbe un CPI pari ad 1,17 e conseguentemente un fattore di rallentamento di 1,17 rispetto al caso ideale (si noti che la figura 3.38 considera anche `slt` e `slti` tra le istruzioni di salto condizionato, anche se queste non introdurrebbero uno stallo, per cui il valore del CPI è solo un'approssimazione).

Se non si potesse decidere l'esito dell'istruzione di salto nel secondo stadio, come spesso avviene in pipeline più lunghe, si incorrerebbe in un rallentamento ancora maggiore a causa degli stalli sulle istruzioni di salto condizionato. In quasi tutti gli elaboratori il costo introdotto da questa soluzione sarebbe troppo elevato per prenderla in considerazione, per cui si ricorre ad una seconda possibilità per risolvere le criticità sul controllo:

> *Predizione:* Se si è quasi sicuri di conoscere la combinazione giusta per lavare le divise, si assume di essere in grado di fare una predizione corretta e si lava il secondo carico mentre il primo si asciuga. In tal modo non si rallenta la pipeline quando la predizione si rivela corretta. In caso di predizione errata, invece, occorre ripetere la lavorazione del carico che era stato lavato mentre si stava cercando di indovinare la combinazione.

In effetti i calcolatori usano la tecnica della predizione per gestire i salti condizionati. L'approccio più semplice consiste nel predire sempre che il salto non venga eseguito: in caso di predizione corretta la pipeline procede al massimo della velocità, mentre solo quando i salti vengono eseguiti si ha uno stallo della pipeline. I due casi possibili sono esemplificati nella figura 6.5.

Una versione più sofisticata della predizione dei salti richiederebbe di predire alcuni salti come eseguiti ed altri come non eseguiti. Nell'analogia le uniformi scure richiederebbero una combinazione di parametri mentre quelle chiare un'altra. Nel caso degli elaboratori si considerino ad esempio i salti che si trovano al termine dei cicli che servono a ritornare all'inizio dell'iterazione successiva: poiché essi verranno probabilmente eseguiti e poiché saltano all'indietro, il processore potrebbe predire che i salti verso indirizzi minori verranno eseguiti.

Questi approcci rigidi alla predizione dei salti ipotizzano un comportamento che ricade in precisi schemi e non tengono conto dell'individualità delle specifiche istruzioni di salto. I circuiti per la predizione *dinamica* dei salti, in netto contrasto, scelgono la predizione da fare in funzione del comportamento di ciascuna istruzione di salto, e possono modificare la predizione fatta per un certo salto durante l'esecuzione del programma. Secondo l'analogia, nella predizione dinamica ci sarebbe una persona per esaminare il grado di sporcizia della divisa e decidere di conse-

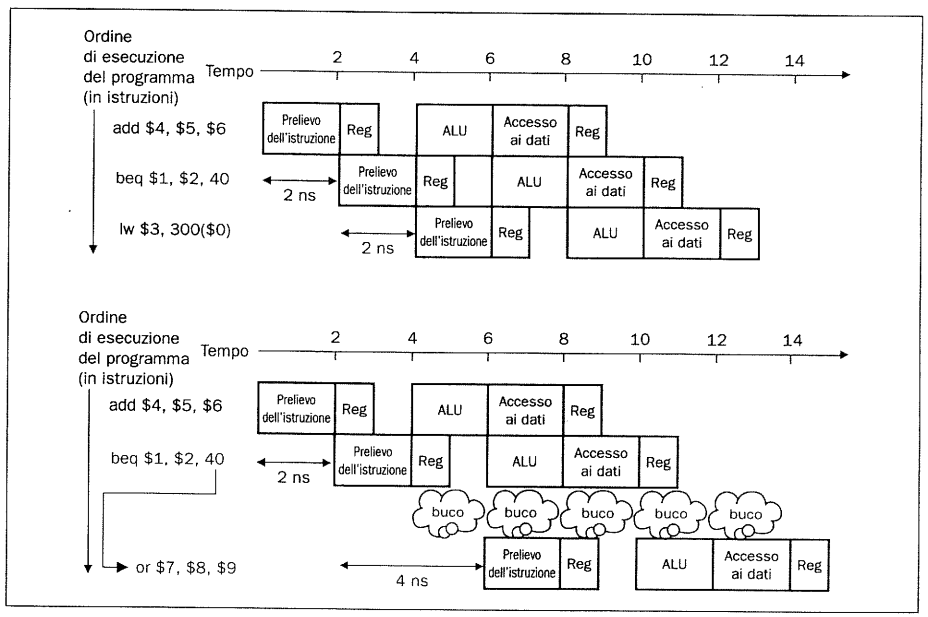


Figura 6.5 Esempio di pipeline in cui le criticità sul controllo sono risolte predicendo che le istruzioni di salto condizionato non vengano eseguite. Il disegno mostra la pipeline quando il salto non viene eseguito (sopra) e quando il salto viene eseguito (sotto).

guenza i parametri, tarando le scelte future sulla base del successo delle scelte fatte più di recente. Un approccio diffuso per implementare la predizione dinamica nei processori consiste nel memorizzare la storia di ciascun salto, ricordando se è stato eseguito o no, e nell'utilizzare il passato per predire il futuro; un circuito di questo tipo ha una precisione pari al 90% circa (si veda il Paragrafo 6.6). Quando la predizione si rivela errata, la logica di controllo della pipeline deve garantire che le istruzioni che seguivano quella di salto non abbiano alcun effetto e deve far ripartire la pipeline dall'indirizzo corretto.

Come avviene in tutte le altre soluzioni che riguardano le criticità sul controllo, il problema si aggrava al crescere della lunghezza della pipeline, poiché aumenta il costo delle predizioni errate. Le diverse soluzioni alle criticità sul controllo sono descritte in maggior dettaglio nel Paragrafo 6.6.

Approfondimento: Vi è un terzo approccio possibile alla soluzione delle criticità sul controllo, detto *decisione ritardata*. Seguendo l'analogia, ogni volta che occorre prendere una decisione sul bucato, si pone un carico di abiti diversi dalle divise dei calciatori nella lavatrice, nell'attesa che le divise precedenti si asciughino. A patto di disporre di una sufficiente quantità di abiti sporchi che non dipendano dall'esito della scelta, questa soluzione funziona correttamente.

Tale soluzione, detta dei *salti ritardati*, è quella effettivamente adottata dall'architettura MIPS: un salto ritardato esegue sempre l'istruzione sequenzialmente successiva, mentre il salto avviene *dopo* un ritardo pari all'esecuzione dell'istruzione. Questo fatto è invisibile al programmatore in linguaggio assembler in quanto l'assemblatore è in grado di riaggiustare automaticamente le istruzioni in modo da ottenere il comportamento previsto dal programmatore per i diversi salti. I programmi per il MIPS pongono sempre un'istruzione non dipendente dal salto immediatamente dopo l'istruzione di salto ritardato, per cui l'esecuzione del salto modifica l'indirizzo dell'istruzione che *segue* questa istruzione "sicura". Nell'esempio di figura 6.4, l'istruzione `add` prima della branch non ha effetti sul salto, per cui nella figura 6.6 è stata posta nella posizione che segue l'istruzione di salto e che è detta *delayed branch slot* (posizione relativa al salto ritardato).

I compilatori tipicamente sono in grado di riempire circa il 50% delle posizioni relative alle istruzioni di salto con istruzioni utili. Se la pipeline fosse più lunga di cinque stadi si potrebbero avere più slot per ciascun salto ritardato, che risulterebbero ancora più difficili da riempire.

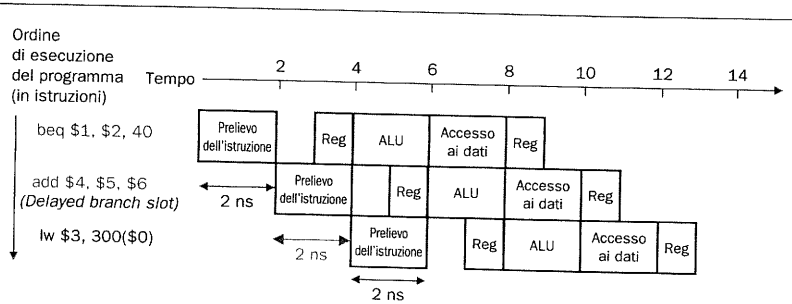


Figura 6.6 Esempio di pipeline con salti ritardati come possibile soluzione alle criticità sul controllo. Il "buco" della pipeline è stato riempito con un'istruzione `add`.

Criticità sui dati

Ritornando alla lavanderia, si supponga di dover riporre un carico che è composto prevalentemente da calzini: il caso più sfortunato è quello in cui il calzino corrispondente a ciascuno di quelli del carico corrente è stato posto in un altro carico, ancora nella lavatrice. Non si possono appaiare i calzini e riporli fino a che il secondo carico non è terminato, per cui è necessario bloccare la pipeline. Il problema analogo nel caso degli elaboratori è detto *criticità sui dati*: un'istruzione dipende dal risultato di un'istruzione precedente che si trova ancora nella pipeline.

Si supponga ad esempio di avere un'istruzione di somma immediatamente seguita da una sottrazione che utilizza il risultato della somma:

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

Se non si prendessero contromisure, le criticità sui dati potrebbero causare dei seri stalli nella pipeline: l'istruzione `add` non scrive il proprio risultato prima del quinto stadio, per cui si dovrebbero aggiungere tre "buchi" alla pipeline.

Se si pensasse di affidare ai compilatori il compito di evitare le criticità sui dati, il risultato non sarebbe soddisfacente: queste dipendenze sono troppo frequenti ed il ritardo è troppo lungo per poter sperare che il compilatore possa risolvere il problema.

La soluzione più utilizzata è basata sull'osservazione che non è necessario attendere il termine dell'istruzione per cercare di risolvere la criticità sui dati: nella sequenza di codice appena vista, non appena la ALU genera il risultato della somma, questo può venire usato come ingresso per l'operazione di sottrazione. In tal modo si prende in anticipo il dato mancante attingendo alle risorse interne; tale operazione è detta *propagazione* (*forwarding*) o *scavalcamento* (*bypassing*).

Propagazione nel caso di due istruzioni

Esempio Nel caso delle due istruzioni sopra riportate, si mostri quali stadi della pipeline dovrebbero essere collegati con la tecnica della propagazione. Si usi lo schema della figura 6.7 per rappresentare l'unità di elaborazione nei cinque stadi della pipeline: si faccia una copia dell'unità di elaborazione per ciascuna istruzione, analogamente all'esempio del bucato di figura 6.1.

Soluzione La figura 6.8 mostra i collegamenti necessari a propagare il valore di `$s0` calcolato nello stadio di esecuzione dell'istruzione `add` fino ad arrivare all'ingresso dello stadio di esecuzione dell'istruzione `sub`.

Usando questa rappresentazione grafica degli eventi, i cammini di propagazione risultano validi solo se lo stadio di destinazione è successivo, nel tempo, a quello sorgente. Per esempio, non vi potrebbe essere un cammino di propagazione valido tra l'uscita dello stadio di accesso alla memoria della prima istruzione e l'ingresso dello stadio di esecuzione dell'istruzione successiva, poiché ciò vorrebbe dire muoversi a ritroso nel tempo.

La tecnica della propagazione è molto utile ed è descritta in dettaglio nel Paragrafo 6.4; tuttavia non è in grado di prevenire tutti gli stalli della pipeline. Si

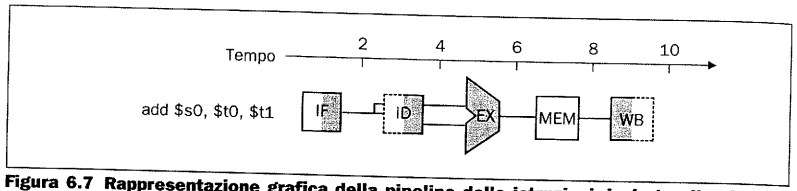


Figura 6.7 Rappresentazione grafica della pipeline delle istruzioni, ispirata alla pipeline del bucato di figura 6.1. In tutto il capitolo si utilizzeranno dei simboli grafici per rappresentare le risorse fisiche e delle abbreviazioni per i nomi degli stadi. Le sigle dei cinque stadi sono le seguenti: *IF* per lo stadio di prelievo dell'istruzione (da Instruction Fetch), in cui il quadrato rappresenta la memoria delle istruzioni; *ID* per lo stadio di decodifica delle istruzioni e lettura da register file (da Instruction Decode), in cui il simbolo rappresenta il register file che viene letto; *EX* per lo stadio di esecuzione, dove il disegno rappresenta la ALU; *MEM* per lo stadio di accesso alla memoria, in cui il quadrato rappresenta la memoria dei dati; WB per lo stadio di scrittura dei risultati (da Write Back), in cui il simbolo rappresenta il register file che viene scritto. Gli stadi evidenziati sono quelli utilizzati dall'istruzione; per esempio, MEM è rappresentato in bianco perché `add` non fa accesso alla memoria dei dati. L'evidenziazione della metà destra del register file o della memoria indica che tale elemento viene letto in quello stadio, mentre l'evidenziazione della metà sinistra si riferisce all'operazione di scrittura. Ad esempio la metà destra di ID nel secondo stadio è stata evidenziata poiché il register file viene letto, mentre nel quinto stadio la scrittura del register file è indicata evidenziando la metà sinistra di WB.

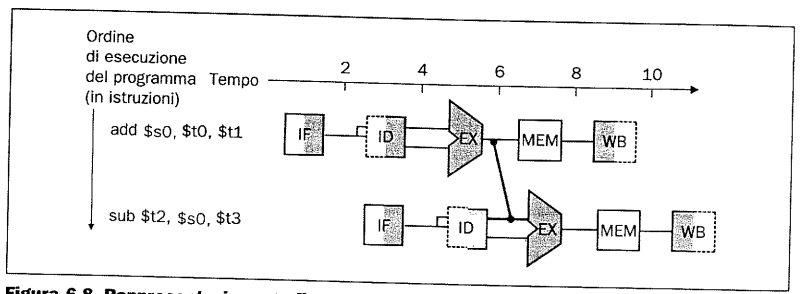


Figura 6.8 Rappresentazione grafica della propagazione. Il collegamento indica il cammino di propagazione dall'uscita dello stadio EX dell'istruzione `add` all'ingresso dello stadio EX dell'istruzione `sub`, sostituendo quindi il valore del registro `$s0` letto nel secondo stadio della `sub`.

supponga ad esempio che la prima istruzione sia un caricamento di `$s0` anziché una `add`: come si può immaginare esaminando la figura 6.8, il dato voluto sarebbe disponibile solo *dopo* il quarto stadio della prima istruzione, ossia troppo tardi per essere utilizzato come *ingresso* del terzo stadio della sottrazione. Di conseguenza, anche disponendo del meccanismo di propagazione, si dovrebbe imporre uno stallo della durata di uno stadio a causa di una *criticità sui dati di tipo carica-e-usa* (in inglese *load-use data hazard*), illustrata in figura 6.9. Il Paragrafo 6.5 indica come i circuiti dedicati alla pipeline possono risolvere i casi difficili di questo tipo.

Esempio

Riordinare il codice per evitare stalli nella pipeline

Si identifichi la criticità nel codice seguente, tratto dal corpo della procedura `scambia` riportata in figura 3.23:

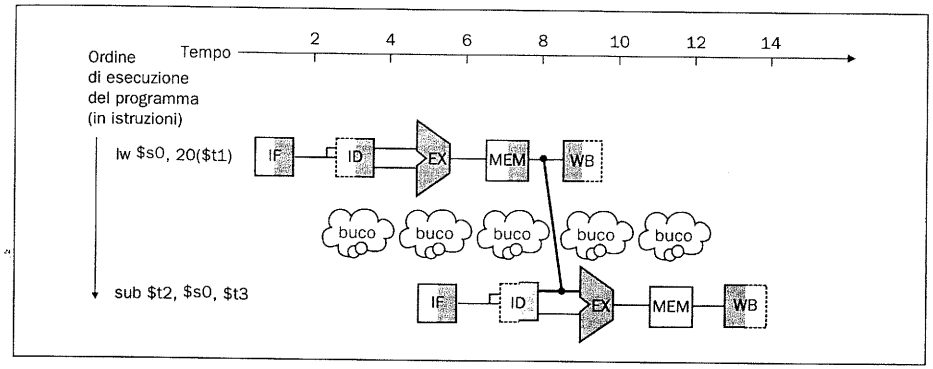


Figura 6.9 Anche se si usa la propagazione, è necessario uno stallo quando un'istruzione di formato-R che segue un'istruzione di load cerca di usare il dato. Senza lo stallo, il cammino dall'uscita dello stadio di accesso alla memoria all'ingresso dello stadio di esecuzione dovrebbe andare indietro nel tempo, il che è impossibile.

```
                    # nel registro $t1 c'è l'indirizzo di v[k]
lw   $t0, 0($t1)    # $t0 (temp) = v[k]
lw   $t2, 4($t1)    # $t2 = v[k+1]
sw   $t2, 0($t1)    # v[k] = $t2
sw   $t0, 4($t1)    # v[k+1] = $t0 (temp)
```

Si riordinino le istruzioni in modo da evitare stalli nella pipeline.

Soluzione

La criticità si presenta sul registro `$t2`, tra la seconda `lw` e la prima `sw`; lo scambio tra le due istruzioni `sw` elimina la criticità:

```
                    # nel registro $t1 c'è l'indirizzo di v[k]
lw   $t0, 0($t1)    # $t0 (temp) = v[k]
lw   $t2, 4($t1)    # $t2 = v[k+1]
sw   $t0, 4($t1)    # v[k+1] = $t0 (temp)
sw   $t2, 0($t1)    # v[k] = $t2
```

Si noti che in questo modo non si crea una nuova criticità in quanto vi è ancora un'istruzione tra la scrittura del registro `$t0` da parte della load e la lettura dello stesso registro da parte della store. In un calcolatore dotato di propagazione la sequenza precedente, riordinata, impiega dunque 4 cicli di clock.

Interfaccia Hardware Software

Come esempio di compromesso tra la complessità dei compilatori e quella dell'hardware, i primi processori MIPS facevano a meno dei circuiti di stallo della pipeline obbligando il software a far seguire ad un'istruzione di load un'istruzione indipendente dal suo risultato. Questa soluzione è anche detta di *caricamento ritardato* (*delayed load*).

Il meccanismo della propagazione fornisce un ulteriore spunto per esaminare l'architettura MIPS, continuando l'analisi iniziata qualche pagina addietro. Ciascuna istruzione MIPS scrive un solo risultato e lo fa sempre al termine della propria esecuzione; la propagazione sarebbe molto più difficile se vi fossero più risultati da propagare per ciascuna istruzione o se questi venissero scritti prima del termine dell'istruzione. Ad esempio le istruzioni di caricamento del PowerPC possono utilizzare la modalità di indirizzamento con aggiornamento (descritta nel Capitolo 3), per cui il processore deve essere in grado di propagare due risultati per ciascuna istruzione di tipo load.

Riepilogo

La tecnica di progetto mediante pipeline sfrutta il parallelismo tra le istruzioni in un flusso sequenziale; a differenza di altre tecniche che aumentano la velocità di esecuzione (si veda il Capitolo 9), essa ha il vantaggio sostanziale di risultare invisibile al programmatore.

I paragrafi successivi di questo capitolo esamineranno il concetto di pipeline utilizzando il sottoinsieme di istruzioni MIPS `lw`, `sw`, `add`, `sub`, `and`, `or`, `slt` e `beq` (lo stesso del Capitolo 5) ed una versione semplificata della pipeline MIPS. Si studieranno i problemi introdotti dall'uso delle pipeline e le tipiche prestazioni ottenibili.

Quadro d'insieme

La presenza di pipeline in un processore fa aumentare il numero di istruzioni contemporaneamente in esecuzione e la frequenza con cui nuove istruzioni vengono iniziate e terminate. Le pipeline non riducono il tempo necessario al completamento di un'istruzione specifica: una pipeline a 5 stadi richiede 5 periodi di clock per completare un istruzione. Usando la terminologia introdotta nel Capitolo 2, le pipeline migliorano il *throughput* delle istruzioni e non il *tempo di esecuzione* di ciascuna di esse.

I set di istruzioni possono semplificare o complicare la vita ai progettisti delle pipeline, i quali devono comunque tenere conto delle criticità strutturali, di quelle sul controllo e di quelle sui dati. La predizione dei salti, la propagazione e gli stalli permettono ai calcolatori di lavorare ad elevata velocità pur producendo risultati corretti.

I lettori meno interessati all'argomento al termine di questo paragrafo hanno già le conoscenze necessarie per poter passare ai Paragrafi 6.8 e 6.9, che trattano concetti più avanzati, come le pipeline dinamiche e superscalari e che descrivono il funzionamento delle architetture basate su pipeline nei microprocessori di recente concezione.

I lettori più metodici al termine di questo paragrafo e con le conoscenze fornite dal Capitolo 5 potranno analizzare le modifiche all'unità di elaborazione dovute all'introduzione delle pipeline, descritte nel Paragrafo 6.2, ed il conseguente aggiornamento dell'unità di controllo, analizzato nel Paragrafo 6.3. Le modifiche introdotte a seguito della propagazione sono oggetto del Paragrafo 6.4, mentre l'integrazione della gestione degli stalli per la risoluzione delle criticità di tipo carica-

e-usa è trattata nel Paragrafo 6.5. Si potrà dunque passare al Paragrafo 6.6 per una soluzione dettagliata ai problemi delle criticità dovute ai salti condizionati ed al Paragrafo 6.7 per il trattamento delle eccezioni.

Approfondimento: Il termine "propagazione" (dall'inglese "forwarding") deriva dall'idea che il risultato viene fatto avanzare da un'istruzione precedente ad una successiva. Il termine "scavalcamento" (dall'inglese "bypassing") indica il passaggio diretto del risultato presente nel register file all'unità che ne necessita.

6.2 L'unità di elaborazione con pipeline

La figura 6.10 riporta l'unità di elaborazione a singolo ciclo progettata nel Capitolo 5. La suddivisione di un'istruzione in cinque fasi implica una pipeline a cinque stadi, il che a sua volta significa che in ciascun periodo di clock vi potranno essere fino a cinque istruzioni in esecuzione. Occorre quindi separare l'unità di elaborazione in cinque parti, ciascuna delle quali prende nome dalla fase corrispondente nell'esecuzione dell'istruzione:

1. IF: Prelievo dell'istruzione (Instruction Fetch).
2. ID: Decodifica dell'istruzione (Instruction Decode) e lettura del register file.
3. EX: Esecuzione o calcolo dell'indirizzo.
4. MEM: Accesso alla memoria dei dati.
5. WB: Scrittura del risultato (Write Back).

Nella figura 6.10 queste cinque fasi corrispondono approssimativamente al modo in cui è disegnata l'unità di elaborazione; durante l'esecuzione istruzioni e dati si spostano generalmente da sinistra a destra, attraverso le cinque fasi. Tornando all'analogia del bucato, i panni diventano puliti, asciutti ed ordinati via via che ci si sposta nella linea di lavoro, e non tornano mai indietro.

Vi sono tuttavia due eccezioni al flusso di istruzioni da sinistra verso destra:

- Lo stadio di scrittura del risultato pone il risultato nel register file (che si trova nel mezzo dell'unità di elaborazione).
- La selezione del valore successivo del PC, scelto tra il valore del PC incrementato e l'indirizzo di salto (proveniente dallo stadio MEM).

Il flusso dei dati da destra a sinistra non ha effetto sull'istruzione corrente: solo le istruzioni successive nella pipeline saranno influenzate da questi movimenti a ritroso dei dati. Si noti che la prima freccia da destra verso sinistra può causare delle criticità sui dati e la seconda delle criticità sul controllo.

Un modo di descrivere ciò che succede nell'esecuzione con pipeline è di ipotizzare che ciascuna istruzione disponga della propria unità di elaborazione, sistemando tali unità lungo l'asse dei tempi per visualizzarne le relazioni. La figura 6.11 mostra l'esecuzione delle istruzioni di figura 6.3 riportando le relative unità di elaborazione su un unico asse dei tempi. Le relazioni di figura 6.11 sono riportate ricorrendo alla versione stilizzata dell'unità di elaborazione di figura 6.10.

La figura 6.11 pare suggerire che tre istruzioni richiedano tre unità di elaborazione. Nel Capitolo 5 si erano aggiunti dei registri per memorizzare dei dati in modo

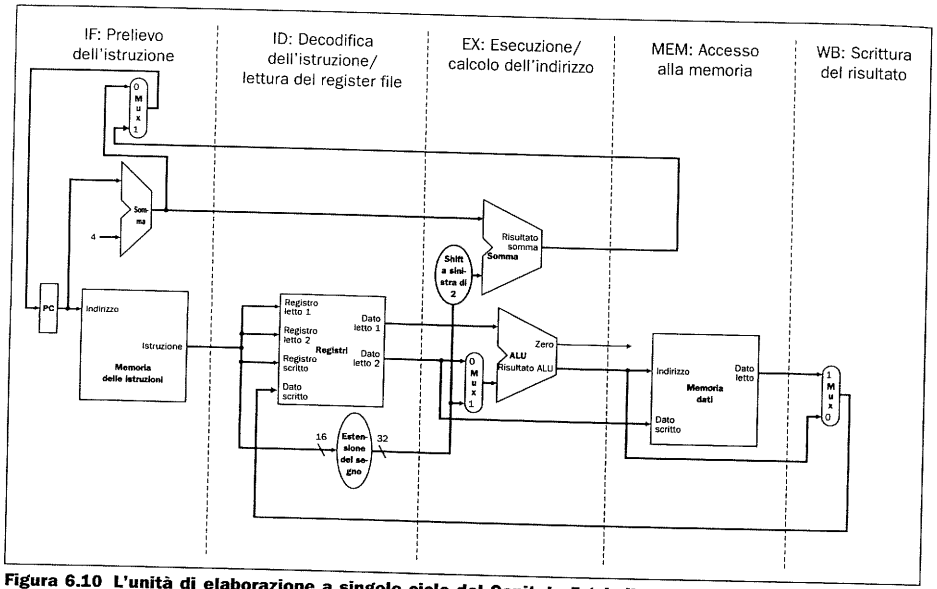


Figura 6.10 L'unità di elaborazione a singolo ciclo del Capitolo 5 (simile alla figura 5.17). Ciascun passo dell'esecuzione dell'istruzione può essere posto in corrispondenza con una zona dell'unità di elaborazione, spostandosi da sinistra verso destra. Le sole eccezioni, riportate in colore, sono costituite dall'aggiornamento del PC e dallo stadio di scrittura dei risultati, che invia verso sinistra il risultato calcolato dalla ALU o il dato proveniente dalla memoria per scriverlo nel register file (solitamente il colore è utilizzato per evidenziare i segnali di controllo, ma in questa figura i segnali evidenziati sono linee di dato).

che alcune parti dell'unità di elaborazione potessero essere riutilizzate; qui si userà la stessa tecnica per riutilizzare le unità di elaborazione multiple. Per esempio, come indicato dalla figura 6.11, la memoria delle istruzioni è utilizzata in una sola delle cinque fasi di esecuzione di un'istruzione, per cui potrà essere riutilizzata da altre istruzioni nelle quattro fasi restanti.

Per memorizzare il valore dell'istruzione letta dalla memoria a beneficio degli altri quattro stadi, occorrerà salvarlo in un registro. Considerazioni analoghe valgono per tutti gli stadi della pipeline, per cui occorrerà aggiungere dei registri ognqualvolta vi siano delle linee di divisione tra gli stadi di figura 6.10 (questa modifica è analoga ai registri aggiunti all'unità di elaborazione nel Capitolo 5 quando si passò da un'implementazione a singolo ciclo ad una multi-ciclo). Ricordando l'analogia della lavanderia, si potrebbe avere un cestino tra ciascuna coppia di stadi per contenere i panni destinati al passo successivo.

La figura 6.12 mostra l'unità di elaborazione con pipeline completa dei *registri di pipeline*, che sono stati evidenziati. In ogni ciclo di clock tutte le istruzioni procedono da un registro di pipeline a quello successivo. Ciascun registro prende il nome dai due stadi che separa; ad esempio il registro di pipeline tra gli stadi IF ed ID è chiamato IF/ID.

Si noti che non vi è alcun registro di pipeline alla fine dello stadio di scrittura del risultato: tutte le istruzioni devono alterare qualche elemento di stato nel processore (il register file, la memoria o il registro PC), per cui un registro di pipeline separato sarebbe ridondante rispetto all'aggiornamento dello stato. Per esempio un'i-

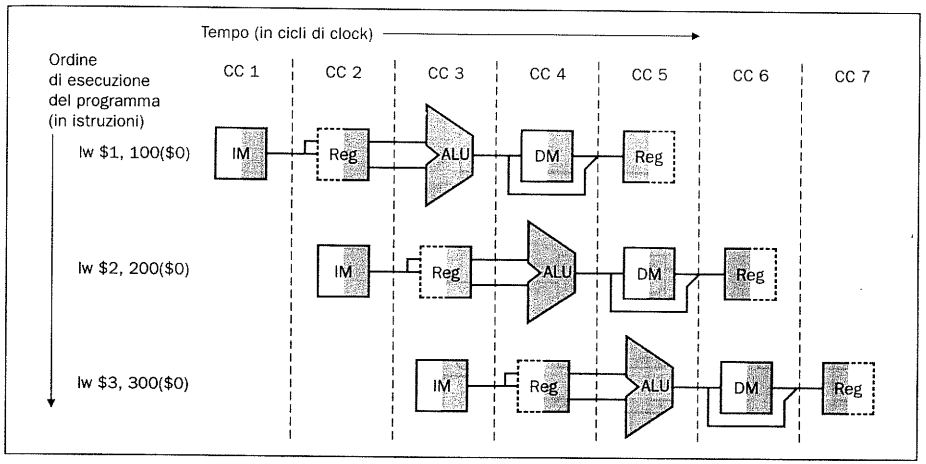


Figura 6.11 Esecuzione delle istruzioni nell'unità di elaborazione a singolo ciclo di figura 6.10 in presenza di pipeline. In modo analogo alle figure dalla 6.7 alla 6.9, in questa figura si finge che ciascuna istruzione disponga della propria unità di elaborazione e si evidenzia la relativa porzione utilizzata. A differenza di tali figure, gli stadi sono etichettati con il nome delle risorse fisiche utilizzate, che corrispondono alle parti dell'unità di elaborazione di figura 6.10. *IM* rappresenta la memoria delle istruzioni, *Reg* il register file e l'unità di estensione del segno nello stadio ID di decodifica dell'istruzione e lettura del register file, e così via. Per garantire un corretto ordinamento temporale, questa unità di elaborazione stilizzata divide il register file in due unità logiche: i registri letti durante la fase di prelievo del registro (ID) ed i registri scritti nella fase finale (WB). Questo uso duplice è rappresentato disegnando nello stadio ID in linea tratteggiata la parte sinistra del register file, non evidenziata, poiché non avviene alcuna operazione di scrittura, mentre nello stadio WB è tratteggiata la parte destra, non evidenziata, in quanto non avvengono operazioni di lettura. Come già in precedenza, si assume che register file venga letto nella prima metà del periodo di clock e scritto nella seconda metà.

struzione di load pone il proprio risultato in uno dei 32 registri ed un'istruzione successiva che richiedesse tale valore potrebbe leggerlo direttamente dai registri. I Paragrafi 6.4 e 6.5 si occuperanno delle criticità sui dati tra le istruzioni presenti nella pipeline; per ora si ignori questo fenomeno.

Per mostrare il funzionamento della pipeline, in questo capitolo si riporteranno un insieme di schemi che rappresentano in modo semplice l'evoluzione nel tempo dei diversi stadi: tutte queste pagine potrebbero dare l'impressione di richiedere molto tempo per essere comprese. Il lettore non tema: le sequenze di figure sono molto più veloci da analizzare di quanto parrebbe, in quanto si possono semplicemente confrontare per vedere che cosa cambia in ciascun ciclo di clock.

Le figure dalla 6.13 alla 6.15, che costituiscono la prima delle sequenze riportate, mostrano in colore le porzioni attive dell'unità di elaborazione durante l'esecuzione di un'istruzione load attraverso i cinque stadi della pipeline. L'istruzione load viene esaminata per prima in quanto è attiva in tutti gli stadi. Come nelle figure comprese tra la 6.7 e la 6.12, si evidenzierà la *metà destra* dei registri o della memoria quando questi vengono *letti*, la *metà sinistra* quando vengono *scritti*. In ciascuna figura verrà riportato il nome dell'istruzione, `lw`, insieme al nome dello stadio attivo; i cinque stadi sono i seguenti:

1. *Prelievo dell'istruzione*: La parte superiore della figura 6.13 mostra l'istruzione mentre viene letta dalla memoria utilizzando PC come indirizzo e viene posta nel registro di pipeline IF/ID (esso è analogo al registro Instruction Register

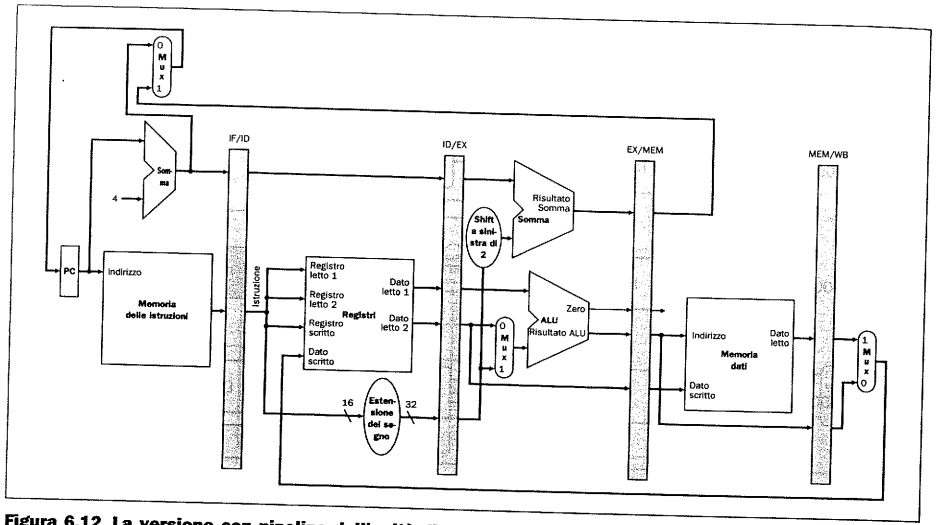


Figura 6.12 La versione con pipeline dell'unità di elaborazione di figura 6.10. I registri di pipeline, evidenziati mediante l'uso del colore, separano gli stadi della pipeline e sono etichettati con il nome degli stadi adiacenti; per esempio, il primo ha l'etichetta *IF/ID* poiché separa lo stadio IF di prelievo dell'istruzione dallo stadio ID di decodifica dell'istruzione. I registri devono avere un'ampiezza sufficiente a memorizzare tutti i dati corrispondenti alle linee che li attraversano. Ad esempio il registro IF/ID deve essere ampio 64 bit poiché deve contenere sia il valore su 32 bit dell'istruzione letta dalla memoria, sia il valore incrementato del registro PC a 32 bit. I registri verranno trattati più in dettaglio nel seguito del capitolo, ma per ora si sappia che gli altri tre registri di pipeline contengono rispettivamente 128, 97 e 64 bit.

della figura 5.30). L'indirizzo contenuto nel registro PC viene incrementato di 4 e riscritto nuovamente in PC in modo da essere pronto per il prossimo ciclo di clock. L'indirizzo incrementato è anche salvato nel registro di pipeline IF/ID perché potrebbe essere richiesto da un'istruzione successiva, ad esempio da `beq`. Il calcolatore non può prevedere quale tipo di istruzione verrà letta dalla memoria, per cui deve essere pronto per ogni istruzione, trasportando lungo la pipeline tutte le informazioni potenzialmente necessarie.

2. *Decodifica dell'istruzione e lettura del register file*: La parte inferiore della figura 6.13 mostra la parte inferiore del registro di pipeline IF/ID che fornisce il campo immediato di 16 bit, convertito a 32 bit tramite l'estensione del segno, ed i numeri dei due registri da leggere. I tre valori sono memorizzati nel registro di pipeline ID/EX insieme al valore incrementato di PC: viene infatti nuovamente trasferito tutto ciò che potrebbe essere necessario ad una qualsiasi istruzione nei cicli di clock successivi.
3. *Esecuzione o calcolo dell'indirizzo*: La figura 6.14 indica che l'istruzione load legge dal registro di pipeline ID/EX il contenuto del registro 1 ed il valore immediato, esteso con segno, e li somma utilizzando la ALU. Il valore della somma è posto nel registro di pipeline EX/MEM.
4. *Accesso alla memoria*: La parte superiore della figura 6.15 illustra l'istruzione di load mentre legge la memoria dei dati utilizzando come indirizzo il valore letto dal registro di pipeline EX/MEM. Il dato letto viene caricato nel registro di pipeline MEM/WB.

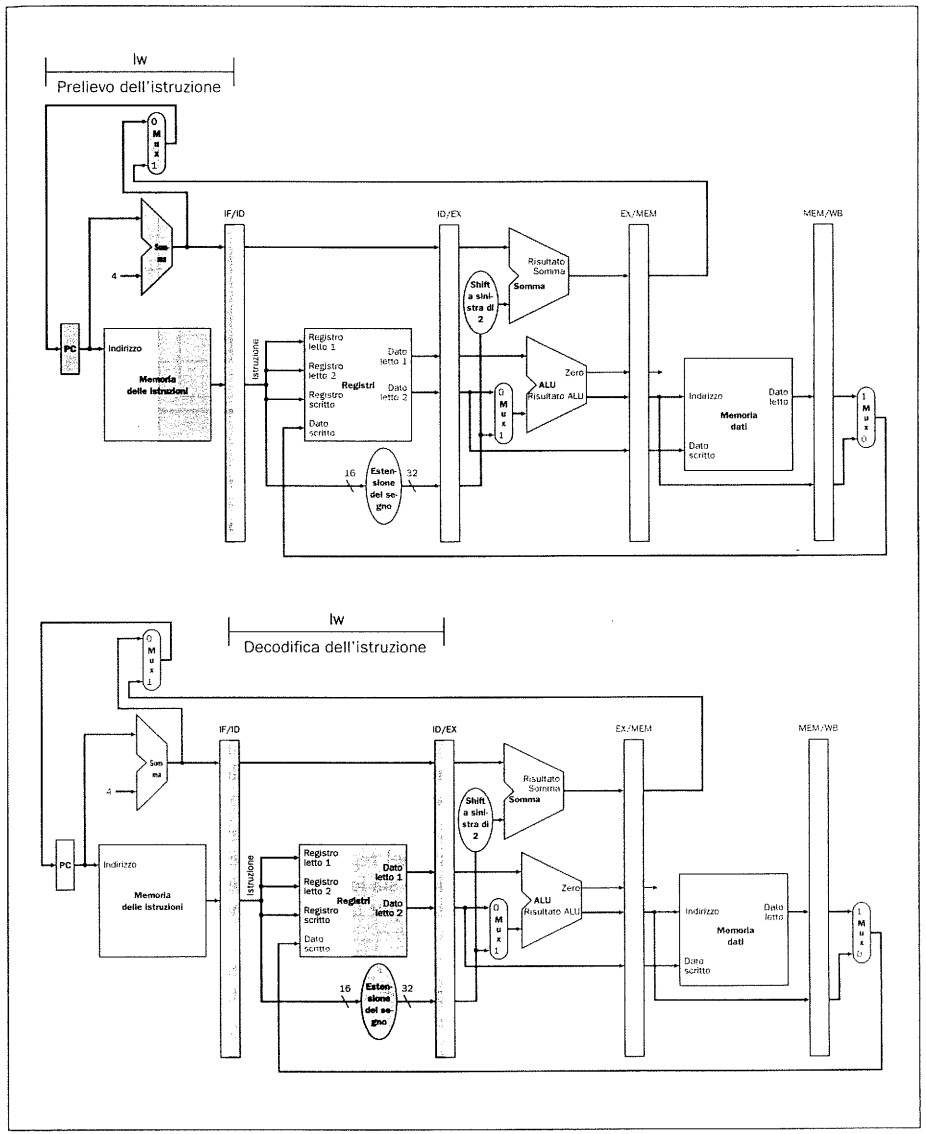


Figura 6.13 IF e ID: il primo ed il secondo stadio di pipeline di un'istruzione, con le parti attive dell'unità di elaborazione della figura 6.12 evidenziate. La convenzione per le parti evidenziate è la stessa utilizzata nella figura 6.7. Come nel Capitolo 5 non vi è confusione tra la lettura e la scrittura dei registri in quanto i contenuti variano solamente in corrispondenza del fronte del clock. Sebbene la load nello stadio 2 richieda solo il primo registro, il processore non conosce il tipo dell'istruzione in fase di decodifica, per cui estende la costante a 16 bit e legge entrambi i registri scrivendoli nel registro di pipeline ID/EX. In questo caso non servono tre operandi, ma la realizzazione dell'unità di controllo è semplificata se tutti gli operandi sono comunque memorizzati.

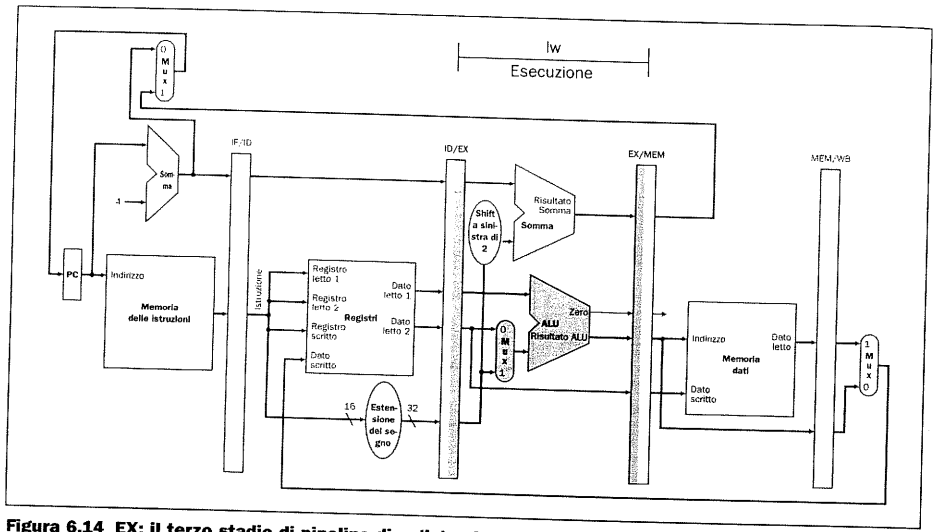


Figura 6.14 EX: il terzo stadio di pipeline di un'istruzione load, con evidenziate le parti dell'unità di elaborazione della figura 6.12 usate in questo stadio. Il registro viene sommato al valore immediato esteso con segno e la somma viene posta nel registro di pipeline EX/MEM.

5. *Scrittura del risultato*: La parte inferiore della figura 6.15 riporta il passo finale: lettura del dato dal registro di pipeline MEM/WB e scrittura dello stesso nel register file al centro della figura.

Questa carrellata sull'istruzione load indica che tutte le informazioni necessarie in uno stadio di pipeline successivo devono essere passate a tale stadio attraverso i registri di pipeline. Un'analoga carrellata su un'istruzione di store evidenzia le analogie tra le esecuzioni di diverse istruzioni ed il passaggio delle informazioni agli stadi successivi. Ecco i cinque stadi di pipeline per un'istruzione store:

1. *Prelievo dell'istruzione*: L'istruzione viene letta dalla memoria utilizzando il valore di PC come indirizzo e posta nel registro di pipeline IF/ID. Questo stadio è eseguito prima che l'istruzione sia identificata, per cui la parte superiore della figura 6.13 è valida non solo per la load, ma anche per l'istruzione store.
2. *Decodifica dell'istruzione e lettura del register file*: L'istruzione presente nel registro di pipeline IF/ID fornisce i numeri dei due registri da leggere ed il campo immediato di 16 bit da estendere a 32 con segno. Questi tre valori a 32 bit sono memorizzati nel registro di pipeline ID/EX. La parte inferiore della figura 6.13 relativa alle istruzioni load descrive anche le operazioni del secondo stadio nel caso delle store. I primi due stadi sono eseguiti nello stesso modo da tutte le istruzioni, poiché è troppo presto per conoscere il tipo dell'istruzione.
3. *Esecuzione e calcolo dell'indirizzo*: La figura 6.16 mostra il terzo passo, in cui l'indirizzo dell'operando viene posto nel registro di pipeline EX/MEM.
4. *Accesso alla memoria*: La parte superiore della figura 6.17 illustra la scrittura del dato in memoria. Si noti che il registro contenente il dato da scrivere era

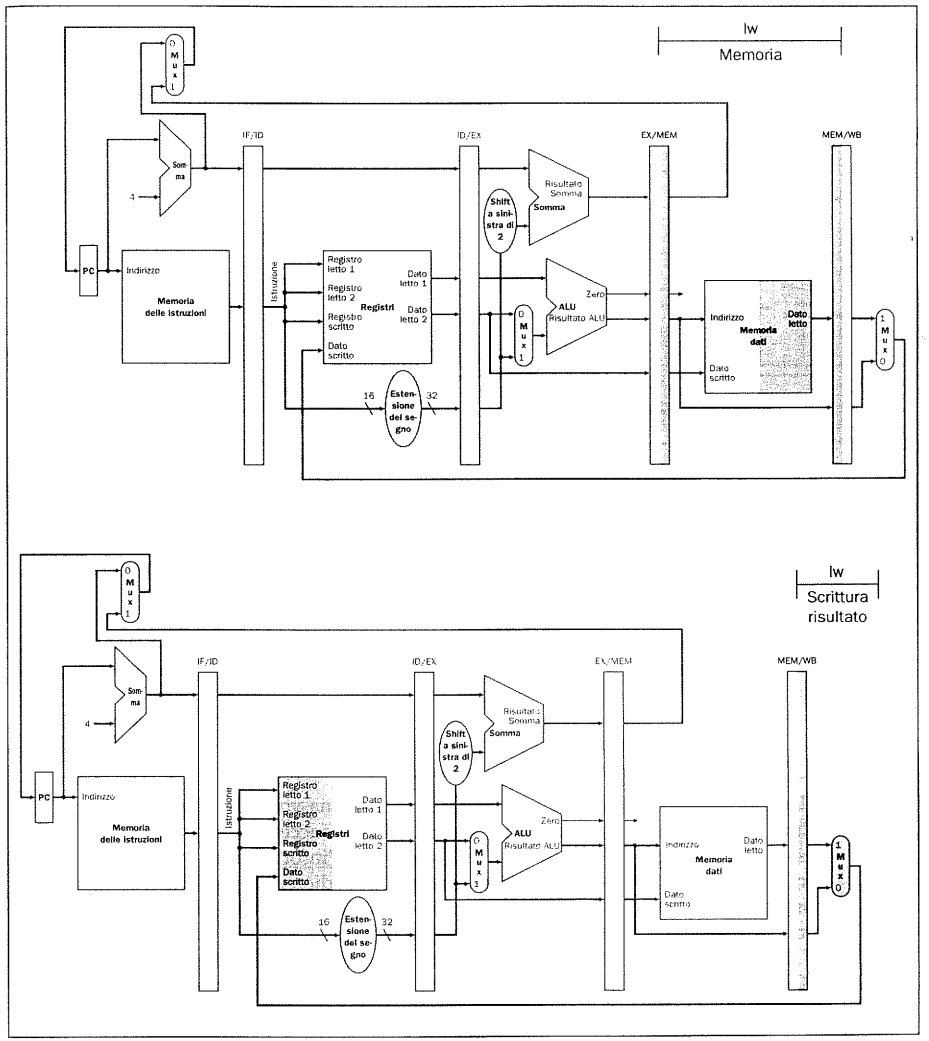


Figura 6.15 MEM e WB: il quarto ed il quinto stadio di pipeline di un'istruzione load, con evidenziate le parti dell'unità di elaborazione della figura 6.12 usate in questi stadi. La memoria dei dati viene letta all'indirizzo specificato dal registro di pipeline EX/MEM ed il dato viene posto nel registro di pipeline MEM/WB. In seguito il dato viene letto dal registro di pipeline MEM/WB e scritto nel register file, al centro della figura.

stato letto in uno stadio precedente ed il relativo valore memorizzato nel registro ID/EX. L'unico modo per rendere il dato disponibile durante lo stadio MEM è quello di memorizzarlo nel registro di pipeline EX/MEM durante lo stadio EX, esattamente come l'indirizzo era stato memorizzato in EX/MEM.

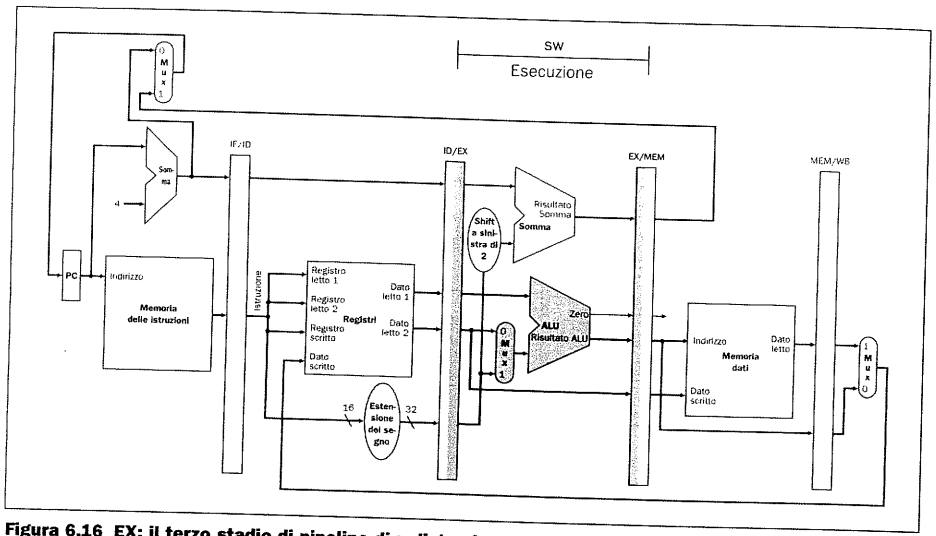


Figura 6.16 EX: il terzo stadio di pipeline di un'istruzione store. A differenza del terzo stadio dell'istruzione load di figura 6.14, il secondo registro viene caricato nel registro di pipeline EX/MEM per essere usato nello stadio seguente. Sebbene non sia sbagliato scrivere sempre il secondo registro nel registro di pipeline EX/MEM, si è scelto di scriverlo solamente se l'istruzione è una store per facilitare la comprensione della pipeline.

5. *Scrittura del risultato*: La parte inferiore della figura 6.17 riporta il passo finale dell'istruzione store: in questo caso non succede nulla. Poiché tutte le istruzioni che seguono la store sono ancora in fase di completamento, non vi è alcun modo per accelerarle: di conseguenza un'istruzione deve passare attraverso uno stadio anche se non vi è nulla da fare poiché le istruzioni successive stanno già procedendo alla massima velocità possibile.

L'esempio dell'istruzione store sottolinea ancora una volta come, per passare un valore da uno stadio iniziale della pipeline ad uno successivo, esso debba essere posto in un registro di pipeline; in caso contrario l'informazione verrebbe persa non appena l'istruzione successiva raggiunge lo stadio della pipeline. Per l'istruzione store si è dovuto passare il valore di uno dei due registri letti nello stadio ID fino allo stadio MEM, dove è stato scritto in memoria. Il valore è stato prima posto nel registro di pipeline ID/EX e poi copiato nel successivo EX/MEM.

Gli esempi relativi alle istruzioni load e store permettono di introdurre una seconda considerazione: ciascun componente logico dell'unità di elaborazione (come la memoria delle istruzioni, le porte di lettura dei registri, la ALU, la memoria dei dati e le porte di scrittura dei registri) può essere usato in *un solo* stadio della pipeline; in caso contrario, si avrebbe una *criticità strutturale* (si veda il Paragrafo 6.1). Di conseguenza questi componenti, e la relativa logica di controllo, possono essere associati ad un solo stadio di pipeline.

A questo punto si può svelare un errore nel progetto dell'istruzione load, che alcuni lettori avranno già notato. Quale registro viene modificato nello stadio finale della load? In particolare, quale istruzione fornisce il numero del registro da scrivere? Il numero del registro da scrivere è preso dall'istruzione presente nel registro di pipeline IF/ID, ma tale istruzione è decisamente *successiva a* quella di load!

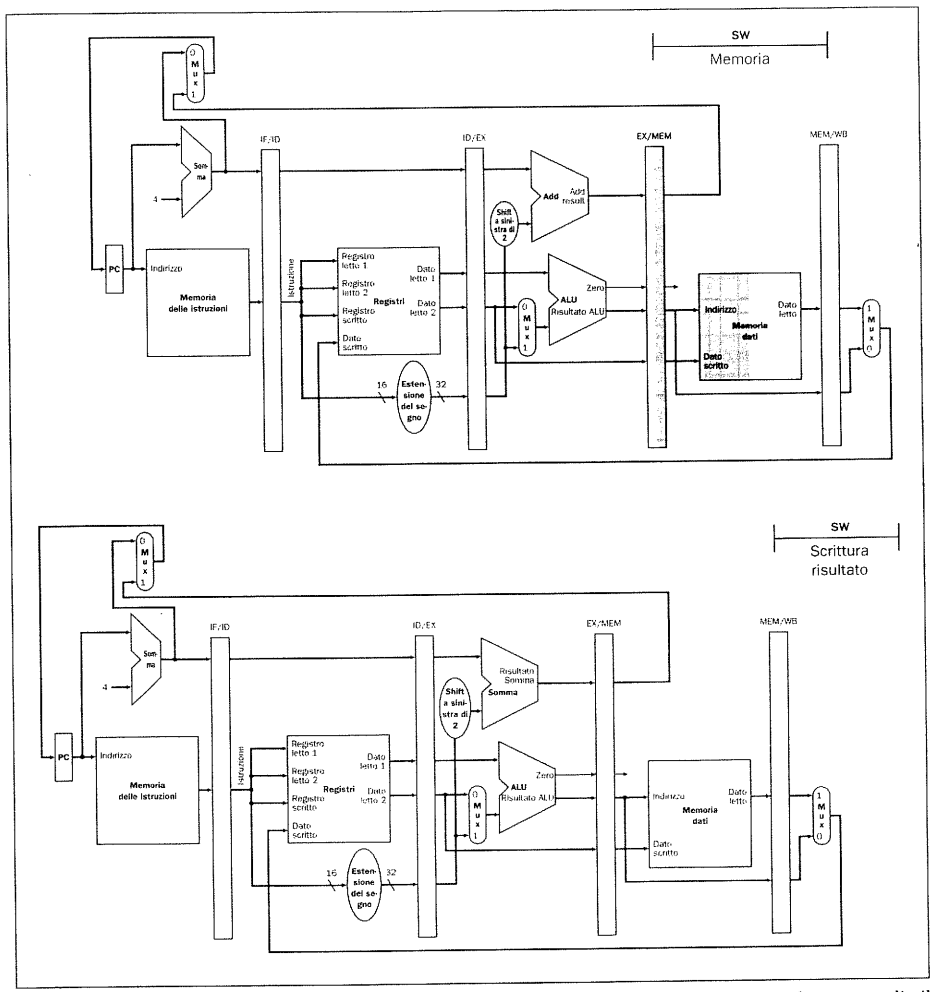


Figura 6.17 MEM e WB: il quarto ed il quinto stadio di pipeline di un'istruzione store. Nel quarto stadio il dato viene scritto nella memoria dati; si noti che esso proviene dal registro di pipeline EX/MEM e che nulla viene modificato nel registro di pipeline MEM/WB. Non appena il dato è scritto in memoria non rimane più nulla da fare per l'istruzione store, per cui nello stadio 5 non accade niente.

In conclusione, è necessario preservare il numero del registro di destinazione per l'istruzione load: così come la store ha inviato il *contenuto* del registro dal registro di pipeline ID/EX al registro EX/MEM per utilizzarlo nello stadio MEM, la load deve inviare il *numero* del registro da ID/EX attraverso EX/MEM fino a MEM/WB per poterlo utilizzare nello stadio WB. Un modo alternativo di vedere il passaggio del numero del registro è di pensare che, per poter condividere l'unità di elaborazione con pipeline, si è dovuta preservare l'istruzione letta durante lo stadio IF, in

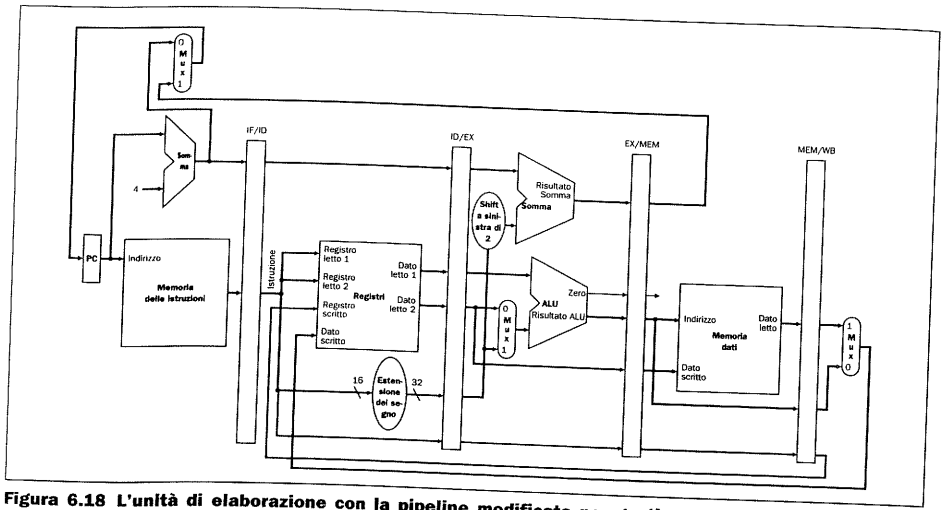


Figura 6.18 L'unità di elaborazione con la pipeline modificata per gestire correttamente l'istruzione load. Il numero del registro da scrivere ora proviene dal registro di pipeline MEM/WB insieme al dato. Il numero del registro è stato passato dallo stadio ID fino a raggiungere il registro di pipeline MEM/WB, aggiungendo 5 bit agli ultimi 3 registri di pipeline. Il nuovo collegamento è riportato in colore.

modo che ciascun registro di pipeline contenga la parte di istruzione usata da quello stadio e da quelli successivi.

La figura 6.18 mostra la versione corretta dell'unità di elaborazione, che passa il numero di registro prima al registro di pipeline ID/EX, poi a EX/MEM ed infine a MEM/WB. Il numero del registro è usato nello stadio WB per specificare il registro da scrivere. La figura 6.19 riporta uno schema unico dell'unità di elaborazione corretta ed evidenzia i componenti utilizzati nei cinque stadi dell'istruzione load word

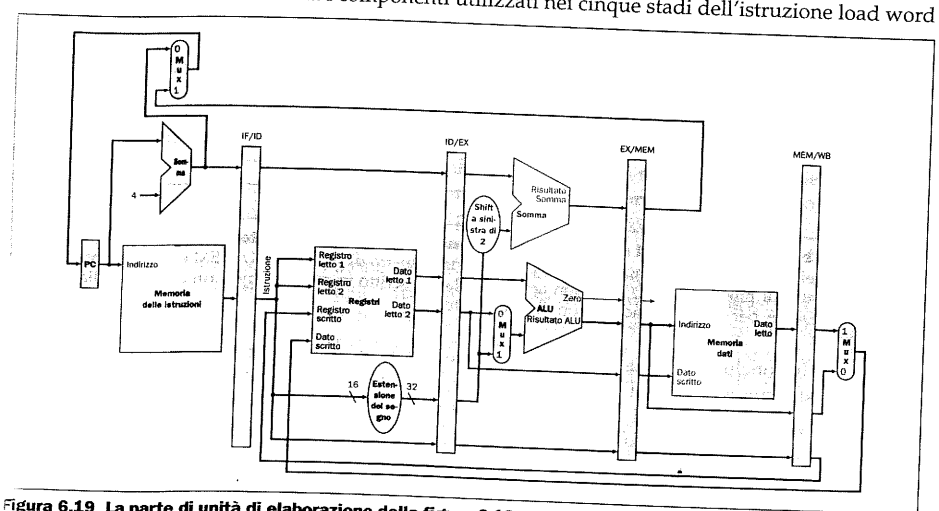


Figura 6.19 La parte di unità di elaborazione della figura 6.18 utilizzata nei cinque stadi di un'istruzione load.

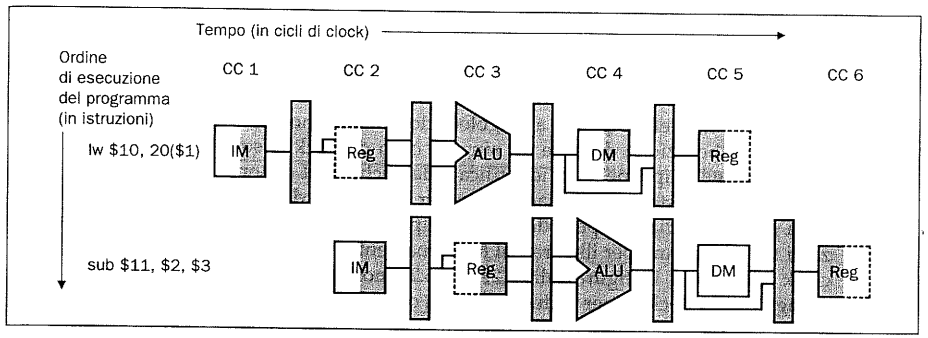


Figura 6.20 Diagramma della pipeline a più cicli di clock per due istruzioni. Questo stile di rappresentazione delle pipeline mostra in una sola figura l'esecuzione completa delle istruzioni. Le istruzioni sono riportate dall'alto al basso in ordine di esecuzione ed i cicli di clock procedono da sinistra verso destra. A differenza della figura 6.7, qui i registri di pipeline sono indicati tra gli stadi. La figura 6.21 mostra il modo tradizionale di disegnare questo diagramma.

nelle figure dalla 6.13 alla 6.15. Si veda il Paragrafo 6.6 per la spiegazione su come far sì che anche l'istruzione di salto condizionato venga eseguita correttamente.

Rappresentazione grafica delle pipeline

Le pipeline possono essere complesse da comprendere, perché in ciascun ciclo di clock vi sono molte istruzioni contemporaneamente in esecuzione nella stessa unità di elaborazione. Per facilitare la comprensione vi sono due modi fondamentali per realizzare le figure che rappresentano le pipeline: i *diagrammi delle pipeline a più cicli di clock*, come quello di figura 6.11 ed i *diagrammi delle pipeline a singolo ciclo di clock*, come quelli delle figure da 6.13 a 6.17. Per meglio chiarire la questione, nel seguito verrà rappresentata una stessa sequenza di istruzioni secondo entrambi gli stili. La sequenza è la seguente:

```
lw   $10, 20($1)
sub  $11, $2, $3
```

La figura 6.20 mostra il diagramma della pipeline a più colpi di clock per tali istruzioni: in questo tipo di diagrammi il tempo scorre da sinistra verso destra, mentre le istruzioni procedono dall'alto verso il basso, come nella pipeline del bucato di figura 6.1. Lungo l'asse delle istruzioni sono rappresentati gli stadi della pipeline, posizionati nei periodi di clock opportuni: i cinque stadi sono rappresentati da una versione stilizzata delle unità funzionali, ma un semplice rettangolo con il nome dello stadio sarebbe altrettanto valido. La figura 6.21 mostra la versione

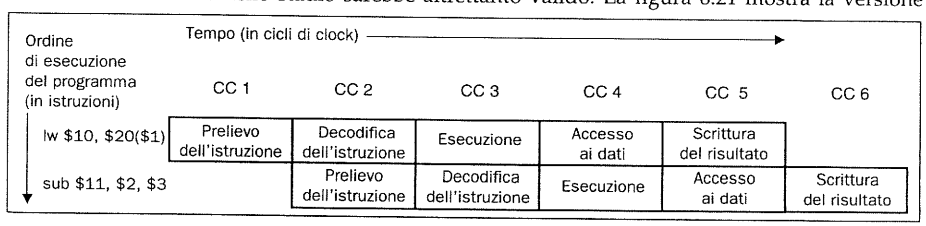


Figura 6.21 Diagramma della pipeline a più cicli di clock in forma tradizionale per le stesse due istruzioni della figura 6.20.

più tradizionale del diagramma della pipeline a più cicli di clock; si noti che la figura 6.20 mostra le risorse fisiche utilizzate in ciascuno stadio, mentre la figura 6.21 riporta il *nome* di ciascuno stadio. Verranno utilizzati dei diagrammi a più cicli di clock per fornire delle viste d'insieme di situazioni che coinvolgono la pipeline.

I diagrammi delle pipeline a singolo ciclo di clock mostrano lo stato dell'intera unità di elaborazione durante un singolo ciclo di clock e, solitamente, le cinque istruzioni nella pipeline sono identificate da etichette al di sopra dei relativi stadi. Si utilizzerà questo tipo di diagramma per mostrare i dettagli di ciò che avviene nella pipeline in ciascun periodo di clock; tipicamente si riporteranno delle sequenze di tali diagrammi per illustrare il funzionamento della pipeline su più periodi di clock consecutivi. Le figure dalla 6.22 alla 6.24 mostrano i diagrammi delle pipeline a singolo ciclo di clock per le due istruzioni considerate.

Queste due rappresentazioni delle pipeline sono, ovviamente, equivalenti. Un aspetto che potenzialmente è fonte di confusione è l'ordine delle istruzioni nei due diagrammi: l'istruzione più recente è *in basso e a destra* dei diagrammi a più cicli di clock, mentre è *a sinistra* nei diagrammi ad un ciclo di clock.

Estrarre una porzione verticale di ampiezza pari ad un periodo di clock da un diagramma a più cicli di clock mostra lo stato della pipeline come un diagramma a singolo ciclo di clock. La conversione di una sequenza di diagrammi a singolo ciclo di clock in un diagramma a più cicli di clock è più complessa: poiché l'istruzione più recente si trova in basso, si ruota ciascun diagramma a singolo ciclo di clock di 90 gradi in senso antiorario; i diagrammi così ruotati sono poi affiancati e separati da un periodo di clock, in modo che le unità di elaborazione dei cinque stadi di ciascuna istruzione siano allineate su una singola riga orizzontale (vedere l'Esercizio 6.8).

Approfondimento: Poiché il contatore di programma trasferisce informazioni tra due istruzioni, e non all'interno di una stessa istruzione, i diagrammi come quello di figura 6.22 riportano il registro PC come un registro esplicito. Lo si potrebbe anche considerare come un registro di pipeline prima dello stadio di reperimento dell'istruzione o, equivalentemente, tra lo stadio di scrittura del risultato di un'istruzione e lo stadio di prelievo di quella successiva. In questo caso il registro PC sarebbe disegnato come un rettangolo di forma allungata, come gli altri registri di pipeline.

6.3 L'unità di controllo con pipeline

Nel calcolatore 6600, forse più che in ogni altro calcolatore, era l'unità di controllo a fare la differenza.

James Thornton, *Design of a Computer: The Control Data 6600*, 1970

Così come nel Paragrafo 5.2 si era aggiunta l'unità di controllo all'unità di elaborazione elementare, ora si può aggiungere l'unità di controllo all'unità di elaborazione con pipeline. Si partirà con un progetto semplificato che ignora la reale complessità del problema; nei Paragrafi che vanno dal 6.4 al 6.8 si approfondirà l'analisi prendendo in considerazione le difficoltà esistenti nel progetto di un sistema reale.

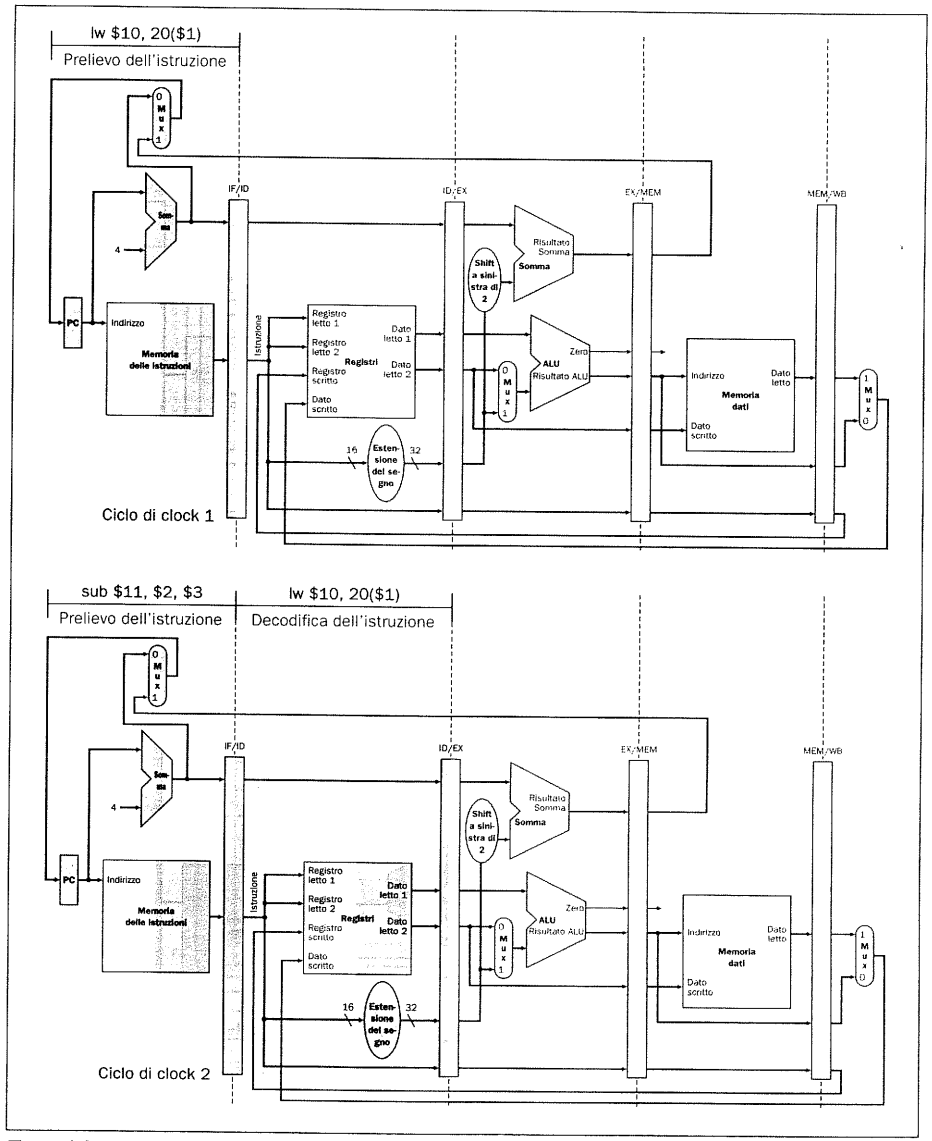


Figura 6.22 Diagramma della pipeline a singolo ciclo di clock per i cicli di clock 1 (in alto) e 2 (in basso). Questo stile di rappresentazione della pipeline è un'istantanea di tutte le istruzioni che sono in esecuzione nel ciclo di clock 1. L'esempio considerato ha solo due istruzioni, per cui saranno identificati al più due stadi per ciclo di clock; in condizioni normali tutti i cinque stadi sarebbero occupati. Le parti evidenziate dell'unità di elaborazione corrispondono a quelle attive nel ciclo di clock. L'istruzione load viene reperita durante il ciclo di clock 1 e decodificata nel ciclo di clock 2, mentre la sottrazione viene reperita nel secondo ciclo. Per facilitare la comprensione delle figure, gli stadi successivi della pipeline sono rappresentati vuoti, ma in condizioni normali vi sarebbe un'istruzione in ciascuno di essi.

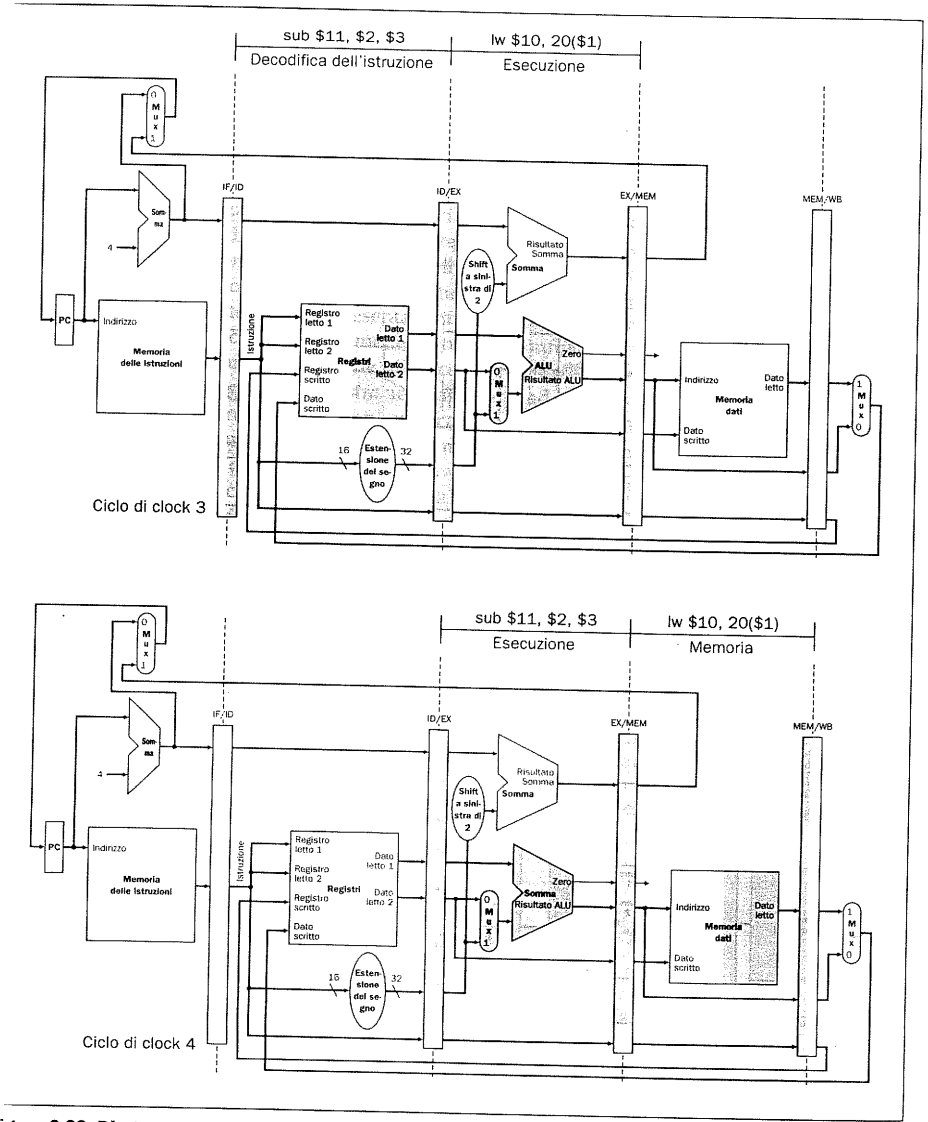


Figura 6.23 Diagramma della pipeline a singolo ciclo di clock per i cicli di clock 3 (in alto) e 4 (in basso). Nel terzo periodo di clock, nella parte superiore, `lw` entra nello stadio EX; contemporaneamente `sub` entra in ID. Nel quarto ciclo, in basso, `lw` si sposta nello stadio MEM, leggendo la memoria all'indirizzo specificato da EX/MEM all'inizio del ciclo di clock 4. Nello stesso tempo la ALU calcola la sottrazione e pone la differenza in EX/MEM al termine del periodo di clock.

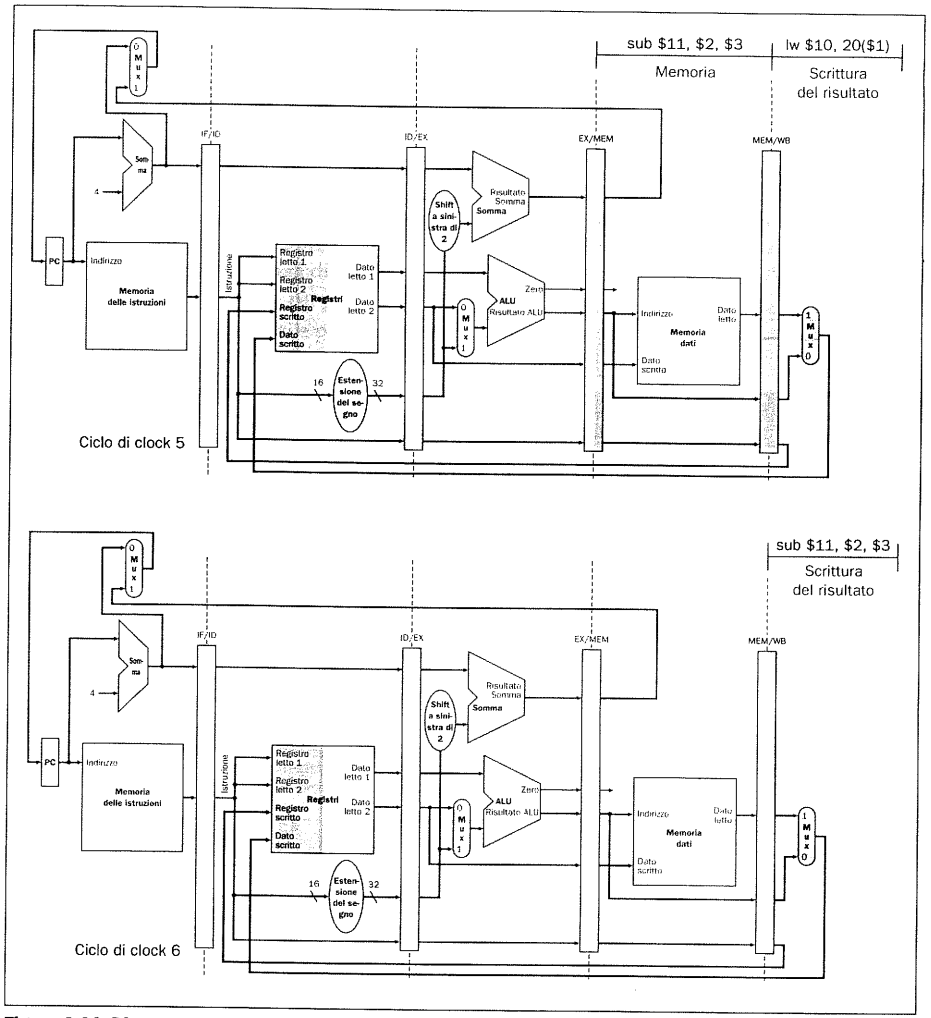


Figura 6.24 Diagramma della pipeline a singolo ciclo di clock per i cicli di clock 5 (in alto) e 6 (in basso). Nel ciclo di clock 5 `lw` viene completata scrivendo il dato presente in MEM/WB nel registro 10 e `sub` invia la differenza presente in EX/MEM verso MEM/WB. Al ciclo di clock successivo `sub` scrive il valore di MEM/WB nel registro 11.

Il primo passo consiste nell'etichettare i segnali di controllo dell'unità di elaborazione esistente, come riportato in figura 6.25. Si cercherà di riutilizzare il più possibile il progetto dell'unità di controllo per l'unità di elaborazione elementare della figura 5.17; in particolare si utilizzeranno la stessa unità di controllo della ALU, la stessa logica per i salti, gli stessi multiplexer per la selezione del numero del registro destinazione e le stesse linee di controllo. Tali funzionalità sono descritte nelle

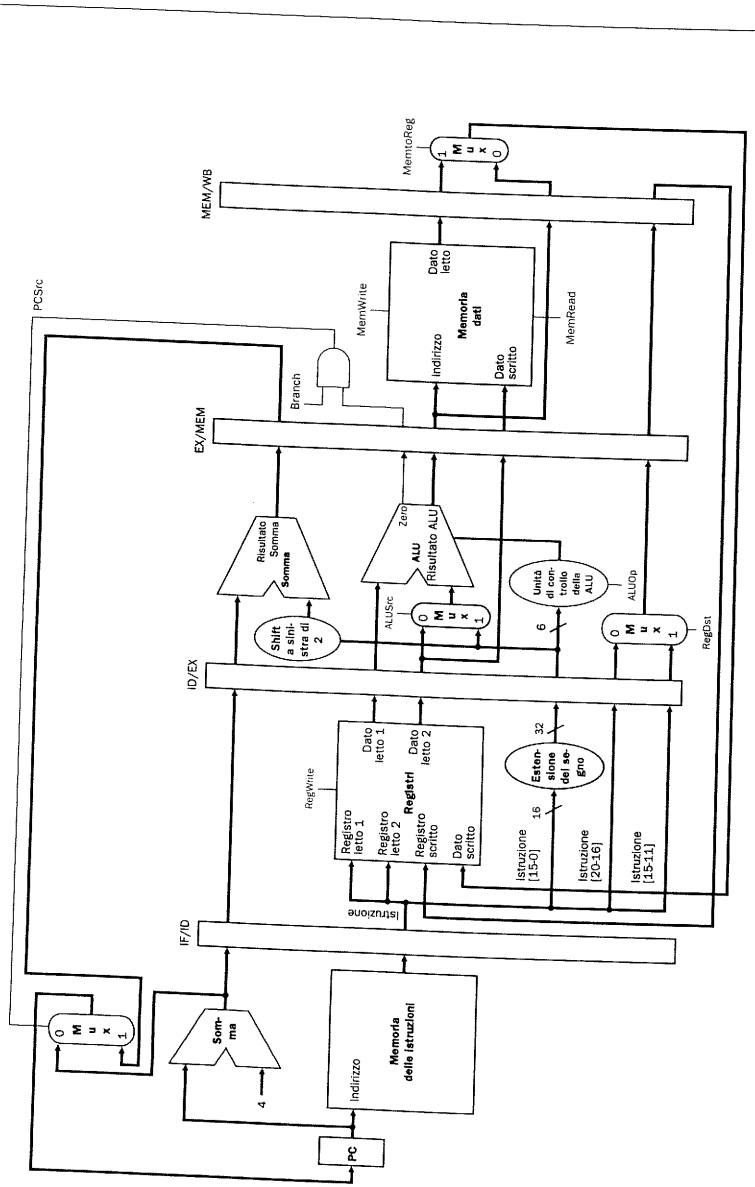


Figura 6.25 L'unità di elaborazione con pipeline della figura 6.18 completa dei segnali di controllo. Questa unità di elaborazione ricalca dal Capitolo 5 la logica di controllo per il caricamento di PC, la selezione del numero del registro destinazione e l'unità di controllo della ALU. Si noti che nello stadio EX è necessario avere come ingresso dell'unità di controllo della ALU il campo dell'istruzione funct (codice di funzione) di 6 bit, quindi tali bit devono essere inclusi nel registro di pipeline ID/EX. Si ricordi che tali bit sono anche i bit meno significativi del campo immediato dell'istruzione: il registro di pipeline ID/EX li può quindi fornire a partire dal campo immediato, in quanto vengono lasciati invariati dall'operazione di estensione del segno.

figure 5.14, 5.18 e 5.20 (per facilitare la comprensione del testo che segue, le informazioni chiave sono riprodotte nelle figure dalla 6.26 alla 6.28).

Come già per l'implementazione a singolo ciclo discussa nel Capitolo 5 si assumerà che il registro PC venga scritto ad ogni ciclo di clock, per cui non vi sarà alcun segnale di controllo per la scrittura di tale registro. Per lo stesso motivo non vi saranno segnali di scrittura espliciti per i registri di pipeline (IF/ID, ID/EX, EX/MEM e MEM/WB), essendo questi scritti ad ogni ciclo di clock.

Per definire l'unità di controllo con pipeline è sufficiente calcolare i valori dei segnali di controllo in ciascuno stadio. Poiché ciascuna linea di controllo è associata ad un componente e questo è attivo in un solo stadio della pipeline, si possono dividere i segnali di controllo in cinque gruppi secondo il relativo stadio:

1. *Prelievo dell'istruzione*: I segnali di controllo per leggere la memoria delle istruzioni e per scrivere il PC sono sempre affermati, quindi non vi è nulla di particolare da controllare in questo stadio.

Codice operativo	ALUOp	Operazione svolta	Codice funzione	Azione della ALU	Ingresso di controllo della ALU
LW	00	load word	XXXXXX	somma	010
SW	00	store word	XXXXXX	somma	010
Branch equal	01	branch equal	XXXXXX	sottrazione	110
Tipo-R	10	somma	100000	somma	010
Tipo-R	10	sottrazione	100010	sottrazione	110
Tipo-R	10	AND	100100	and	000
Tipo-R	10	OR	100101	or	001
Tipo-R	10	set on less than	101010	set on less than	111

Figura 6.26 Copia della figura 5.14. La figura mostra come sono calcolati i bit di controllo della ALU a partire dai bit di controllo ALUOp e dai diversi codici di funzione per le istruzioni di tipo-R.

Nome del segnale	Effetto quando non affermato (0)	Effetto quando affermato (1)
RegDst	Il numero del registro destinazione per Registro scritto proviene dal campo rt (bit 20-16).	Il numero del registro destinazione per Registro scritto proviene dal campo rd (bit 15-11).
RegWrite	Nessuno.	Nel registro specificato dall'ingresso Registro scritto è scritto il valore presente sull'ingresso Dato scritto.
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2).	Il secondo operando della ALU è la versione estesa dei 16 bit meno significativi dell'istruzione.
PCSrc	Il valore di PC viene sostituito dall'uscita del sommatore che calcola il valore di PC+4.	Il valore di PC viene sostituito dall'uscita del sommatore che calcola la destinazione del salto.
MemRead	Nessuno.	Il contenuto della cella di memoria dati determinato dall'ingresso Indirizzo è posto sull'uscita Dato letto.
MemWrite	Nessuno.	Il contenuto della cella di memoria dati determinato dall'ingresso Indirizzo è sostituito dal valore presente sull'ingresso Dato scritto.
MemtoReg	Il valore inviato all'ingresso Dato scritto dei registri proviene dalla ALU.	Il valore inviato all'ingresso Dato scritto dei registri proviene dalla memoria dati.

Figura 6.27 Copia della figura 5.18. Viene definita la funzione di ciascuno dei sette segnali di controllo. I segnali di controllo della ALU (ALUOp) sono definiti nella seconda colonna della figura 6.26. Quando il segnale di controllo di 1 bit di un multiplexer a 2 vie viene affermato, il multiplexer seleziona l'ingresso etichettato con 1; in caso contrario, cioè se il segnale di controllo è non affermato, il multiplexer seleziona l'ingresso 0. Si noti che nella figura 6.25 PCSrc è controllato da una porta AND: se il segnale Branch ed il segnale Zero della ALU sono entrambi affermati, allora PCSrc vale 1, altrimenti 0. L'unità di controllo porterà Branch ad un valore affermato solamente durante le operazioni beq: in tutti gli altri casi PCSrc è sempre posto a 0.

2. *Decodifica dell'istruzione/lettura del register file*: Come nello stadio precedente, ad ogni ciclo di clock avviene sempre la stessa cosa, quindi non vi sono segnali di controllo da calcolare.
3. *Esecuzione/calcolo dell'indirizzo*: I segnali da calcolare sono RegDst, ALUOp e ALUSrc (si vedano le figure 6.26 e 6.27); essi selezionano il registro risultato, l'operazione della ALU e scelgono il dato da inviare alla ALU tra Dato letto 2 e l'immediato esteso con segno.
4. *Accesso alla memoria*: I segnali di controllo per questo stadio sono Branch, MemRead e MemWrite; essi devono essere affermati rispettivamente nelle istruzioni branch equal, load e store. Si ricordi che PCSrc nella figura 6.27 seleziona l'indirizzo successivo in ordine sequenziale, a meno che si affermi il segnale Branch ed il risultato della ALU valga zero.
5. *Scrittura del risultato*: I due segnali di controllo sono MemtoReg, che sceglie tra inviare al register file l'uscita della ALU oppure il valore proveniente dalla memoria, e RegWrite, che scrive il valore selezionato.

Siccome l'introduzione della pipeline nell'unità di elaborazione lascia invariato il significato dei segnali di controllo, si possono usare gli stessi valori calcolati in precedenza. La figura 6.28 riporta gli stessi valori del Capitolo 5, ma ora i nove segnali di controllo sono raggruppati secondo lo stadio della pipeline.

Implementare l'unità di controllo significa forzare i nove segnali di controllo ai valori corretti in ciascuno stadio e per ciascuna istruzione. Il modo più semplice di far ciò è di estendere i registri di pipeline in modo da includere le informazioni di controllo.

Poiché i segnali di controllo nascono nello stadio EX, si può generare l'informazione di controllo durante la decodifica dell'istruzione. La figura 6.29 mostra che i segnali di controllo sono usati nello stadio appropriato della pipeline, via via che l'istruzione scorre lungo la pipeline stessa, in modo analogo a come il numero del registro destinazione si sposta lungo la pipeline nella figura 6.18. La figura 6.30 riporta l'unità di elaborazione completa dei registri di pipeline estesi e dei segnali di controllo connessi allo stadio opportuno.

	Segnali di controllo dello stadio di esecuzione/calcolo dell'indirizzo				Segnali di controllo dello stadio di accesso alla memoria			Segnali di controllo dello stadio di scrittura del risultato	
Istruzione	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
formato-R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Figura 6.28 I valori dei segnali di controllo sono gli stessi della figura 5.20, ma sono stati riordinati in tre gruppi corrispondenti agli ultimi tre stadi della pipeline.

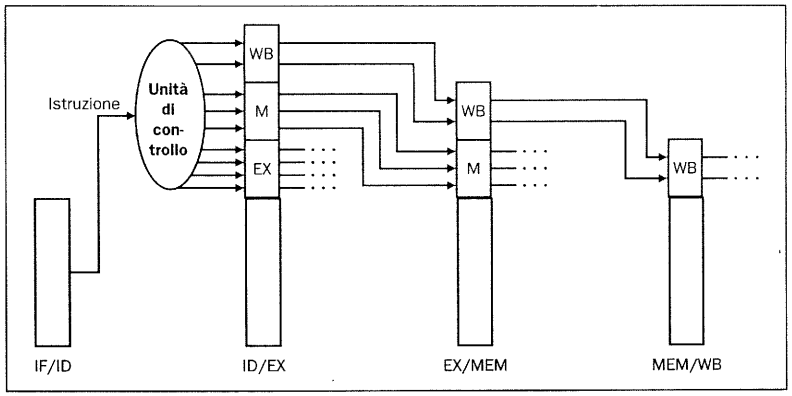


Figura 6.29 I segnali di controllo per gli ultimi tre stadi. Si noti che quattro delle nove linee di controllo sono usate nella fase EX, mentre le rimanenti cinque procedono attraverso il registro di pipeline EX/MEM, esteso in modo da memorizzare i segnali di controllo. Di questi, tre segnali sono usati nello stadio MEM, mentre i restanti due sono inviati a MEM/WB in modo da essere utilizzati nello stadio WB.

Esempio

Dettagli sull'esecuzione in presenza di pipeline e sull'unità di controllo

Si mostri l'esecuzione delle seguenti cinque istruzioni nel loro cammino attraverso la pipeline:

```
lw    $10, 20($1)
sub   $11, $2, $3
and   $12, $4, $5
or    $13, $6, $7
add   $14, $8, $9
```

Si etichettino le istruzioni nella pipeline che precedono la `lw` come precedente<1>, precedente<2>, e così via, e le istruzioni che seguono la `add` come seguente<1>, seguente<2> eccetera.

Soluzione

Le figure dalla 6.31 alla 6.35 mostrano il progresso delle istruzioni attraverso i nove periodi di clock necessari a completarne l'esecuzione, evidenziando gli elementi attivi in ciascuno stadio ed identificando l'istruzione associata allo stadio nei diversi cicli di clock.

Un esame approfondito di tali figure può aiutare la comprensione del funzionamento delle pipeline; alcuni aspetti in particolare sono degni di nota:

- Nella parte bassa dei registri di pipeline della figura 6.33 è possibile vedere la sequenza dei numeri dei registri di destinazione procedere da sinistra verso destra ad ogni periodo di clock. Il numero del registro scritto nello stadio WB è fornito dal registro di pipeline MEM/WB.
- Quando uno stadio non è attivo i valori dei segnali di controllo che non sono affermati sono indicati con 0 oppure X (per indicarne l'indifferenza).

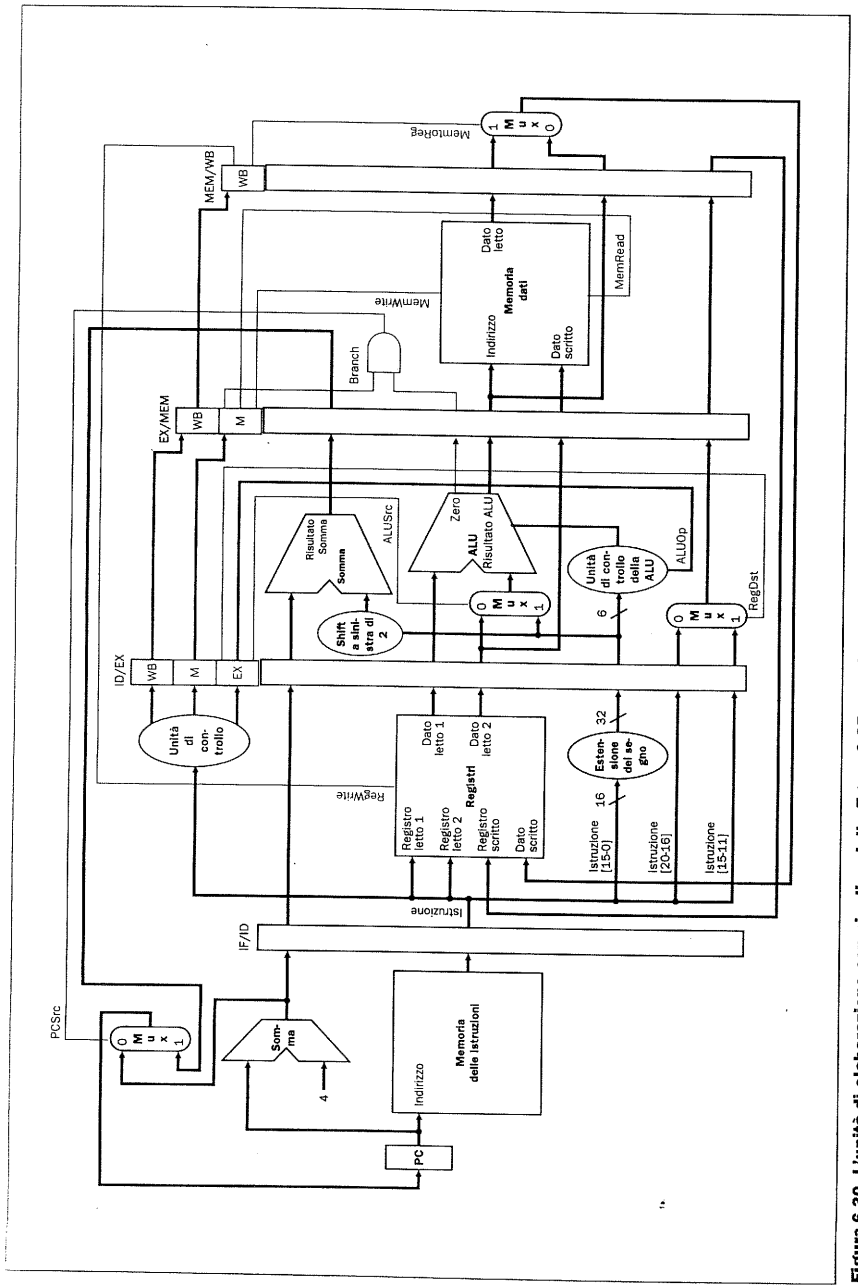


Figura 6.30 L'unità di elaborazione con pipeline della figura 6.25 completa dei segnali di controllo connessi alle parti dei registri di pipeline contenenti le informazioni di controllo. I valori dei segnali di controllo degli ultimi tre stadi sono generati durante lo stadio di decodifica dell'istruzione e sono posti nel registro di pipeline ID/EX. In ciascuno stadio sono utilizzati alcuni segnali di controllo, mentre i rimanenti sono inviati allo stadio successivo della pipeline.

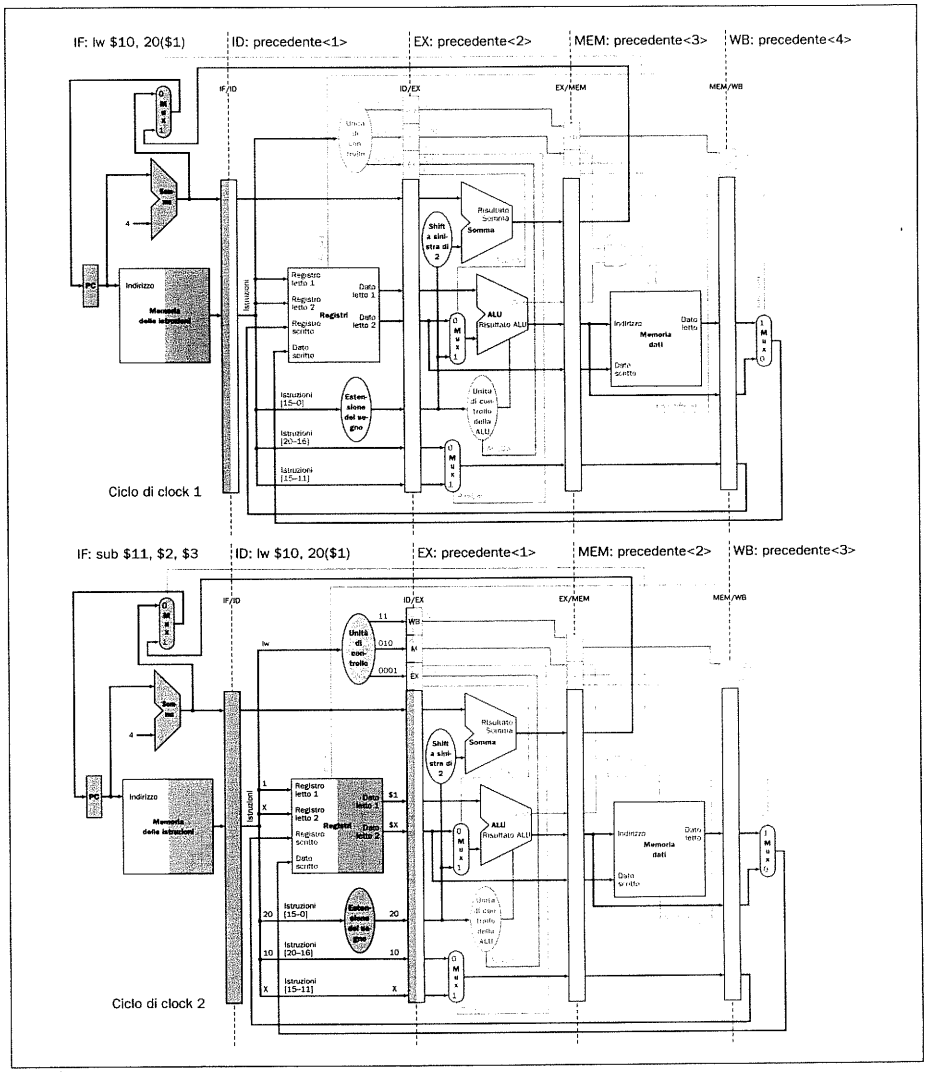


Figura 6.31 Periodi di clock 1 e 2. Il termine "precedente<*i*>" indica la *i*-esima istruzione prima della `lw`. Nel diagramma superiore l'istruzione `lw` si trova nello stadio IF; alla fine del periodo di clock l'istruzione `lw` si trova nel registro di pipeline IF/ID. Nel secondo ciclo di clock, nell'unità di elaborazione riportata in basso, `lw` si sposta nello stadio ID, mentre `sub` entra nello stadio IF. Si noti che nello stadio ID sono riportati i valori dei campi dell'istruzione ed i registri sorgente selezionati. Vengono quindi scritti all'interno del registro di pipeline ID/EX gli operandi di `lw` (il registro `$1` e la costante 20), insieme al numero 10, che rappresenta il numero del registro destinazione per la stessa istruzione. I bit da 15 ad 11 valgono 0, ma viene riportata una X per indicare che tale campo non svolge alcun ruolo nell'ambito dell'istruzione considerata. La parte superiore del registro di pipeline ID/EX mostra i valori dei segnali di controllo da utilizzare negli stadi successivi dell'istruzione `lw`; tali valori sono deducibili dalla riga `lw` della tabella di figura 6.28.

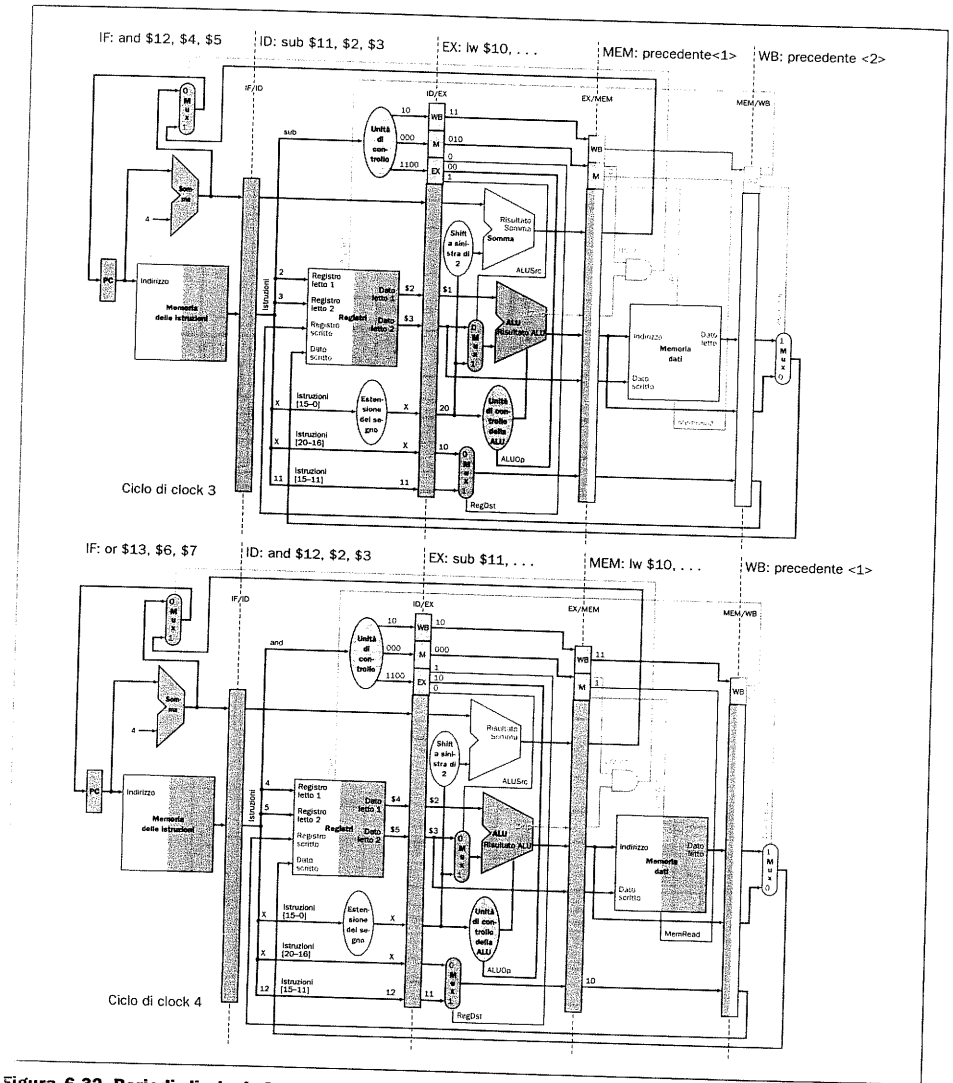


Figura 6.32 Periodi di clock 3 e 4. Nel diagramma superiore l'istruzione `lw` nel terzo ciclo di clock si porta nello stadio EX, sommando `$1` a 20 per ottenere l'indirizzo che verrà scritto nel registro di pipeline EX/MEM (l'istruzione è scritta come `lw $10, ...` nello stadio EX poiché l'identità degli operandi dell'istruzione non è più richiesta in tale stadio ed in quelli successivi; in questa versione della pipeline le azioni di EX, MEM e WB dipendono solamente dall'istruzione e dal suo registro di destinazione o dal suo indirizzo di salto). Nello stesso tempo l'istruzione `sub` raggiunge lo stadio ID, leggendo i registri `$2` e `$3`, mentre l'istruzione `and` inizia in IF. Nel quarto ciclo (in basso) `lw` si sposta nello stadio MEM, leggendo la memoria all'indirizzo specificato da EX/MEM. Contemporaneamente la ALU sottrae `$3` da `$2` e pone la differenza in EX/MEM, la `and` legge i registri `$4` e `$5` in ID e l'istruzione `or` raggiunge IF. I due diagrammi illustrano come i segnali di controllo siano creati nello stadio ID e utilizzati progressivamente negli stadi di pipeline seguenti.

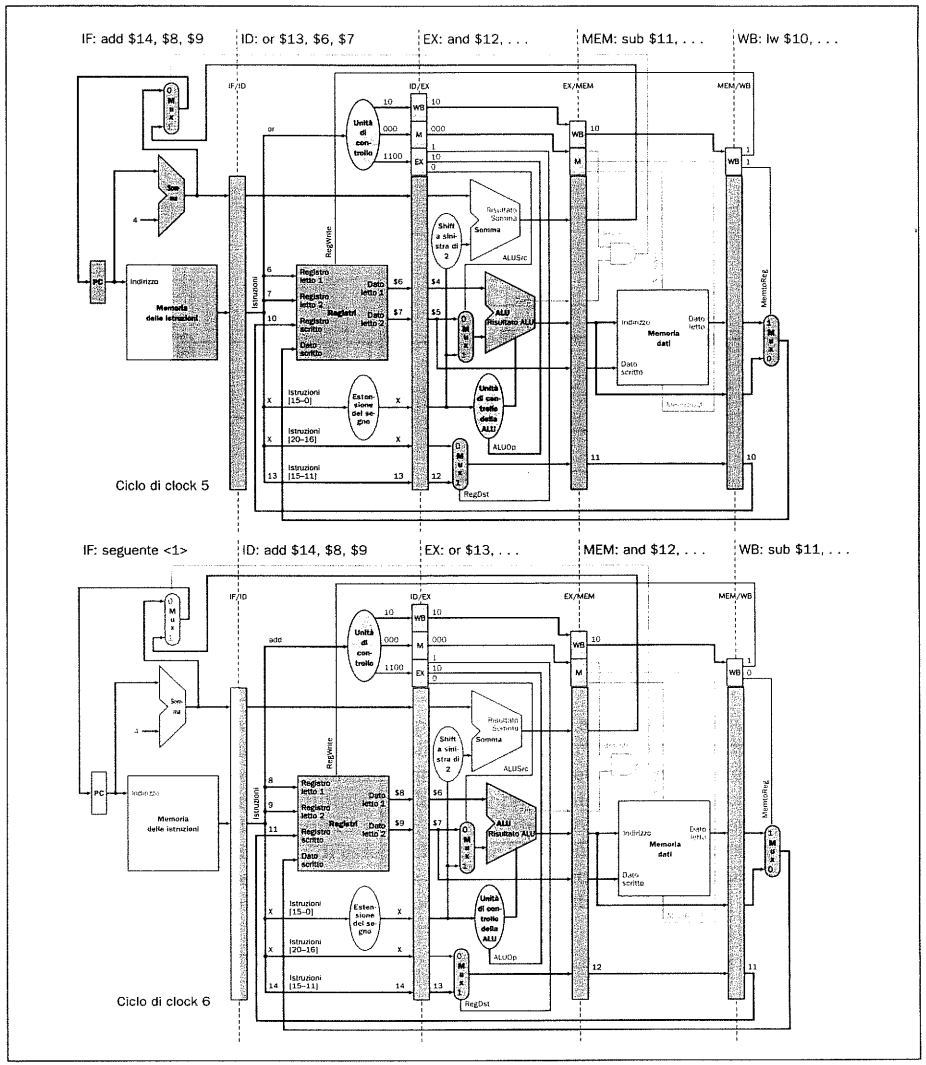


Figura 6.33 Periodi di clock 5 e 6. Con l'ingresso nello stadio IF, nella parte superiore della figura, dell'istruzione `add`, ultima istruzione di questo esempio, tutte le istruzioni sono attive. L'istruzione `lw` viene completata scrivendo il dato presente in MEM/WB nel registro 10: sia il dato che il numero del registro sono in MEM/WB. Nello stesso ciclo di clock, `sub` invia la differenza presente in EX/MEM verso MEM/WB e le restanti istruzioni fanno un passo avanti. Nel ciclo di clock successivo, `sub` usa il valore in MEM/WB per scrivere nel registro numero 11, specificato ancora una volta da MEM/WB. Le restanti istruzioni replicano quanto già visto: la ALU calcola l'OR di `$6` e `$7` per l'istruzione `or` presente nello stadio EX, mentre i registri `$8` e `$9` vengono letti nello stadio ID a beneficio dell'istruzione `add`. Le istruzioni che seguono la `add` sono indicate come se non fossero attive per dare maggior enfasi alle cinque istruzioni dell'esempio; con il termine "seguente<*i*>" si indica la *i*-esima istruzione dopo la `add`.

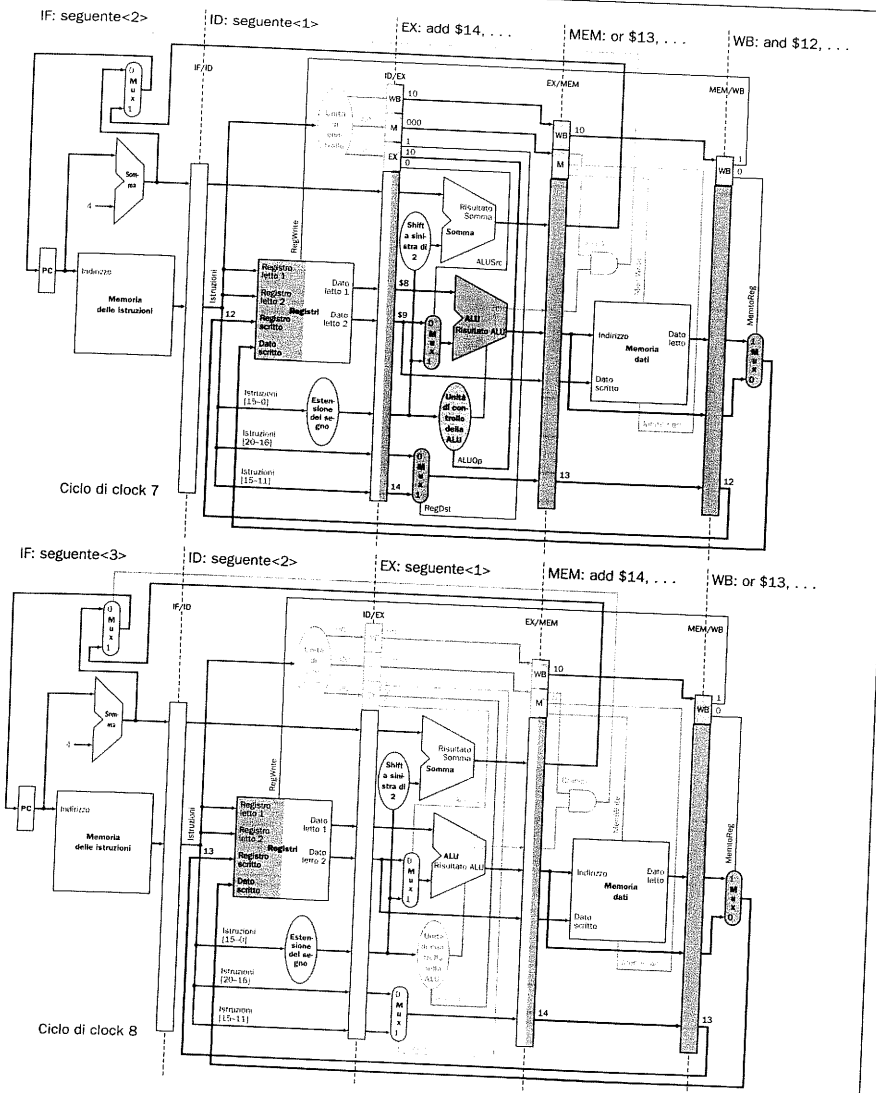


Figura 6.34 Periodi di clock 7 e 8. Nell'unità di elaborazione riportata in alto, l'istruzione add somma nello stadio EX i valori corrispondenti ai registri $8 e $9. Il risultato dell'istruzione or viene copiato da EX/MEM a MEM/WB nello stadio MEM, mentre lo stadio WB scrive nel registro $12 il risultato dell'istruzione and contenuto in MEM/WB. Si noti che i segnali di controllo sono non affermati (posti a 0) nello stadio ID, poiché non vi è alcuna istruzione in esecuzione. Nel ciclo di clock successivo (in basso) lo stadio WB scrive il risultato nel registro $13, completando così l'istruzione or, mentre lo stadio MEM passa il valore della somma di add da EX/MEM a MEM/WB. Le istruzioni che seguono la add sono supposte inattive per motivi didattici.

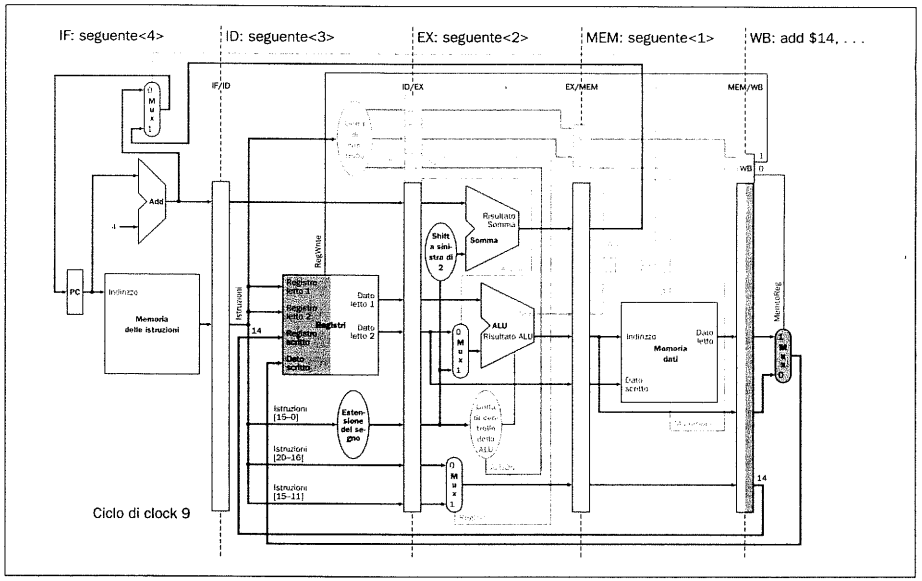


Figura 6.35 Periodo di clock 9. Lo stadio WB scrive la somma presente in MEM/WB nel registro $14, completando l'esecuzione della add e quindi della sequenza di cinque istruzioni considerata. Le istruzioni che seguono la add sono supposte inattive per motivi didattici.

- A differenza del Capitolo 5, in cui la sequenzializzazione delle operazioni dell'unità di controllo richiedeva un circuito dedicato, in questo caso la sequenzializzazione è già insita nella struttura stessa della pipeline. Innanzitutto tutte le istruzioni richiedono lo stesso numero di cicli di clock, quindi non è necessario controllare specificatamente la durata di ciascuna di esse; inoltre tutte le informazioni relative al controllo vengono calcolate durante la fase di decodifica dell'istruzione e successivamente passate ai diversi stadi grazie ai registri di pipeline.

6.4 Criticità sui dati e propagazione

Tutto ciò salta meno all'occhio.

Tallulah Bankhead, comunicazione ad Alexander Wollcott, 1922

Gli esempi visti nel paragrafo precedente mostrano le potenzialità dell'esecuzione con la tecnica delle pipeline ed illustrano le modalità con cui l'hardware implementa tale meccanismo. A questo punto è però necessario abbandonare le condizioni ideali ed affrontare la situazione relativa alle istruzioni reali. Le istruzioni delle figure dalla 6.31 alla 6.35 erano indipendenti: nessuna utilizzava i risultati calcolati

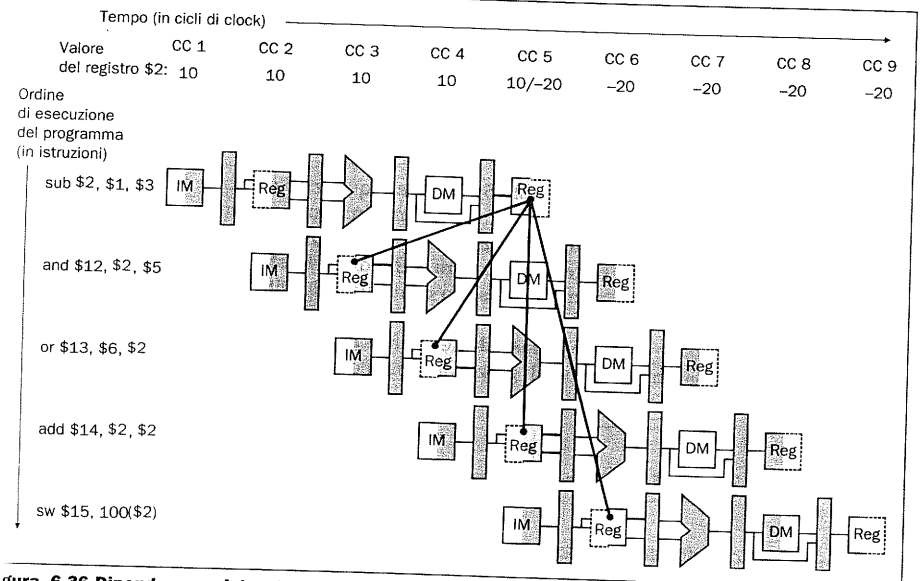
da una delle altre. Si è già visto però nel Paragrafo 6.1 che le criticità sui dati costituiscono degli ostacoli all'esecuzione in pipeline.

Ecco una sequenza di istruzioni con molte dipendenze, indicate in colore:

```
sub  $2,  $1, $3   # sub scrive nel registro $2
and  $12, $2, $5   # il primo operando ($2) dipende da sub
or   $13, $6, $2   # il secondo operando ($2) dipende da sub
add  $14, $2, $2   # il primo ($2) ed il secondo ($2) dipendono
                   # da sub
sw   $15, 100($2)  # la base più $2 dipende da sub
```

Le ultime quattro istruzioni dipendono tutte dal risultato della prima istruzione, posto nel registro $2. Se il registro $2 avesse il valore 10 prima dell'istruzione di sottrazione, e –20 dopo, il programmatore vorrebbe che nelle istruzioni seguenti, che fanno riferimento al registro $2, fosse utilizzato il valore –20.

Come si comporterebbe la sequenza se venisse eseguita sulla pipeline vista in precedenza? La figura 6.36 descrive l'esecuzione delle istruzioni considerate utilizzando una rappresentazione della pipeline a più cicli di clock. Per illustrare ciò che succede all'interno della pipeline durante l'esecuzione della sequenza di istruzioni, la parte in alto della figura 6.36 riporta il valore del registro $2 all'inizio di ciascun ciclo di clock.



igura 6.36 Dipendenze esistenti in una sequenza di cinque istruzioni: si utilizza un'unità di elaborazio- e stilizzata per meglio evidenziare le dipendenze. Tutte le azioni soggette a dipendenza sono disegnate in ~~ulete~~; nella parte alta "CC i" indica l'*i*-esimo ciclo di clock. La prima istruzione scrive in $2 e tutte le successive ~~ruzioni~~ leggono $2. L'operazione di scrittura in tale registro avviene nel quinto ciclo di clock per cui il valore ~~retto~~ non è disponibile prima di questo ciclo; il register file legge il valore scritto in tale ciclo di clock. Le linee ~~tuate~~ che partono dall'unità di elaborazione in alto e vanno verso quelle in basso mostrano le dipendenze. ~~nelle~~ che corrispondono a dei collegamenti a ritroso nel tempo costituiscono delle *criticità sui dati nella pipeline* ~~pelline data hazard~~). (pipeline data hazard)

colore istruzioni corretto colorate Quelle

Una delle potenziali criticità può essere risolta progettando opportunamente l'hardware che implementa il register file: cosa succede quando un registro è letto e scritto nello stesso ciclo di clock? Si può fare l'assunzione che la scrittura avvenga nella prima metà del ciclo di clock e la lettura nella seconda metà: in questo modo la lettura fornisce il valore che è stato scritto. Questa situazione si verifica in molte implementazioni dei register file, ed in questo caso non si ha dunque criticità sui dati.

La figura 6.36 mostra che i valori letti dal registro $2 *non* possono essere quelli prodotti dall'istruzione sub, a meno che l'operazione di lettura non avvenga durante il quinto ciclo di clock o nei successivi. Le istruzioni che leggerebbero il valore corretto (-20) sarebbero quindi add e sw; and e or leggerebbero invece il valore scorretto 10. Se si usa la convenzione adottata nella figura, questi problemi saltano agli occhi quando viene tracciata una linea di dipendenza a ritroso nel tempo. Di conseguenza, nella figura 6.36 vi sono problemi con le istruzioni and e or, dal momento che esse dipendono da un valore che viene scritto successivamente.

Interfaccia Hardware Software

Verranno introdotti tra poco degli schemi circuitali atti a risolvere il problema delle criticità sui dati. Un'altra possibilità consiste nel definire una regola secondo la quale le criticità sui dati non sono accettate: il compilatore ha il divieto di generare sequenze quali quella composta dalle cinque istruzioni viste sopra. Per esempio il compilatore dovrebbe inserire due istruzioni indipendenti tra l'istruzione sub e l'istruzione and, facendo sì che la criticità scompaia. Quando non si possono trovare simili istruzioni, il compilatore inserisce istruzioni che sono sicuramente indipendenti: le istruzioni nop. L'abbreviazione sta per "no operation" (nessuna operazione), dal momento che l'istruzione nop non modifica alcun dato e non genera alcun risultato. Il codice che segue usa le istruzioni nop per ottenere il risultato corretto:

```
sub   $2, $1, $3
nop
nop
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

Benché tale codice funzioni correttamente con la pipeline descritta, le due istruzioni nop occupano 2 cicli di clock che non svolgono alcuna funzione utile. Come si è detto nel Paragrafo 6.1 queste dipendenze compaiono troppo frequentemente perché ci si possa affidare ai compilatori per risolvere il problema.

È necessario innanzitutto accorgersi della criticità, per poi *propagare* il valore corretto in avanti in modo da risolvere la criticità, come visto in precedenza. Ad un'analisi più attenta ci si accorge che quando un'istruzione cerca di leggere nel proprio stadio EX un registro che una precedente istruzione scrive nel proprio stadio WB, è necessario disporre dei valori sugli ingressi della ALU.

Una notazione che assegni dei nomi ai campi dei registri di pipeline permette di formalizzare le dipendenze con maggior precisione. Per esempio la notazione "ID/EX.RegistroRs" fa riferimento al numero di un registro il cui valore si trova nel registro di pipeline ID/EX; in altre parole, quello che proviene dalla prima porta di ingresso del register file. La prima parte del nome, a sinistra del punto, è il nome del registro di pipeline; la seconda parte corrisponde al nome del campo all'interno del registro. Se si utilizza questa notazione le due coppie di condizioni che generano una criticità si possono esprimere come segue:

1a. EX/MEM.RegistroRd = ID/EX.RegistroRs

1b. EX/MEM.RegistroRd = ID/EX.RegistroRt

2a. MEM/WB.RegistroRd = ID/EX.RegistroRs

2b. MEM/WB.RegistroRd = ID/EX.RegistroRt

La prima criticità nella sequenza introdotta prima del capoverso "Interfaccia Hardware-Software" era relativa al registro `$2` e si verificava tra la scrittura del risultato dell'istruzione `sub $2, $1, $3` e l'operazione di lettura del primo operando dell'istruzione `and $12, $2, $5`. Tale criticità può essere rilevata quando l'istruzione `and` si trova nello stadio EX e l'istruzione precedente si trova nello stadio MEM; si tratta quindi di una criticità di tipo 1a:

EX/MEM.RegistroRd = ID/EX.RegistroRs = $2

Esempio

Rilevamento delle dipendenze

Si classifichino le dipendenze esistenti all'interno della seguente sequenza (la stessa vista in precedenza):

```
sub $2,   $1, $3  # il registro $2 è scritto da sub
and $12,  $2, $5  # il primo operando ($2) è scritto da sub
or  $13,  $6, $2  # il secondo operando ($2) è scritto da
sub
add $14,  $2, $2  # il primo ($2) ed il secondo ($2) sono
                  # scritti da sub
sw  $15,  100($2) # la base più $2 dipende da sub
```

Soluzione

Come già detto, la criticità tra `sub` e `and` è di tipo 1a. Le altre criticità sono:

- tra `sub` e `or` vi è una criticità di tipo 2b:

  `MEM/WB.RegistroRd = ID/EX.RegistroRt = $2;`

- Le due dipendenze tra `sub` e `add` non sono delle criticità, dal momento che il register file produce i dati corretti durante lo stadio ID dell'istruzione `add`.
- Non vi è criticità sui dati tra `sub` e `sw`, poiché `sw` legge `$2` nel ciclo di clock successivo a quello in cui `sub` scrive in `$2`.

Poiché non tutte le istruzioni scrivono nei registri, questa strategia non è precisa: in alcuni casi propaga dei valori anche quando non è necessario. Una possibile

soluzione consiste nel verificare se il segnale RegWrite è attivo; se si esamina il campo di controllo WB del registro di pipeline durante le fasi EX e MEM si può determinare se il segnale RegWrite è stato affermato. Inoltre il MIPS richiede che qualunque uso di `$0` come operando debba corrispondere ad un valore dell'operando pari a zero. Nel caso in cui un'istruzione nella pipeline abbia `$0` come destinazione (ad esempio `sll $0, $1, 2`), si cerca di evitare che il valore eventualmente non nullo del risultato sia propagato in avanti. Se si evita di propagare i risultati destinati al registro `$0` si alleggeriscono l'assemblatore ed il compilatore da ogni vincolo relativo al divieto di usare `$0` come destinazione. Le condizioni riportate sopra funzionano quindi correttamente se si aggiunge EX/MEM.RegistroRd ≠ 0 alla prima condizione di criticità e MEM/WB.RegistroRd ≠ 0 alla seconda.

Ora che si è in grado di rilevare le criticità, metà del problema è risolto: si devono a questo punto propagare i valori corretti.

La figura 6.37 mostra le dipendenze tra i registri di pipeline e gli ingressi della ALU per la stessa sequenza di codice della figura 6.36. La differenza consiste nel fatto che la linea di dipendenza parte da un registro di pipeline anziché dover attendere lo stadio WB per scrivere nel register file. In tal modo i dati necessari sono disponibili in tempo utile per le istruzioni successive ed i registri di pipeline contengono i dati che devono essere propagati.

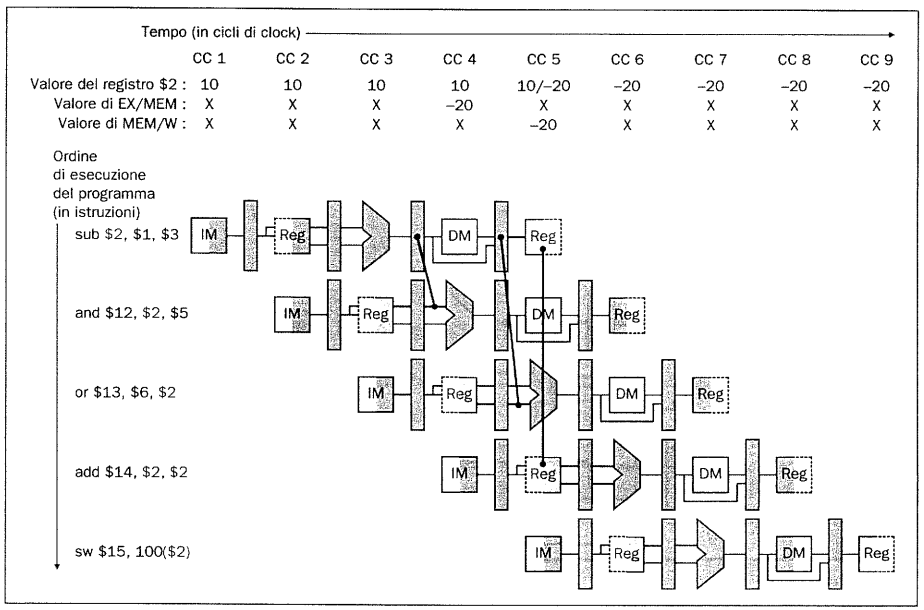
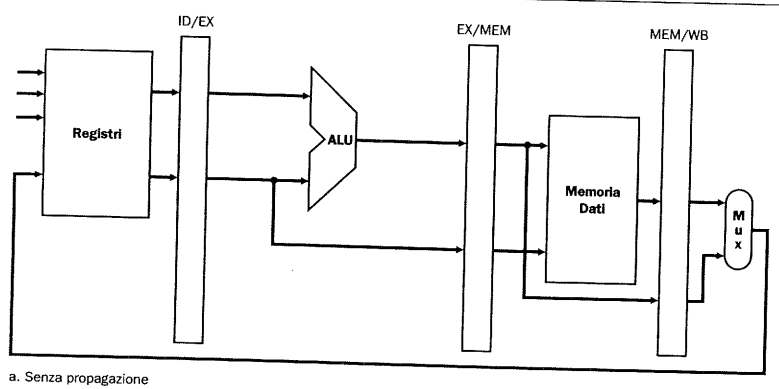


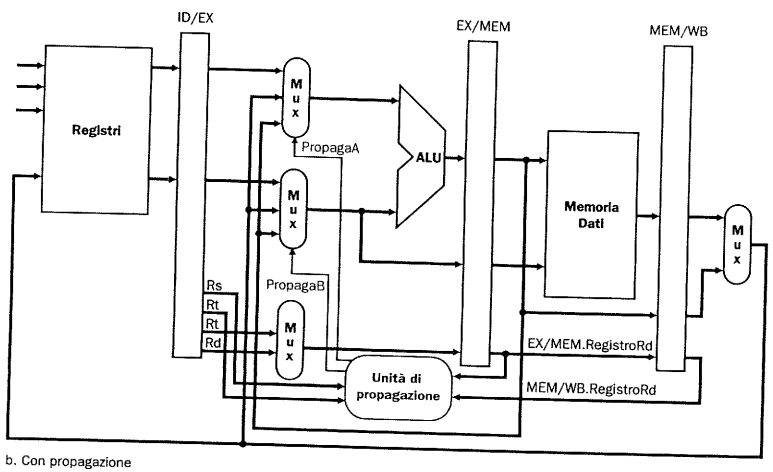
Figura 6.37 Le dipendenze tra i registri di pipeline si muovono in avanti nel tempo, così che è possibile fornire alla ALU gli ingressi necessari per l'esecuzione dell'istruzione `and` e dell'istruzione `or` propagando i risultati presenti nei registri di pipeline. I valori nei registri di pipeline mostrano che il valore desiderato è disponibile prima che sia scritto nel register file. Si assume che il register file propaghi i valori che sono letti e scritti nello stesso ciclo di clock, per cui `add` non va in stallo, ed i valori possono provenire dal register file anziché da un registro di pipeline. La propagazione del register file, ossia la possibilità per le letture di ottenere il valore prodotto da una scrittura eseguita nello stesso periodo di clock, è la ragione per cui nel quinto ciclo di clock il registro `$2` contiene il valore 10 all'inizio del ciclo e -20 alla fine.

Se gli ingressi della ALU possono essere presi da *qualunque* registro di pipeline, anziché soltanto da ID/EX, allora è possibile propagare i dati corretti. Aggiungendo dei multiplexer sugli ingressi della ALU ed utilizzando dei segnali di controllo adeguati, è possibile far funzionare la pipeline alla velocità massima in presenza delle dipendenze di dato considerate.

Per ora si farà l'assunzione che le sole istruzioni che è necessario propagare siano le quattro istruzioni di formato-R: `add`, `sub`, `and` e `or`. La figura 6.38 mostra un dettaglio della ALU e del registro di pipeline prima e dopo l'aggiunta della pro-



a. Senza propagazione



b. Con propagazione

Figura 6.38 Nella parte alta sono mostrate la ALU ed i registri di pipeline prima di aggiungere l'hardware che esegue la propagazione. Nella parte bassa i multiplexer sono stati ampliati in maniera da implementare i cammini di propagazione e viene riportata l'unità di propagazione. L'hardware aggiunto è evidenziato con il colore. La figura è tuttavia semplificata, in quanto non comprende vari dettagli relativi all'unità di elaborazione completa, quali ad esempio i circuiti per l'estensione del segno. Si noti che il campo ID/EX.RegistroRt compare due volte (anche se corrisponde ad un singolo segnale): una per realizzare il collegamento con il multiplexer ed una per il collegamento con l'unità di propagazione.

Controllo Multiplexer	Sorgente	Spiegazione
PropagaA=00	ID/EX	Il primo operando della ALU proviene dal register file.
PropagaA=10	EX/MEM	Il primo operando della ALU è propagato a partire dal precedente risultato della ALU.
PropagaA=01	MEM/WB	Il primo operando della ALU è propagato dalla memoria dati o da un precedente risultato della ALU.
PropagaB=00	ID/EX	Il secondo operando della ALU proviene dal register file.
PropagaB=10	EX/MEM	Il secondo operando della ALU è propagato a partire dal precedente risultato della ALU.
PropagaB=01	MEM/WB	Il secondo operando della ALU è propagato dalla memoria dati o da un precedente risultato della ALU.

Figura 6.39 I valori di controllo per i multiplexer di propagazione della figura 6.38. L'immediato con segno che costituisce un ulteriore ingresso della ALU è descritto nell'approfondimento al termine di questo paragrafo.

pagazione. La figura 6.39 mostra i valori delle linee di controllo dei multiplexer della ALU che selezionano o i valori provenienti dal register file oppure uno dei valori propagati.

Il controllo della propagazione si svolge nello stadio EX, dal momento che i multiplexer di propagazione della ALU si trovano in questo stadio. Di conseguenza per determinare se propagare i valori è necessario passare i numeri dei registri usati come operandi dallo stadio ID attraverso il registro di pipeline ID/EX. È disponibile a tal fine anche il campo rt (bit 20-16). Prima della propagazione, il registro ID/EX non aveva nessuna ragione di includere lo spazio per memorizzare il campo rs, quindi ad ID/EX occorre aggiungere rs (bit 25-21).

Si possono ora scrivere sia le condizioni per rilevare le criticità, sia i segnali di controllo per risolverle (le piccole differenze sono evidenziate):

1. criticità EX:

   se (EX/MEM.RegWrite
   e (EX/MEM.RegistroRd $\neq$ 0)
   e (EX/MEM.RegistroRd = ID/EX.RegistroRs)) PropagaA = 10

   se (EX/MEM.RegWrite
   e (EX/MEM.RegistroRd $\neq$ 0)
   e (EX/MEM.RegistroRd = ID/EX.RegistroRt)) PropagaB = 10

In questo caso il risultato proveniente dalla precedente istruzione viene propagato ad uno dei due ingressi della ALU. Se la precedente istruzione sta per scrivere sul register file e l'indice del registro che sta per venire scritto coincide con quello del registro da cui viene letto il valore dell'ingresso A o B della ALU (purché questo non sia il registro 0), allora il multiplexer viene pilotato in maniera da prelevare il valore dal registro di pipeline EX/MEM.

2. criticità MEM:

   se (MEM/WB.RegWrite
   e (MEM/WB.RegistroRd $\neq$ 0)
   e (MEM/WB.RegistroRd = ID/EX.RegistroRs)) PropagaA = 01

   se (MEM/WB.RegWrite
   e (MEM/WB.RegistroRd $\neq$ 0)
   e (MEM/WB.RegistroRd = ID/EX.RegistroRt)) PropagaB = 01

Come già accennato in precedenza, non vi può essere criticità nello stadio WB, dal momento che si assume che il register file fornisca il risultato corretto se l'istruzione nello stadio ID legge lo stesso registro scritto dall'istruzione nello stadio WB. Un tale register file implementa un'altra forma di propagazione, che si verifica all'interno del register file stesso.

Una possibile fonte di problemi è costituita dalle criticità sui dati tra il risultato di un'istruzione nello stadio WB, il risultato di un'istruzione nello stadio MEM, e l'operando sorgente di un'istruzione nello stadio ALU. Se per esempio si somma un vettore di numeri e si scrive il risultato in un singolo registro, si ha una sequenza di istruzioni che leggono e scrivono nello stesso registro:

```
add $1, $1, $2;
add $1, $1, $3;
add $1, $1, $4;
...
```

In questo caso il risultato viene propagato dallo stadio MEM, dal momento che il risultato presente nello stadio MEM è il più recente. Quindi il controllo da eseguire per rilevare la criticità di tipo MEM dovrebbe essere (le aggiunte sono evidenziate) come segue:

se (MEM/WB.RegWrite
e (MEM/WB.RegistroRd ≠ 0)
e (EX/MEM.RegistroRd ≠ ID/EX.RegistroRs)
e (MEM/WB.RegistroRd = ID/EX.RegistroRs)) PropagaA = 01

se (MEM/WB.RegWrite
e (MEM/WB.RegistroRd ≠ 0)
e (EX/MEM.RegistroRd ≠ ID/EX.RegistroRt)
e (MEM/WB.RegistroRd = ID/EX.RegistroRt)) PropagaB = 01

La figura 6.40 mostra l'hardware necessario per implementare la propagazione.

Esempio

Propagazione

Si mostri come funziona la propagazione con la seguente sequenza di istruzioni, nella quale sono state evidenziate le dipendenze:

```
sub   $2, $1, $3
and   $4, $2, $5
or    $4, $4, $2
add   $9, $4, $2
```

Soluzione

Le figure 6.41 e 6.42 mostrano gli eventi relativi ai cicli di clock dal 3 al 6 prodotti dall'esecuzione di queste istruzioni. Nel ciclo di clock 4 l'unità di propagazione vede la scrittura fatta dall'istruzione `sub` nel registro `$2` all'interno dello stadio MEM, mentre l'istruzione `and` all'interno dello stadio EX sta leggendo il registro `$2`. L'unità di propagazione seleziona il registro di pipeline EX/MEM anziché il registro di pipeline ID/EX come ingresso superiore della ALU, in maniera da ottenere il valore corretto per il registro `$2`. La successiva istruzione `or` legge il registro `$4`, scritto dall'istruzione `and`, ed il registro `$2`,

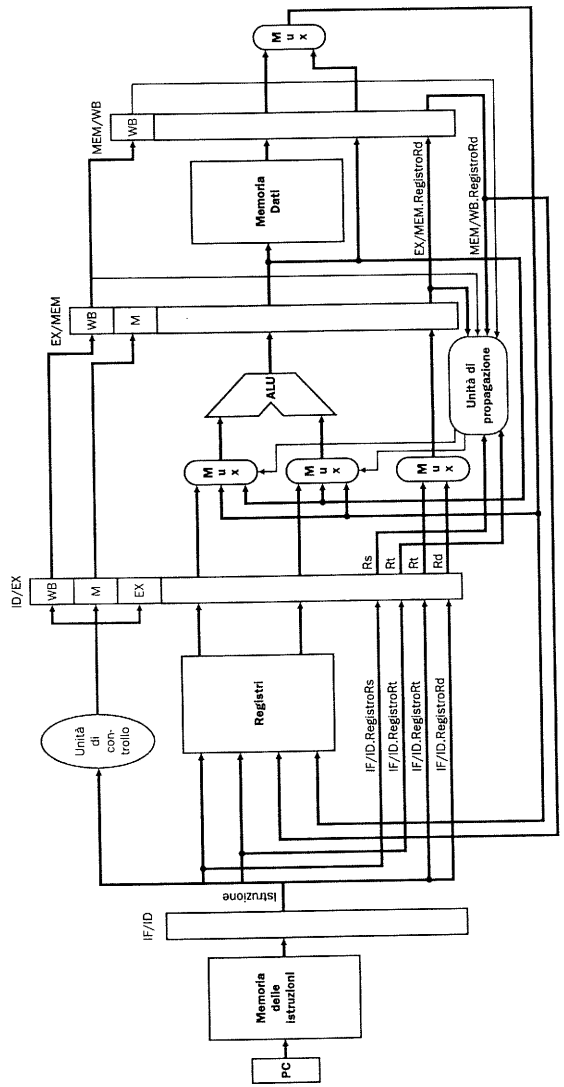


Figura 6.40 L'unità di elaborazione dopo le modifiche per risolvere le criticità tramite la propagazione. Se si confronta con l'unità di elaborazione della figura 6.30, le aggiunte sono costituite dai multiplexer sugli ingressi della ALU. La figura è in parte stilizzata, e tralascia alcuni dettagli relativi all'hardware per i salti e per l'estensione del segno rispetto all'unità di elaborazione completa.

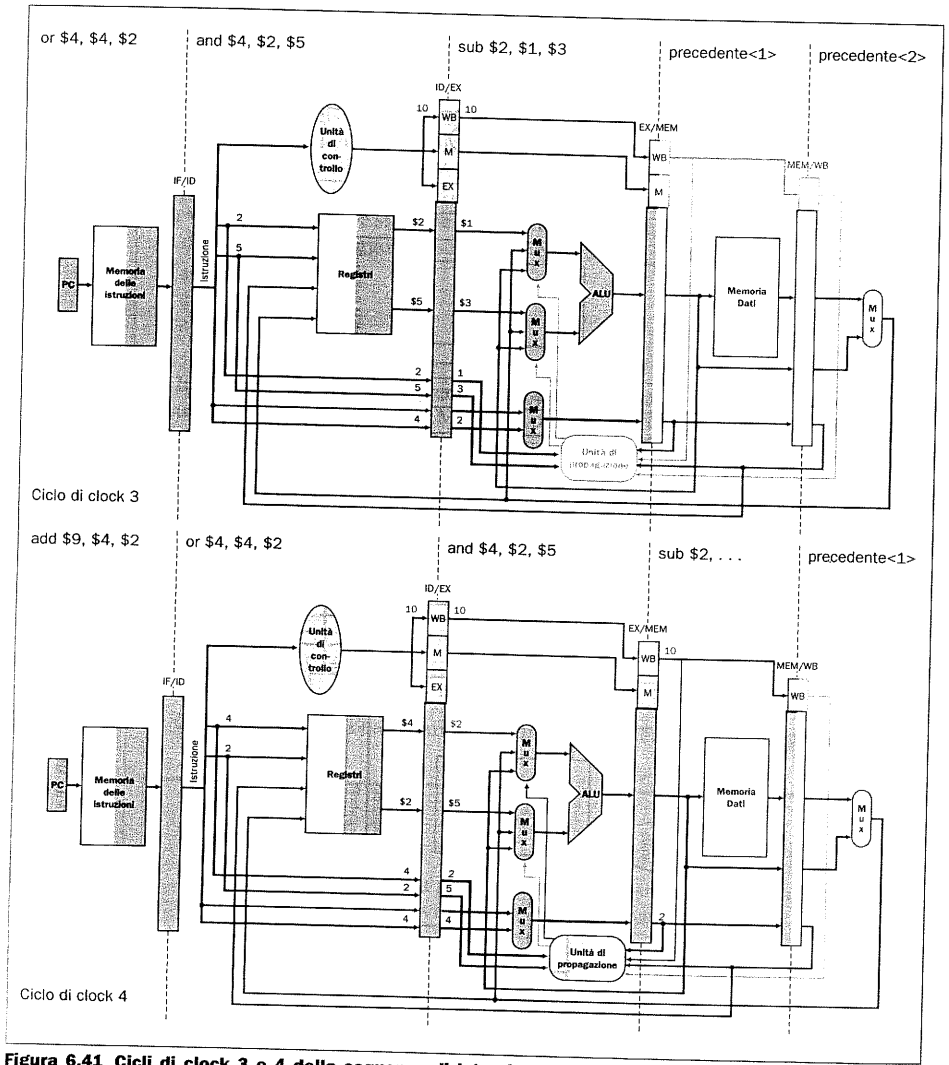


Figura 6.41 Cicli di clock 3 e 4 della sequenza di istruzioni riportata nell'esempio. Le linee in grassetto sono quelle attive in ciascun ciclo di clock, ed i numeri dei registri colorati in corsivo indicano una criticità. L'unità di elaborazione è evidenziata con l'ombreggiatura quando sta propagando dei dati alla ALU. Le istruzioni precedenti a quella di sub sono considerate inattive semplicemente per enfatizzare ciò che avviene in relazione alle quattro istruzioni nell'esempio. I nomi degli operandi sono utilizzati nello stadio EX per il controllo della propagazione, per cui sono stati riportati nell'istruzione che etichetta lo stadio EX. I nomi degli operandi non sono utilizzati negli stadi MEM o WB, e quindi si usa al loro posto il simbolo . . . . Si confronti questa con le figure dalla 6.32 alla 6.35 che mostrano l'unità di elaborazione senza propagazione, in cui ID è l'ultimo stadio ad avere bisogno delle informazioni sugli operandi.

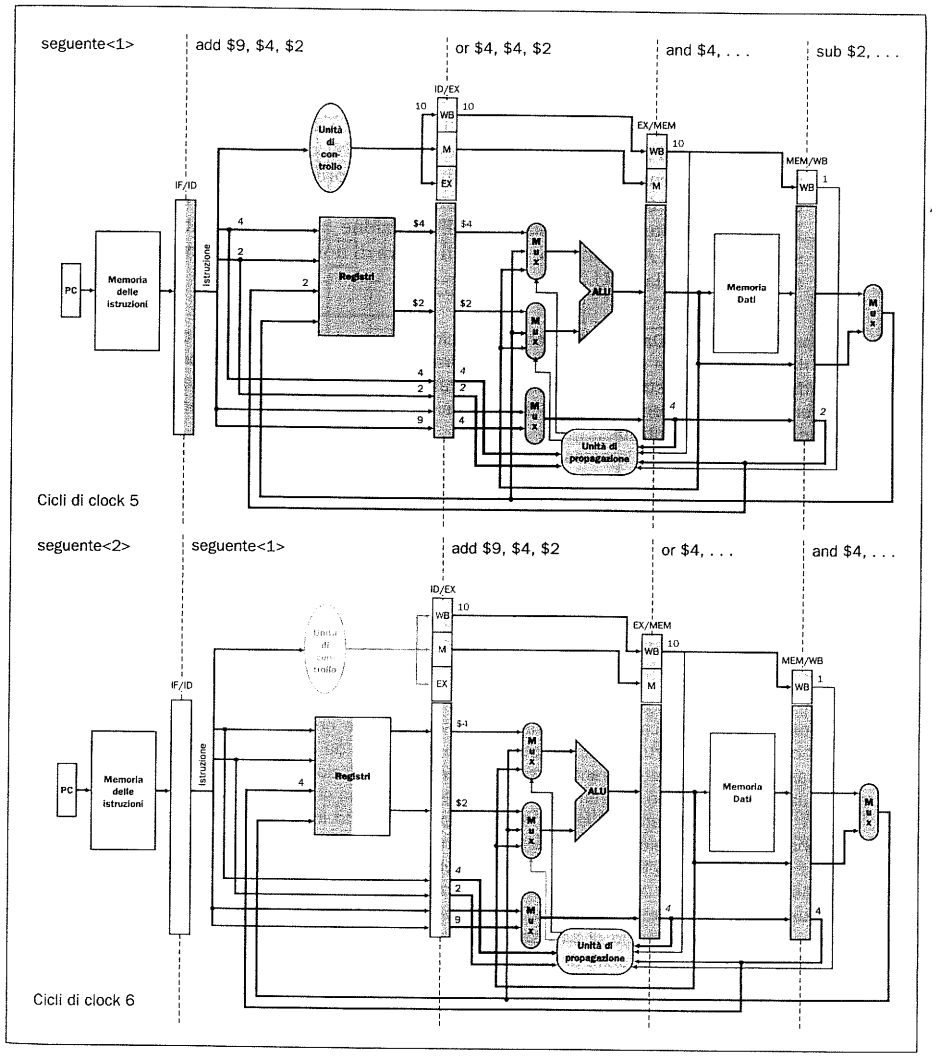


Figura 6.42 Cicli di clock 5 e 6 della sequenza di istruzioni riportata nell'esempio. L'unità di elaborazione è evidenziata quando sta propagando dei dati alla ALU. Le due istruzioni successive a quella di add sono considerate inattive semplicemente per enfatizzare ciò che avviene in relazione alle quattro istruzioni dell'esempio. Le linee in grassetto sono quelle attive in ciascun ciclo di clock, ed i numeri dei registri colorati in corsivo indicano una criticità.

scritto dall'istruzione sub. Nel ciclo di clock 5 l'unità di propagazione seleziona il registro di pipeline EX/MEM come ingresso superiore della ALU, ed il registro di pipeline MEM/WB come ingresso inferiore della ALU. La successiva istruzione add legge sia il registro $4, destinazione dell'istruzione and, sia il registro $2, che è già stato scritto dall'istruzione sub. Si noti che le due precedenti istruzioni scrivono entrambe nel registro $4, così che l'unità di propagazione deve prendere il valore immediatamente precedente (stadio MEM). Nel ciclo di clock 6 l'unità di propagazione seleziona il registro di pipeline EX/MEM, contenente il risultato dell'istruzione or, per alimentare l'ingresso superiore della ALU, mentre utilizza il valore di non-propagazione per l'ingresso inferiore della ALU.

Approfondimento: Dal momento che le criticità sono formalmente definite come tali facendo riferimento ad una specifica unità di elaborazione, le sequenze di istruzioni caratterizzate da dipendenze sui risultati non sono considerate come criticità se la propagazione risolve le dipendenze. Di conseguenza si utilizza il termine "unità di propagazione" anziché "unità di propagazione delle criticità".

Un secondo punto di interesse concerne una possibile soluzione alternativa a quella fornita in questo paragrafo: essa prevede che si determinino i segnali di controllo dei multiplexer sugli ingressi della ALU durante lo stadio ID, scrivendo i relativi valori all'interno di campi aggiuntivi di controllo nel registro di pipeline ID/EX. L'hardware potrebbe in questo modo essere più veloce poiché il tempo necessario per selezionare gli ingressi della ALU è con ogni probabilità sul cammino critico.

L'istruzione nop del MIPS introdotta in precedenza è codificata con una stringa di 0. Essa corrisponde a sll $0, $0, 0, che esegue lo scalamento a sinistra di 0 posizioni del registro 0. L'istruzione non esegue alcuna operazione sul registro 0 (che comunque non potrebbe essere modificato) e quindi è utilizzata come nop dal software del MIPS.

Approfondimento: La propagazione può essere di aiuto nei confronti delle criticità anche nel caso in cui le istruzioni di tipo store siano dipendenti da altre istruzioni. Poiché esse utilizzano soltanto un dato durante lo stadio MEM, la propagazione è facile. Si consideri tuttavia il caso in cui una load sia immediatamente seguita da una store. È allora necessario aggiungere ulteriore hardware per la propagazione per far sì che le copie da memoria a memoria siano eseguite più velocemente. Se si ridisegna la figura 6.37 sostituendo le istruzioni sub e and con lw e sw, si vede che è possibile evitare uno stallo, poiché il dato è presente nel registro MEM/WB di un'istruzione load in tempo per essere usato nello stadio MEM di un'istruzione store. Per realizzare questa opzione sarebbe necessario aggiungere la propagazione all'interno dello stadio di accesso alla memoria. L'Esercizio 6.17 esamina le modifiche che vanno eseguite sull'unità di elaborazione per evitare questo tipo di criticità.

L'ingresso immediato con segno della ALU richiesto da load e store manca nell'unità di elaborazione della figura 6.40. Dal momento che è l'unità di controllo centrale a decidere se selezionare un registro o l'immediato, ed è l'unità di propagazione a scegliere il registro di pipeline da usare quando l'ingresso della ALU è un registro, la soluzione più semplice consiste nell'aggiungere un multiplexer 2:1 che seleziona tra l'output PropagaB del multiplexer e l'immediato con segno. La figura 6.43 mostra questa aggiunta. Si noti che questa soluzione differisce da quanto fatto nel Capitolo 5, nel quale il multiplexer controllato dalla linea ALUSelB era stato espanso per tenere conto dell'ingresso immediato. Questa soluzione risolve anche il problema della propagazione per la store collegando l'uscita del multiplexer di propagazione, che in questo caso contiene il dato della store, al registro di pipeline EX/MEM.

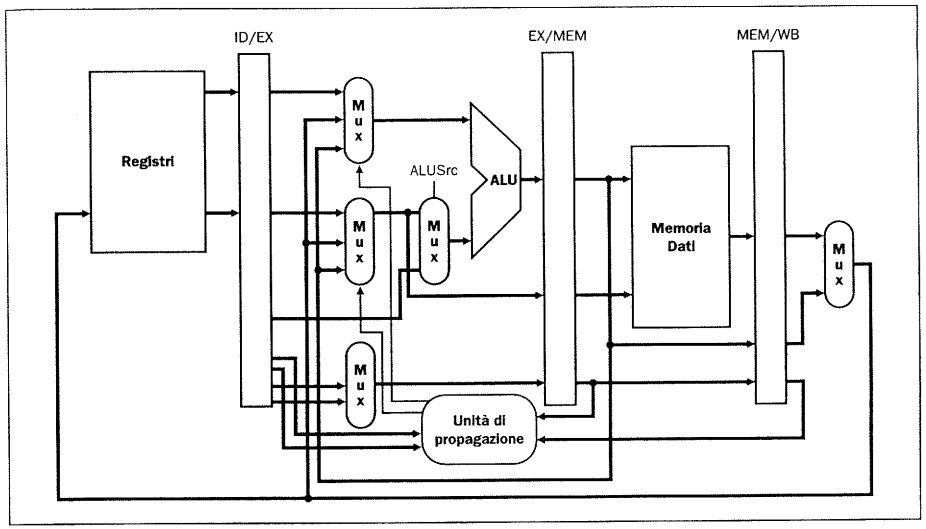


Figura 6.43 Questo dettaglio dell'unità di elaborazione riportata in figura 6.38 contiene un multiplexer 2:1, che è stato aggiunto per selezionare l'operando immediato con segno quale ingresso della ALU.

6.5 Criticità sui dati e situazioni di stallo

Se al primo tentativo non riesci, ridefinisci il concetto di successo.

Anonimo

Come si è detto nel Paragrafo 6.1, un caso nel quale la propagazione non può risolvere tutti i problemi è quello in cui un'istruzione tenti di leggere un registro subito dopo un'istruzione di load che scrive nello stesso registro. La figura 6.44 illustra questa situazione. Durante il ciclo 4 il dato deve ancora essere letto dalla memoria quando la ALU si appresta ad eseguire l'operazione prevista dall'istruzione successiva. Qualcosa deve mandare in stallo la pipeline quando vi è una load seguita da una istruzione che ne legge il risultato.

È quindi necessario disporre, oltre che di una unità di propagazione, anche di una *unità di rilevamento delle criticità*. Questa opera durante lo stadio ID inserendo lo stallo tra il caricamento e l'utilizzo. Poiché si limita a controllare le istruzioni load, il controllo eseguito dall'unità di rilevamento delle criticità coincide con questa semplice condizione:

```
se (ID/EX.MemRead e
  ((ID/EX.RegistroRt = IF/ID.RegistroRs) o
   (ID/EX.RegistroRt = IF/ID.RegistroR )
   metti in stallo la pipeline
```

La prima linea verifica se l'istruzione è una load: l'unica istruzione che legge dati dalla memoria è infatti la load. Le successive due linee verificano se il campo regi-

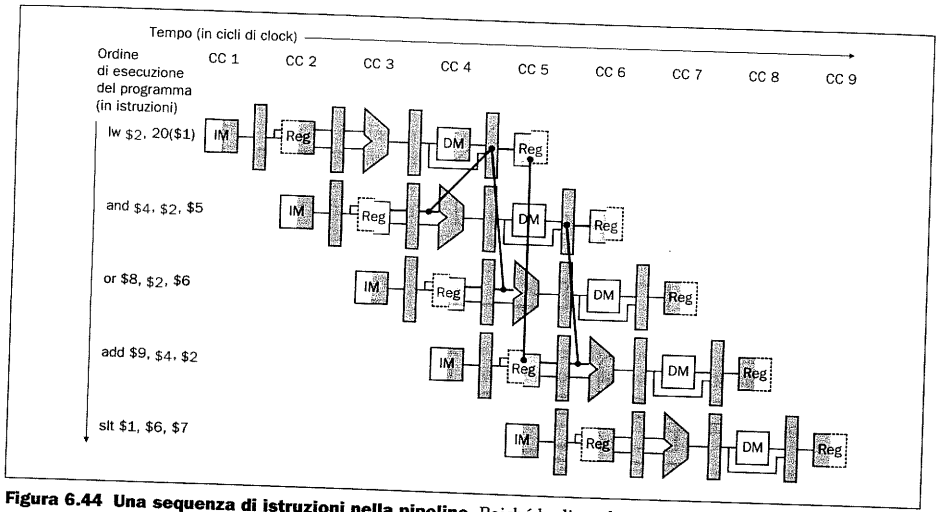


Figura 6.44 Una sequenza di istruzioni nella pipeline. Poiché la dipendenza tra l'istruzione load e l'istruzione successiva (and) procede a ritroso nel tempo, questa criticità non può essere risolta con la propagazione. Di conseguenza questa situazione deve tradursi in uno stallo controllato dall'unità di rilevamento delle criticità.

stro destinazione della load nello stadio EX coincide con uno dei registri sorgente dell'istruzione nello stadio ID. Se la condizione è verificata, l'istruzione viene messa in stallo per un periodo di clock. Dopo questo stallo di un ciclo, la logica di propagazione può gestire la dipendenza e l'esecuzione può procedere. Se non ci fosse la propagazione le istruzioni della figura 6.44 richiederebbero un altro ciclo di stallo.

Se l'istruzione nello stadio ID è messa in stallo, allora anche l'istruzione nello stadio IF deve essere messa in tale stato; diversamente si perderebbe l'istruzione già caricata. Impedire a queste due istruzioni di procedere può essere ottenuto semplicemente impedendo al registro PC e al registro di pipeline IF/ID di cambiare. Se il valore di questi registri è conservato, l'istruzione nello stadio IF continua ad essere letta utilizzando lo stesso PC, ed i registri nello stadio ID continuano ad essere letti utilizzando gli stessi campi istruzione nel registro IF/ID. Riprendendo l'esempio fatto in precedenza, è come se si facesse ripartire la lavatrice con gli stessi vestiti e si lasciasse che l'asciugatrice continuasse a lavorare pur essendo vuota.

Per mettere in stallo la pipeline è necessario ottenere lo stesso effetto che si otteneva inserendo delle istruzioni nop, come visto nel paragrafo Interfaccia Hardware/Software del Paragrafo 6.4, anche se questa volta le "istruzioni" nop cominciano nello stadio EX della pipeline. Nella figura 6.28 si è visto come disattivando (ossia forzando a 0) tutti e nove i segnali di controllo negli stadi EX, MEM e WB sia possibile creare una istruzione che dice di "non far nulla". Identificando la criticità nello stadio ID si può inserire nella pipeline un "buco" (in inglese bubble, cioè bolla) mettendo a 0 i campi di controllo EX, MEM e WB del registro di pipeline ID/EX. Questi valori di controllo si diffondono poi in avanti nella pipeline ad ogni colpo di clock producendo i loro effetti: se i valori dei segnali di controllo sono tutti a 0 non si eseguono scritture in alcun registro o parola di memoria.

La figura 6.45 mostra ciò che avviene in pratica nell'hardware. La criticità costringe le istruzioni and e or a ripetere nel ciclo di clock numero 4 ciò che avevano fatto

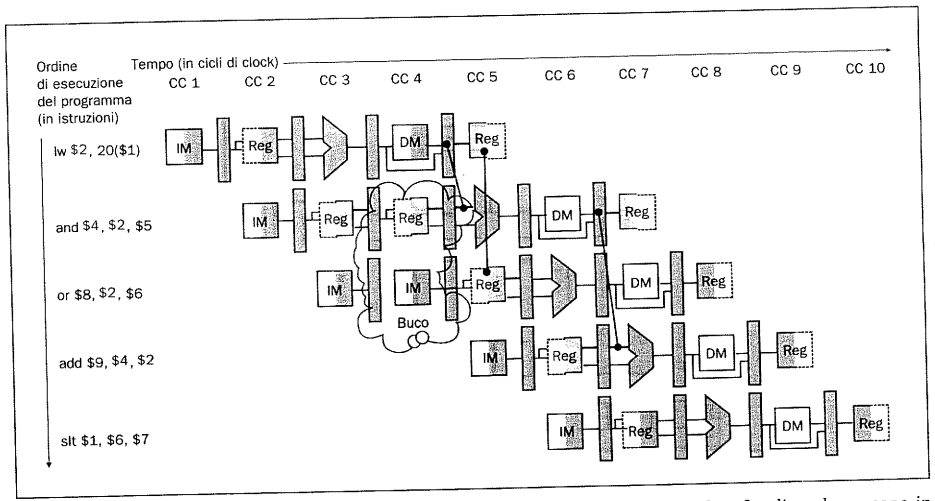


Figura 6.45 Modalità con cui gli stalli vengono realmente inseriti nella pipeline. Le dipendenze sono in avanti, per cui non ci sono criticità sui dati.

nel ciclo di clock numero 3: l'istruzione and legge i registri e viene decodificata, mentre l'istruzione or viene ricaricata dalla memoria delle istruzioni. Una situazione di stallo corrisponde quindi a ripetere del lavoro, ed il suo effetto consiste nell'allungare il tempo delle istruzioni and e or, e nel ritardare il caricamento dell'istruzione add. Come una bolla d'aria in un tubo pieno d'acqua, uno stallo ritarda tutto ciò che vi è dietro, e si muove lungo il tubo fino a giungere al suo estremo.

La figura 6.46 mette in evidenza le connessioni all'interno della pipeline per quanto riguarda sia l'unità di rilevamento delle criticità, sia l'unità di propagazione. Come prima, l'unità di propagazione controlla i multiplexer della ALU per sostituire il valore proveniente da un registro general-purpose con il valore proveniente dall'opportuno registro di pipeline. L'unità di rilevamento delle criticità controlla la scrittura del PC e dei registri IF/ID, oltre al multiplexer che sceglie tra i valori reali di controllo e tutti zeri. L'unità di rilevamento delle criticità mette in stallo e deseleziona i campi di controllo se la condizione di criticità sulla load è verificata.

Le figure dalla 6.47 alla 6.49 mostrano il diagramma dei cicli di clock da 2 al 7 per lo stesso esempio di prima, questa volta con un'istruzione di load al posto dell'istruzione di sottrazione.

Quadro d'insieme

Benché l'hardware possa fare affidamento o meno sul compilatore per risolvere le dipendenze che causano criticità in maniera da assicurare la correttezza dell'esecuzione, il compilatore deve tener conto delle caratteristiche della pipeline per ottenere le massime prestazioni. Diversamente si potranno verificare delle condizioni di stallo che riducono le prestazioni del codice compilato.

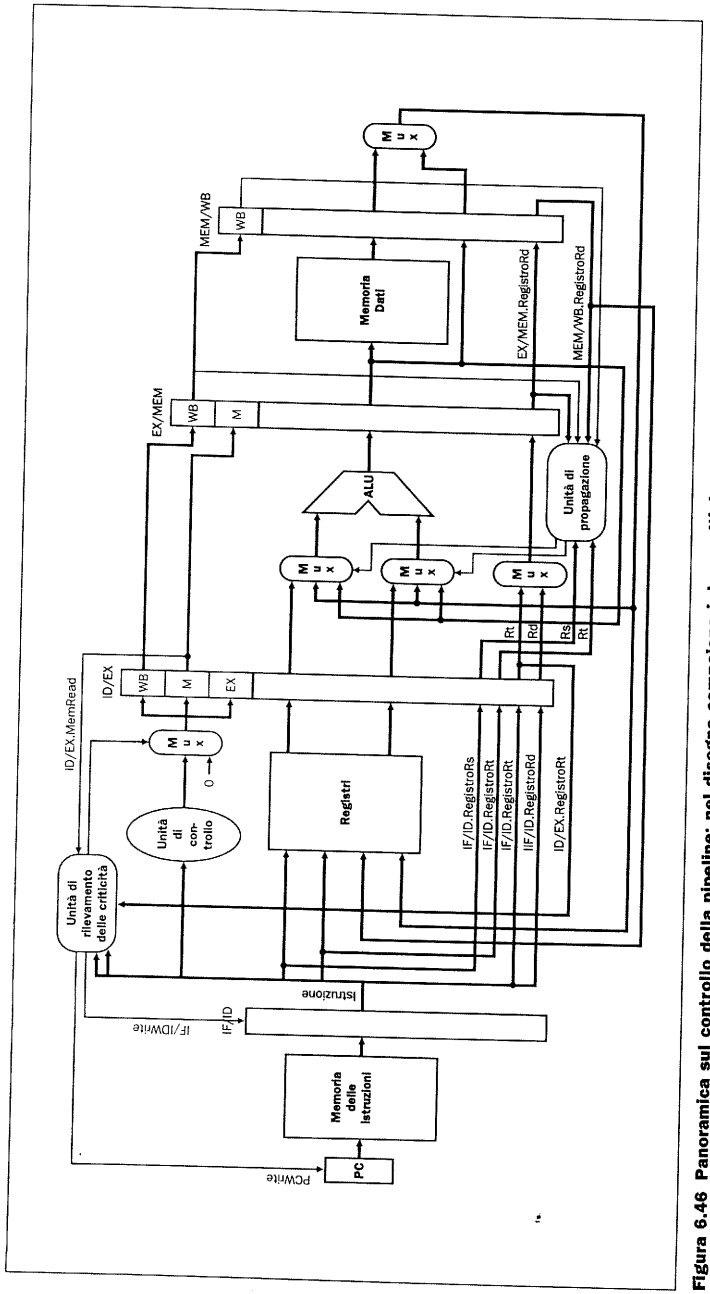


Figura 6.46 Panoramica sul controllo della pipeline: nel disegno compaiono i due multiplexer per la propagazione, l'unità di rilevamento delle criticità e l'unità di propagazione. Benché gli stadi ID e EX siano stati semplificati (mancano la logica per l'estensione del segno dell'immediato e quella per i salti), lo schema mostra gli elementi essenziali dell'hardware per la propagazione.

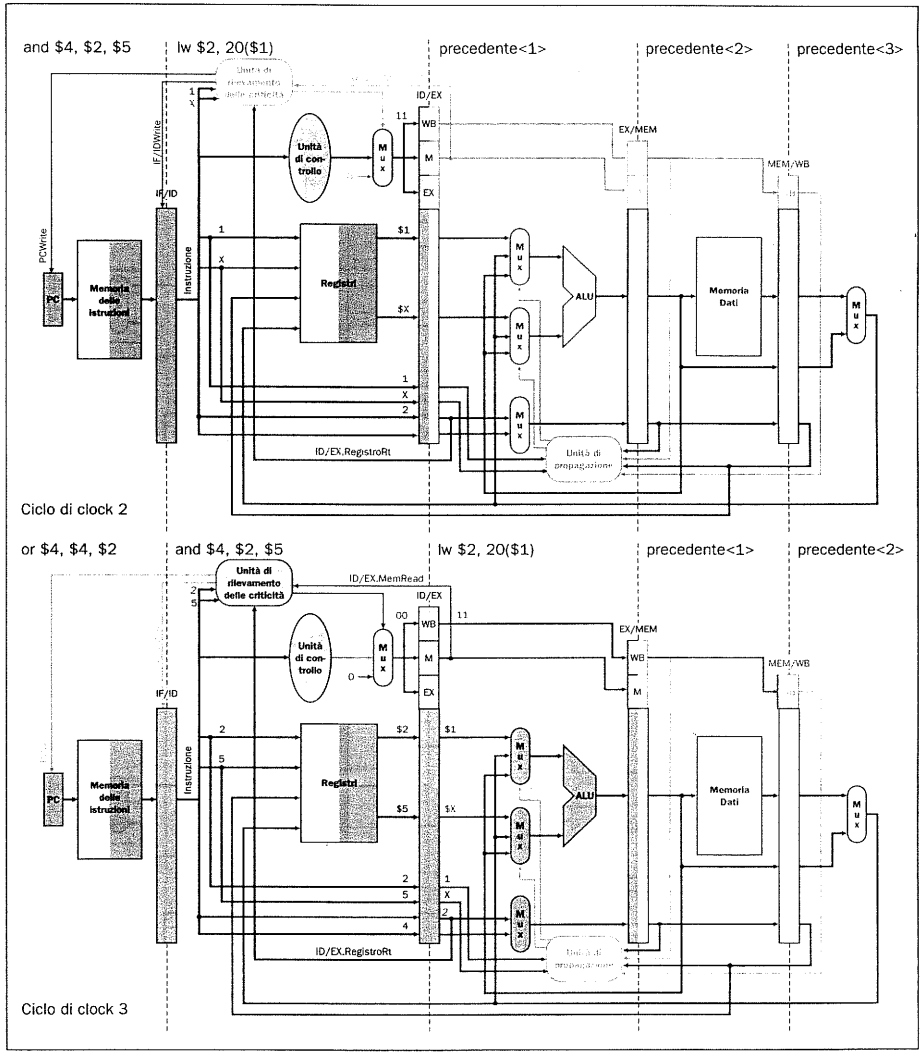


Figura 6.47 I cicli di clock 2 e 3 della sequenza di istruzioni dell'esempio riportato nel testo, modificata sostituendo l'istruzione `sub` **con una load.** Le linee in grassetto sono quelle attive in ciascun ciclo di clock, i numeri dei registri colorati in corsivo indicano una criticità, ed il carattere ... al posto degli operandi significa che lo stadio non necessita dell'informazione concernente la loro identità. La figura riporta anche i valori delle linee di controllo e dei registri, nonché i numeri dei registri più significativi. L'istruzione `and` tenta di leggere il valore fornito dall'istruzione `lw` nel ciclo di clock 3, così che l'unità di rilevamento delle criticità mette in stallo le istruzioni `and` e `or`: per questo l'unità di rilevamento delle criticità è evidenziata.

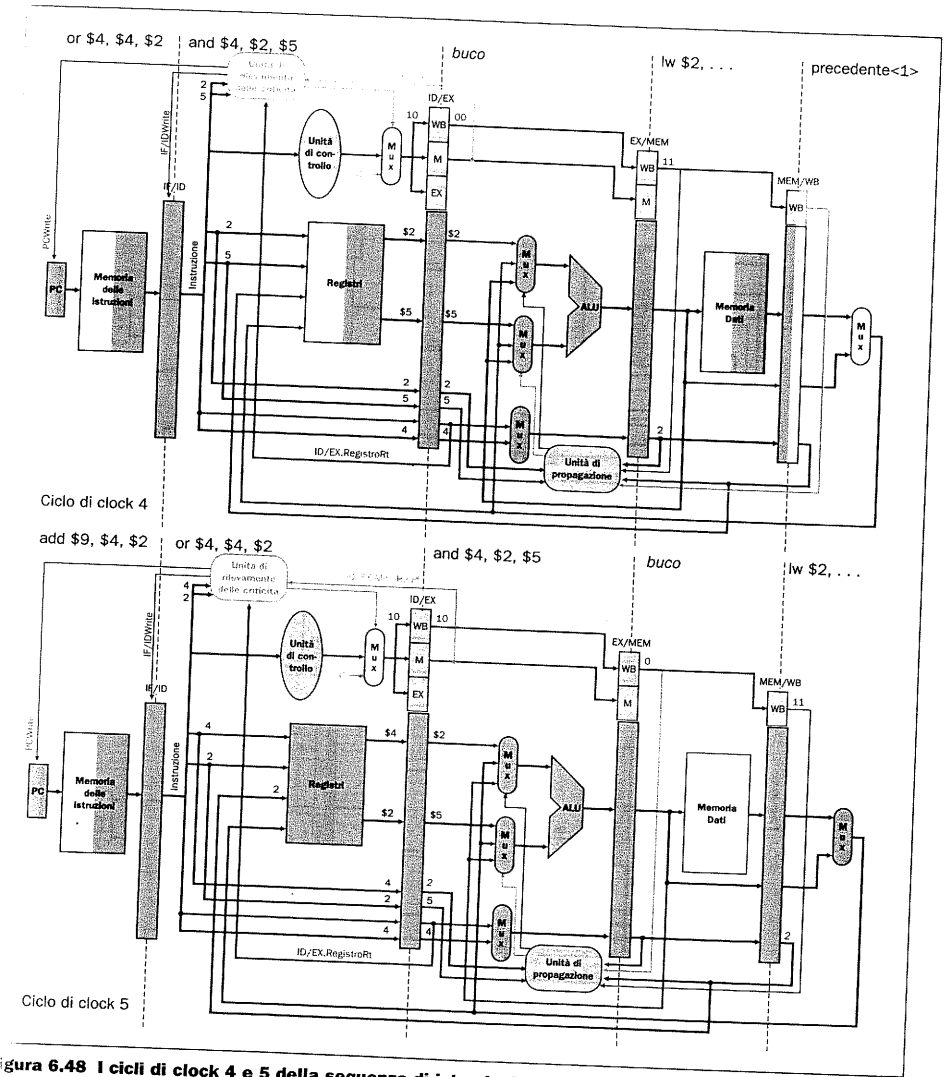


Figura 6.48 I cicli di clock 4 e 5 della sequenza di istruzioni dell'esempio riportato nel testo, modificata sostituendo l'istruzione `sub` **con una load.** In corrispondenza del periodo di clock 4 è inserito un "buco" nella pipeline, e l'istruzione `and` può procedere solo al ciclo di clock 5. L'unità di propagazione è evidenziata nel ciclo di clock 5 poiché sta propagando dei dati dalla `lw` alla ALU. Si noti che nel ciclo di clock 4 l'unità di propagazione propaga l'indirizzo dell'istruzione `lw` come se fosse il contenuto del registro `$2`; gli effetti di questa operazione sono eliminati grazie all'inserimento del buco. Le linee in grassetto sono quelle attive in ciascun ciclo di clock, mentre i numeri dei registri colorati in corsivo indicano una criticità.

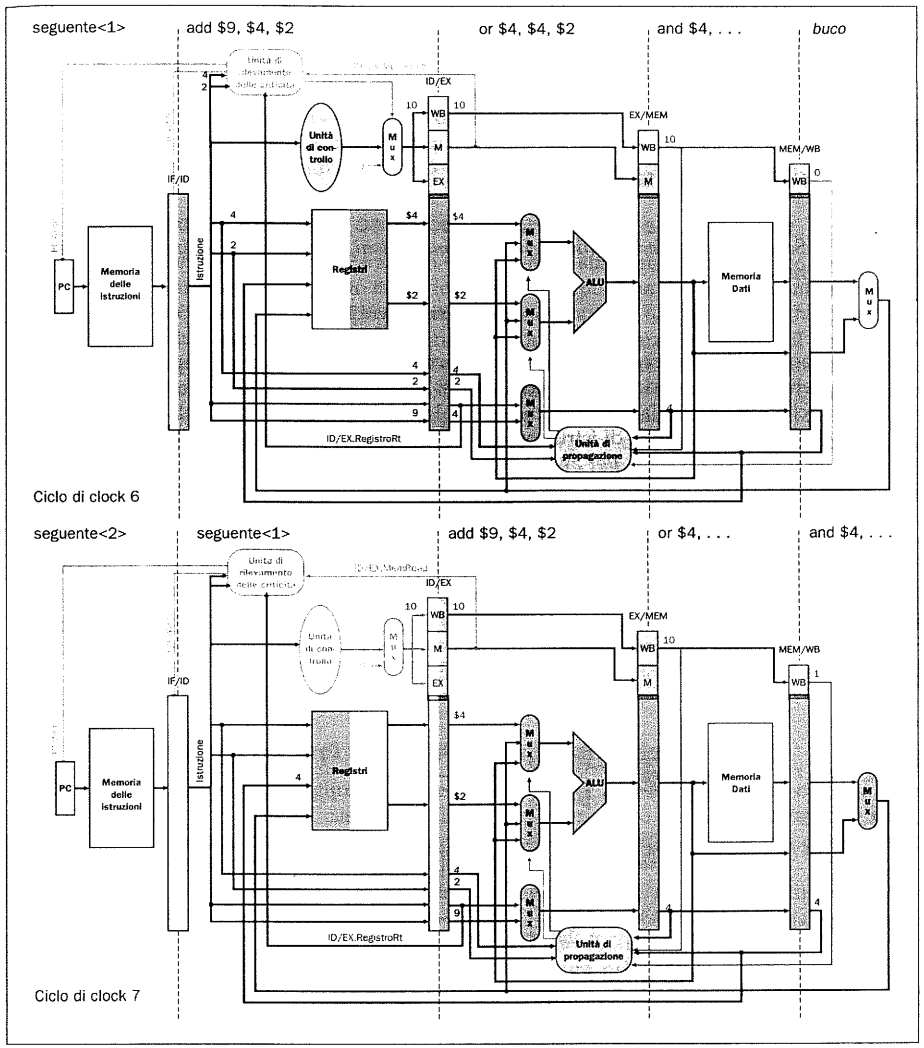


Figura 6.49 I cicli di clock 6 e 7 della sequenza di istruzioni dell'esempio riportato nel testo, modificata sostituendo l'istruzione `sub` **con una load.** Si noti che, contrariamente a quanto avviene nella figura 6.42, lo stallo permette all'istruzione `lw` di essere completata, e non vi è quindi propagazione da parte di MEM/WB nel ciclo di clock 6. Il registro `$4` per l'istruzione `add` nello stadio EX dipende ancora dal risultato dell'istruzione `or` in EX/MEM, così che l'unità di propagazione passa il risultato alla ALU. Le linee in grassetto mostrano le linee di ingresso della ALU attive nel colpo di clock, ed i numeri dei registri in corsivo indicano una criticità.

Approfondimento: A proposito della precedente osservazione riguardante l'azzeramento delle linee di controllo per evitare la scrittura nei registri o in memoria, si noti che soltanto i segnali RegWrite e MemWrite devono necessariamente essere forzati a 0, mentre gli altri segnali di controllo possono assumere un valore indifferente.

6.6 Criticità sui salti

Sono in migliaia ad accanirsi contro le ramificazioni del male, ma solo uno a colpirne la radice.

Henry David Thoreau, *Walden*, 1854

Finora ci si è preoccupati soltanto delle criticità che coinvolgono le operazioni aritmetiche ed i trasferimenti di dati. Come si è visto nel Paragrafo 6.1, tuttavia vi sono anche criticità della pipeline che coinvolgono i salti condizionati. La figura 6.50 mostra una sequenza di istruzioni ed indica il punto in cui all'interno della pipeline si verifica un salto. Per alimentare la pipeline è necessario caricare un'istruzione per ogni colpo di clock, eppure nell'attuale progetto le decisioni relative ai salti non vengono prese se non nello stadio MEM della pipeline. Come detto nel Paragrafo 6.1, tale ritardo nella determinazione dell'istruzione da caricare viene denominato *criticità sul controllo* o *criticità sui salti condizionati*, in contrapposizione con le *criticità sui dati* appena esaminate.

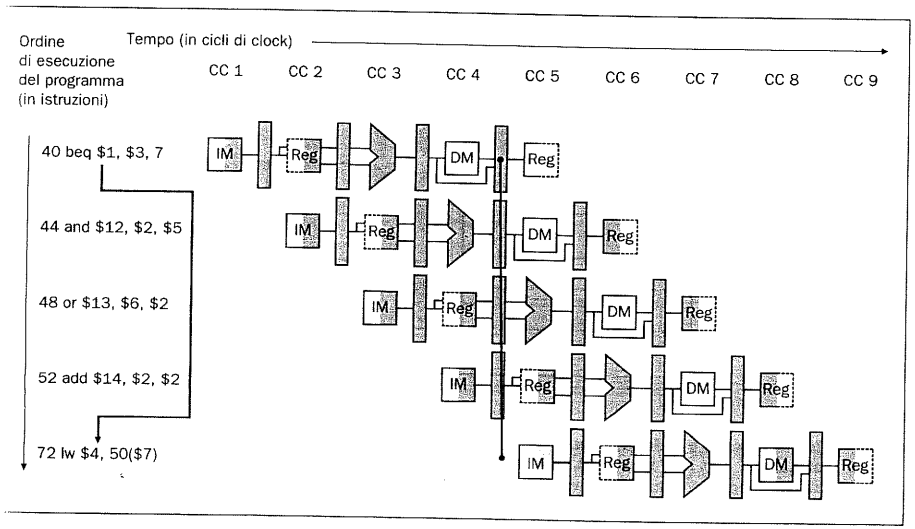


Figura 6.50 Effetti della pipeline sulle istruzioni di salto. I numeri a sinistra di ciascuna istruzione (40, 44, ...) sono gli indirizzi delle istruzioni. Poiché l'istruzione di salto decide se saltare nello stadio MEM (corrispondente al ciclo di clock 4 per l'istruzione beq mostrata sopra) le tre istruzioni che seguono quella di salto vengono caricate, e poi comincia l'esecuzione. Se non si intervenisse queste tre istruzioni inizierebbero a venire eseguite prima che beq salti a lw nella cella di indirizzo 72. La precedente figura 6.4 assumeva che esistesse dell'hardware aggiuntivo per ridurre la criticità sul controllo ad 1 ciclo di clock; la figura 6.50 usa l'unità di elaborazione non ottimizzata.

Questo paragrafo sulle criticità sul controllo è più breve dei paragrafi precedenti (relativi alle criticità sui dati) in quanto tali criticità sono relativamente semplici da capire, si verificano molto meno frequentemente delle criticità sui dati, e non vi è nulla che risulti così efficace per la risoluzione delle criticità sul controllo come lo è la propagazione nei confronti delle criticità sui dati. Per questo gli schemi utilizzati sono più semplici: si vedranno due schemi per la risoluzione delle criticità sul controllo ed un'ottimizzazione che permette di migliorare tali schemi.

Ipotizzare che il salto condizionato non sia eseguito

Come si è visto nel Paragrafo 6.1, restare in stallo sino a che il salto condizionato non è stato completato fa perdere troppo tempo. Una tecnica frequentemente usata per ottenere un comportamento migliore consiste nell'assumere che il salto non debba essere eseguito e nel continuare quindi l'esecuzione secondo il normale flusso di esecuzione delle istruzioni. Se poi il salto viene eseguito, allora le istruzioni che si stanno caricando e decodificando devono venire scartate e l'esecuzione riprende dall'istruzione corrispondente alla destinazione del salto. Se i salti non sono eseguiti nella metà dei casi, e se costa poco scartare le istruzioni, questa ottimizzazione dimezza il costo delle criticità sul controllo.

Per scartare le istruzioni ciò che si deve fare è semplicemente cambiare i valori dei segnali di controllo forzandoli a 0, sostanzialmente come si è fatto per ottenere lo stallo nel caso delle criticità sui dati legate alle istruzioni load. La differenza consiste nel fatto che si devono anche cambiare le tre istruzioni presenti negli stadi IF, ID ed EX quando il salto raggiunge lo stadio MEM; per le situazioni di stallo legate alle istruzioni load, si sono semplicemente messi a 0 i valori di controllo nello stadio ID e si è lasciato che questi si propagassero lungo la pipeline. Scartare delle istruzioni significa quindi essere in grado di annullare le istruzioni presenti negli stadi IF, ID ed EX della pipeline.

Ridurre i ritardi associati ai salti condizionati

Un modo per migliorare le prestazioni relative ai salti condizionati consiste nel ridurre il costo nel caso in cui il salto venga eseguito. Se si sposta l'esecuzione del salto più indietro nella pipeline si riduce il numero di istruzioni che devono essere scartate.

Finora si è assunto che il valore da caricare nel PC nel caso di salto condizionato venga selezionato nello stadio MEM. Si potrebbe risparmiare un ciclo di clock se invece si selezionasse l'indirizzo di salto già alla fine dello stadio EX: se si potesse spostare la decisione relativa al salto ancora più indietro nella pipeline, allora sarebbe necessario scartare una sola istruzione. Molte implementazioni del processore MIPS spostano allo stadio ID l'esecuzione del salto.

La parte facilmente realizzabile di questa modifica è quella relativa all'anticipazione del calcolo dell'indirizzo di salto. Il valore del PC ed il campo immediato sono già disponibili nel registro IF/ID della pipeline, e quindi è sufficiente spostare il sommatore per i salti dallo stadio MEM allo stadio ID.

La parte più difficile consiste invece nella decisione stessa relativa al salto condizionato. Se il salto è del tipo "salta se uguale" si devono confrontare i due registri letti durante lo stadio ID per verificare se sono uguali. Il test di uguaglianza può essere eseguito mettendo innanzitutto in OR esclusivo i bit corrispondenti, e poi mettendo in AND i risultati. Tale approccio è molto più veloce rispetto ad utilizzare

la ALU per eseguire la sottrazione e poi verificare se l'uscita è nulla, dal momento che non vi sono riporti nell'OR esclusivo. Spostando l'esecuzione del salto nello stadio ID vi è una sola istruzione da scartare nel caso in cui il salto debba venire eseguito, ossia quella che si trova nella fase di caricamento.

Per scartare l'istruzione nello stadio IF si aggiunge una linea di controllo, denominata IF.Scarta, che azzera il campo istruzione del registro IF/ID della pipeline. Azzerando il registro si trasforma l'istruzione appena caricata in una `nop`, ossia in un'istruzione che non esegue nessuna operazione in grado di cambiare lo stato del sistema (si veda l'approfondimento all'interno del Paragrafo 6.4). La figura 6.51 mostra l'unità di elaborazione opportunamente modificata.

Esempio

I salti all'interno della pipeline

Si mostri che cosa succede quando all'interno della seguente sequenza il salto condizionato viene eseguito, assumendo che la pipeline sia ottimizzata per i salti non eseguiti e che l'esecuzione del salto sia stata spostata nello stadio ID:

```
36  sub  $10, $4, $8
40  beq   $1, $3, 7 # PC-salto relativo: 40 + 4 + 7*4 = 72
44  and  $12, $2, $5
48  or   $13, $2, $6
52  add  $14, $4, $2
56  slt  $15, $6, $7
...
72  lw   $4, 50($7)
```

Soluzione

La figura 6.52 mostra ciò che succede quando viene eseguito il salto. A differenza della figura 6.50, nel caso il salto venga eseguito vi è un solo buco nella pipeline.

Previsione dinamica dei salti condizionati

Quando si assume che il salto condizionato non debba venire eseguito si realizza una forma rozza di *previsione dei salti* (*branch prediction*). In tal caso si prevede che il salto non venga eseguito, e si scarta il contenuto della pipeline nel caso di previsione errata. Come già detto nel Paragrafo 6.1, se si ha a disposizione dell'hardware addizionale è possibile sperimentare altri schemi di previsione dei salti.

Una possibilità consiste nel verificare l'indirizzo dell'istruzione per controllare se l'ultima volta che si era considerata quell'istruzione il salto fosse stato eseguito: in caso positivo, si caricano le istruzioni successive a partire dalla stessa posizione della volta precedente.

Un modo per implementare questa strategia consiste nell'utilizzare un *buffer di previsione dei salti* (*branch prediction buffer*), o *tabella di storia dei salti* (*branch history table*). Un buffer di previsione dei salti è una piccola memoria indicizzata attraverso la parte bassa dell'indirizzo dell'istruzione di salto. La memoria contiene un bit che indica se recentemente il salto è stato eseguito o meno.

Questa è la tipologia di buffer più semplice, in quanto non si sa se la previsione è quella giusta: ad esempio potrebbe essere relativa ad un altro salto caratterizzato

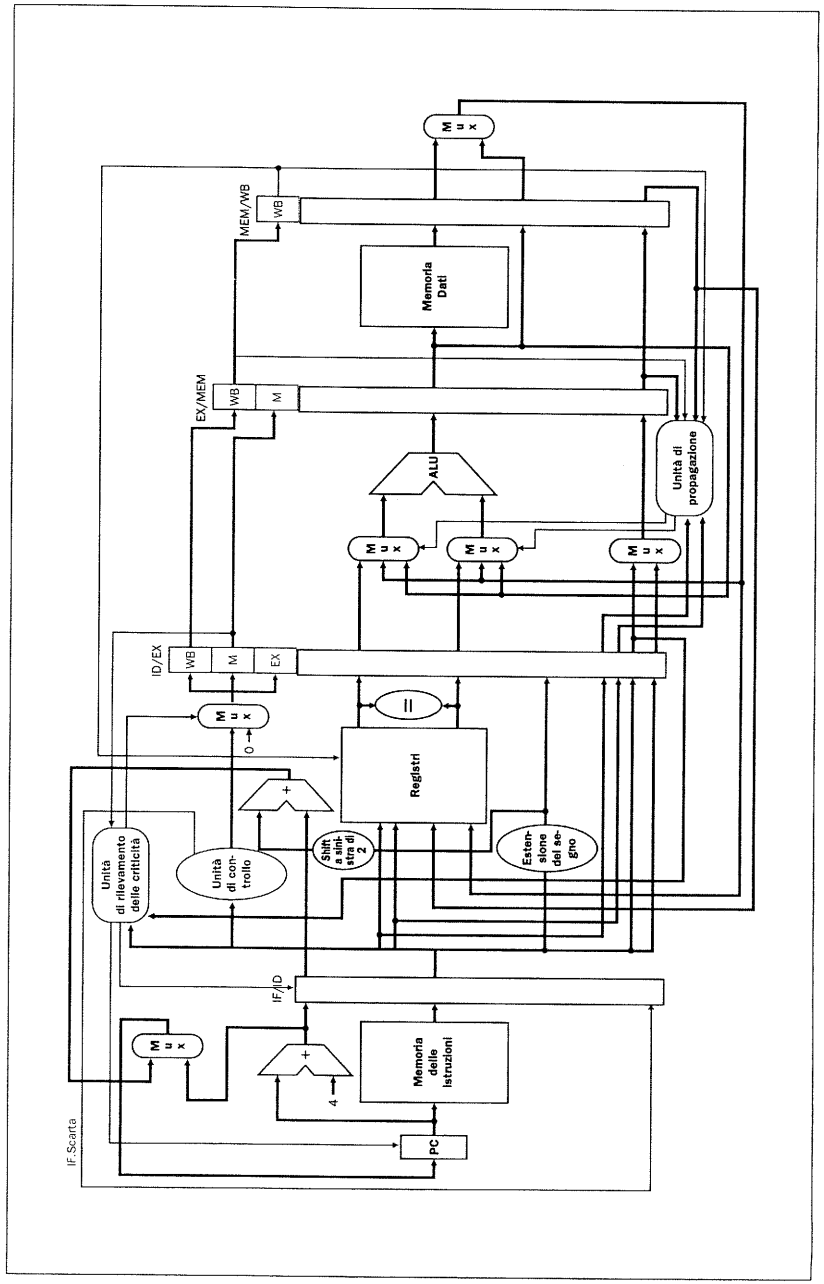


Figura 6.51 Unità di elaborazione per i salti, compreso l'hardware per scartare istruzioni successive al salto. L'ottimizzazione sposta la decisione relativa al salto dal quarto stadio della pipeline al secondo; solo un'istruzione tra quelle che seguono il salto si trova nella pipeline in quel momento. La linea di controllo IF.Scarta trasforma l'istruzione caricata in una `nop` azzerando il registro IF/ID della pipeline. Benché la linea per lo scarto compaia qui in uscita dall'unità di controllo, in realtà proviene dall'hardware che determina se il salto debba venire eseguito (etichettato con un segno di uguale a destra dei registri dello stadio ID).

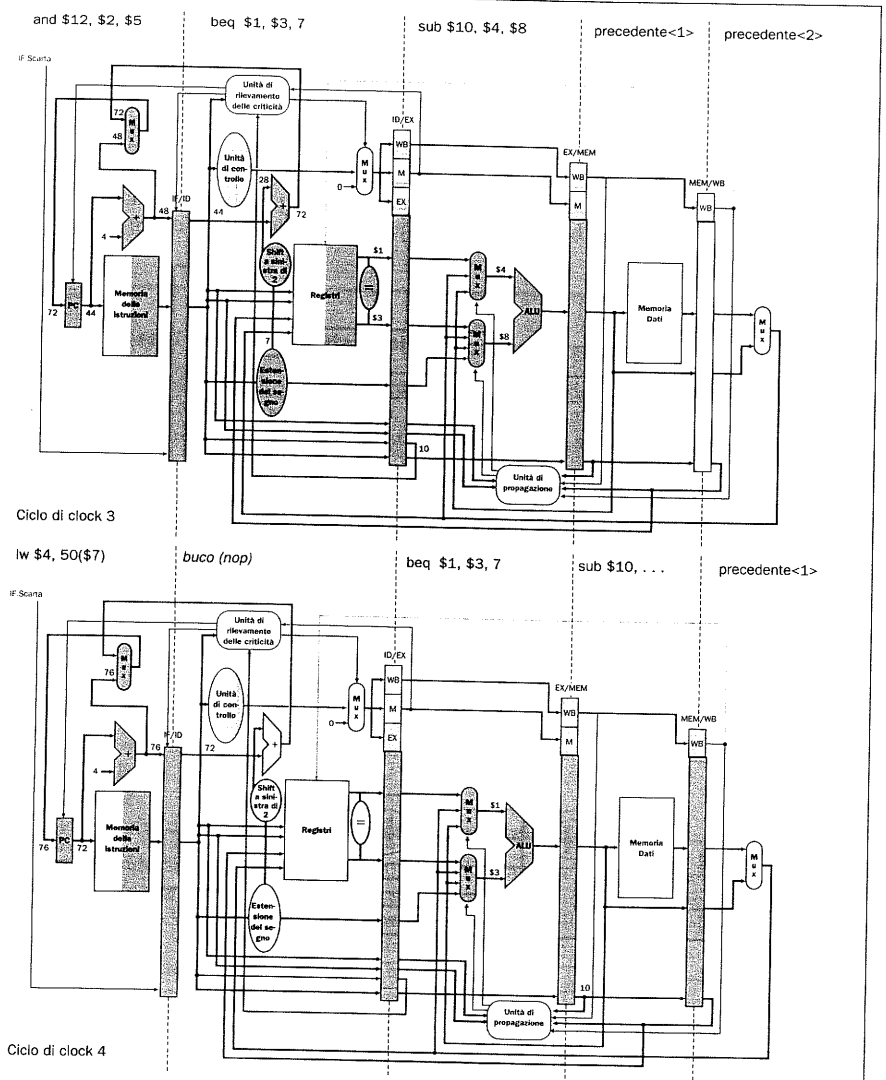


Figura 6.52 Lo stadio ID nel ciclo di clock 3 decide che il salto deve venire eseguito, quindi seleziona 72 come successivo indirizzo da caricare in PC e azzera l'istruzione caricata per il ciclo di clock successivo. Il ciclo di clock 4 mostra l'istruzione all'indirizzo 72 mentre viene caricata, nonché il buco corrispondente all'istruzione nop all'interno della pipeline, risultato dell'esecuzione del salto. Dal momento che nop corrisponde a or $0, $0, 0, si può discutere se lo stadio ID nel clock 4 debba venire evidenziato o meno.

dagli stessi bit per quanto riguarda la parte meno significativa. Tutto ciò però non ha effetto sulla correttezza dell'operazione: la predizione è infatti soltanto un suggerimento che si assume possa essere corretto, così che il caricamento delle istruzioni possa iniziare con l'indirizzo suggerito. Se il suggerimento si rivela sbagliato, il bit di previsione viene invertito e memorizzato con il nuovo valore, dopo di che si va ad eseguire la sequenza corretta di istruzioni.

Lo schema di previsione ad 1 bit ha un difetto in termini di prestazioni: anche se il salto fosse quasi sempre eseguito, si otterrebbe una previsione scorretta due volte (anziché una sola) quando il salto non è eseguito. L'esempio che segue illustra il problema.

Esempio

Cicli e previsioni

Si consideri un salto condizionato presente all'interno di un ciclo, e si supponga che il salto venga eseguito nove volte di seguito, e poi non venga eseguito per una volta. Qual'è l'accuratezza fornita dalla previsione per questo salto, assumendo che il bit di previsione rimanga nel buffer di previsione?

Soluzione

A regime la previsione non sarà corretta nella prima e nell'ultima iterazione del ciclo. L'errore sull'ultima iterazione è inevitabile in quanto la previsione dirà che il salto deve essere eseguito, essendo stato eseguito nove volte di seguito fino a quel punto. L'errore sulla prima iterazione si verifica perché il bit è stato complementato in occasione dell'esecuzione dell'ultima iterazione del ciclo, quando il salto non era stato eseguito. Di conseguenza l'accuratezza della previsione per questo salto (che viene eseguito nel 90% dei casi) è pari all'80% (due previsioni sbagliate e otto corrette).

In linea di principio si vorrebbe che per questi salti altamente regolari l'accuratezza della previsione corrispondesse alla frequenza di esecuzione del salto. Per rimediare a questa mancanza si usano spesso degli schemi di previsione a 2 bit. In uno schema a 2 bit la previsione deve essere sbagliata per due volte prima di essere modificata. La figura 6.53 mostra la macchina a stati finiti relativa a questo schema di previsione.

Un buffer di previsione dei salti può essere realizzato come un piccolo buffer accessibile tramite l'indirizzo dell'istruzione durante lo stadio IF della pipeline. Se si prevede che il salto debba venire eseguito il caricamento viene effettuato a partire dall'indirizzo destinazione non appena è noto il valore del PC; come detto in precedenza questo può avvenire già nello stadio ID, in caso contrario il caricamento e l'esecuzione in sequenza continuano. Se la previsione si rivela sbagliata i bit di previsione sono modificati come mostrato nella figura 6.53.

Approfondimento: Come si è detto nel Paragrafo 6.1, in una pipeline a 5 stadi è possibile rendere trattabili le criticità sul controllo tramite una ridefinizione dei salti. Un salto ritardato esegue sempre l'istruzione successiva, e solo la seconda istruzione dopo il salto subisce gli effetti del salto.

I compilatori e gli assemblatori cercano di mettere dopo il salto (nella posizione nota come *branch delay slot*) un'istruzione che deve sempre essere eseguita. Il compito del

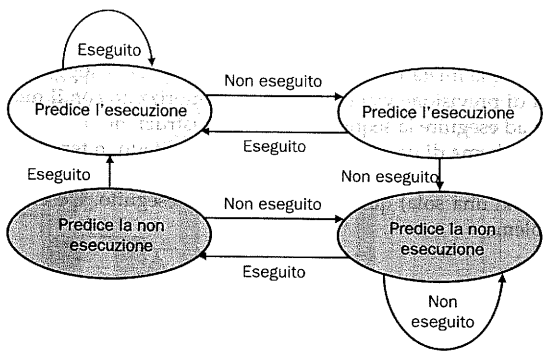


Figura 6.53 Gli stati possibili in uno schema di previsione a 2 bit. Utilizzando 2 bit anziché uno, un salto condizionato che privilegia la sua esecuzione o la sua non-esecuzione (come accade molto frequentemente) sarà oggetto di una previsione errata soltanto una volta. I 2 bit sono utilizzati per codificare i 4 stati del sistema.

software è quello di far sì che le istruzioni successive al salto siano valide ed utili. La figura 6.54 mostra i tre modi in cui può essere riempito il branch delay slot.

I limiti della tecnica del salto ritardato derivano da (1) le limitazioni sulle istruzioni che possono essere posizionate nei delay slot, e (2) la capacità di predire all'atto della compilazione quali siano le probabilità che un salto debba venire eseguito o meno.

La tecnica del salto ritardato viene utilizzata sempre di meno: mano a mano che gli elaboratori possiedono delle pipeline sempre più lunghe e sono in grado di terminare l'esecuzione di più istruzioni per ciascun colpo di clock (si veda il Paragrafo 6.8), la disponibilità di un solo delay slot non è di grande aiuto. D'altra parte le tecniche di previsione dinamica diventano tanto più utilizzate quanto più cresce il numero di transistor per chip.

Riepilogo sulle pipeline

Finora sono stati visti tre modelli di esecuzione: a ciclo singolo, a ciclo multiplo, con pipeline. Il controllo basato su pipeline mira a raggiungere l'obiettivo di eseguire una istruzione per colpo di clock, ma anche a permettere un clock più veloce, come nel caso del clock multiplo. Si riveda a questo proposito l'esempio del Paragrafo 5.3.

Esempio

Confronto tra le prestazioni dei diversi schemi di controllo

Si confrontino le prestazioni relative allo schema di controllo a ciclo singolo, a ciclo multiplo, e con pipeline utilizzando la combinazione di istruzioni di gcc, come da figura 4.54. I tempi di funzionamento per le principali unità funzionali nell'esempio sono pari a 2 ns per gli accessi in memoria, 2 ns per le operazioni della ALU, 1 ns per lettura e scrittura sul register file. Per l'esecuzione con pipeline si assuma che metà delle istruzioni di load sia immediatamente seguita da un'istruzione che ne utilizza il risultato, che il ritardo causato da un salto

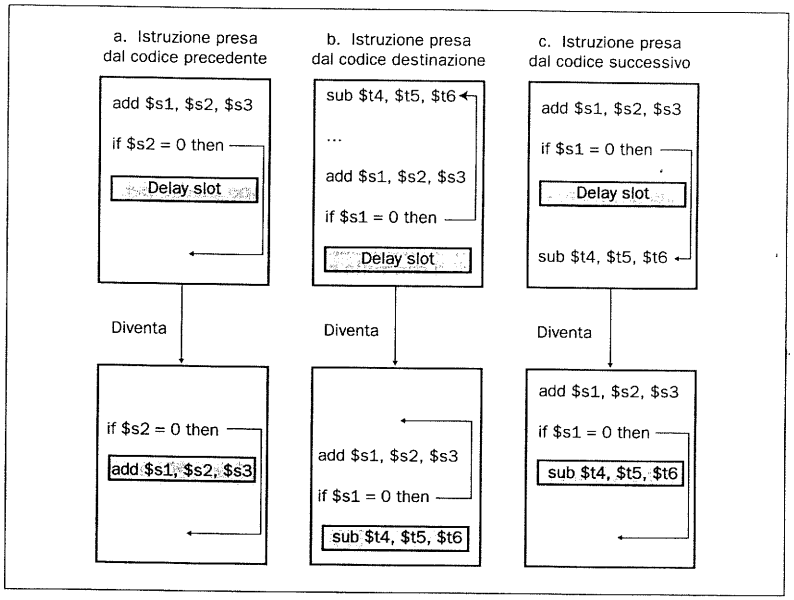


Figura 6.54 Utilizzo del branch delay slot. In ogni coppia il riquadro in alto mostra il codice prima della riorganizzazione, e quello in basso mostra il codice riorganizzato. In (a) il delay slot è utilizzato sistemandovi un'istruzione indipendente proveniente dalla parte di codice precedente il salto: questa è la scelta migliore. Le strategie (b) e (c) sono utilizzate quando (a) non è realizzabile. Nelle sequenze di codice (b) e (c) l'uso di `$s1` all'interno della condizione di salto rende impossibile lo spostamento dell'istruzione `add` nel branch delay slot. In (b) il branch delay slot è riempito con la destinazione del salto; normalmente l'istruzione destinazione deve essere copiata, poiché potrebbe essere raggiungibile attraverso un altro cammino. La strategia (b) è preferita quando il salto ha un'alta probabilità di venire eseguito, come nel salto all'interno di un ciclo. Infine il salto può venire riorganizzato utilizzando istruzioni provenienti dalla sequenza relativa al caso di non esecuzione, come in (c). Perché l'ottimizzazione mostrata sia accettabile bisogna che si possa eseguire l'istruzione `sub` quando il salto si comporta nel modo non previsto. Per accettabile si intende che il lavoro fatto divenga sì inutile, ma che il programma sia eseguito correttamente. Questo succede, ad esempio, se `$t4` è un registro temporaneo non utilizzato quando il salto si comporta nel modo non previsto.

nel caso di previsione errata sia pari ad 1 colpo di clock, e che un quarto dei salti siano oggetto di una previsione errata. Si assuma che i salti incondizionati comportino sempre una penalità pari ad un intero ciclo di clock di ritardo, così che il tempo medio da loro richiesto sia pari a 2 cicli di clock.

Soluzione

Le istruzioni presenti in gcc sono per il 22% load, per l'11% store, per il 49% operazioni di formato-R, per il 16% salti condizionati e per il 2% salti incondizionati (si veda la soluzione dell'esempio del Paragrafo 5.3). Nel progetto con pipeline le load richiedono 1 colpo di clock quando non vi è dipendenza sull'uso del registro letto, e 2 quando vi è dipendenza. Quindi il numero medio di cicli di clock per le istruzioni di load è pari a 1,5. Le istruzioni di store richiedono 1 ciclo di clock, ed altrettanto le istruzioni di formato-R. I salti condizionati

richiedono 1 ciclo di clock quando sono oggetto di una previsione corretta, e 2 nel caso contrario, così che il numero medio di colpi di clock per le istruzioni di salto è pari a 1,25. Il CPI per i salti incondizionati è 2. Il CPI medio è quindi

$$1{,}5 \times 22\% + 1 \times 11\% + 1 \times 49\% + 1{,}25 \times 16\% + 2 \times 2\% = 1{,}17$$

Moltiplicando 1,17 per i 2 ns corrispondenti al tempo del ciclo di clock di un elaboratore con pipeline per ottenere il tempo medio per istruzione in nanosecondi si ha il valore di 2,34 ns. Dagli esempi precedenti si ricava che il tempo medio per istruzione per i progetti a ciclo multiplo è pari a 4,04 ns, ed 8 ns per il ciclo singolo. Quindi il controllo con pipeline è 1,7 volte più veloce di quello a ciclo multiplo, e 3,4 volte più veloce di quello a ciclo singolo.

6.7 Le eccezioni

Far sì che un calcolatore dotato di gestione automatica delle interruzioni si comportasse [in modo sequenziale] non fu facile, dato che quando si verifica un'interruzione il numero di istruzioni nei diversi stadi di elaborazione può essere molto grande.

Fred Brooks Jr., *Planning a Computer System: Project Stretch*, 1962.

Un'altra forma di criticità sul controllo riguarda le eccezioni; si assuma ad esempio che la seguente istruzione

```
add   $1, $2, $1
```

generi un overflow aritmetico. È necessario trasferire immediatamente il controllo alla procedura di gestione dell'eccezione che si trova all'indirizzo $4000\ 0040_{esa}$, dal momento che non si vuole che questo valore non valido contamini altri registri o locazioni di memoria.

Analogamente a quanto fatto nel paragrafo precedente per i salti, è necessario eliminare dalla pipeline l'istruzione che segue la `add` ed iniziare il prelievo dell'istruzione successiva a partire dal nuovo indirizzo. Si userà lo stesso meccanismo utilizzato precedentemente nel caso dei salti, ma questa volta è l'eccezione a provocare l'azzeramento delle linee di controllo.

È già stato visto come eliminare un'istruzione nello stadio IF trasformandola in una `nop`. Per eliminare un'istruzione che si trova nello stadio ID si può utilizzare il multiplexer, già presente in questo stadio, per azzerare i segnali di controllo in caso di stallo. Un nuovo segnale di controllo, chiamato ID.Scarta, è messo in OR logico con il segnale di stallo che proviene dall'unità di rilevamento delle criticità per eseguire l'eliminazione durante lo stadio ID. Per eliminare l'istruzione nella fase EX si usa un nuovo segnale, chiamato EX.Scarta, che fa in modo che dei nuovi multiplexer azzerino le linee di controllo. Per iniziare il prelievo delle informazioni a partire dalla locazione $4000\ 0040_{esa}$ basta semplicemente aggiungere un altro ingresso al multiplexer del PC, in modo che invii il valore $4000\ 0040_{esa}$ al PC. La figura 6.55 mostra queste modifiche.

Questo esempio mostra un problema che si verifica nella gestione delle eccezioni: se non si interrompe l'esecuzione a metà dell'istruzione, il programmatore non

è più in grado di vedere il valore originale del registro `$1` che ha contribuito a generare la condizione di overflow, dal momento che tale registro è anche l'operando destinazione dell'istruzione `add`. Grazie ad un'accurata pianificazione, l'eccezione relativa all'overflow è riconosciuta durante lo stadio EX e quindi il segnale EX.Scarta può prevenire la scrittura del risultato, che avviene nello stadio WB.

L'ultimo passo consiste nel salvare l'indirizzo dell'istruzione che ha causato l'eccezione nell'Exception Program Counter (EPC), come già fatto nel Capitolo 5; in realtà viene salvato l'indirizzo + 4, e quindi la procedura di gestione delle eccezioni deve per prima cosa sottrarre 4 al valore memorizzato. La figura 6.55 mostra una versione schematica dell'unità di elaborazione, inclusi i circuiti per i salti condizionati e le modifiche necessarie alla gestione delle eccezioni.

Esempio

Le eccezioni in un calcolatore dotato di pipeline

Data la seguente sequenza di istruzioni,

```
40esa   sub   $11, $2, $4
44esa   and   $12, $2, $5
48esa   or    $13, $2, $6
4Cesa   add    $1, $2, $1
50esa   slt   $15, $6, $7
54esa   lw    $16, 50($7)
...
```

si assuma che le istruzioni a cui trasferire il controllo in caso di eccezione inizino nel modo seguente:

```
40000040esa   sw   $25, 1000($0)
40000044esa   sw   $26, 1004($0)
...
```

Mostrare che cosa avviene all'interno della pipeline se si verifica un'eccezione di tipo overflow nell'istruzione `add`.

Soluzione

La figura 6.56 mostra i vari eventi a partire dall'istruzione `add` nello stadio EX. L'overflow è riconosciuto durante questa fase, ed il valore $4000\ 0040_{esa}$ è forzato all'interno del PC. Il ciclo di clock 6 mostra che la `add` e le istruzioni successive sono scartate, e che viene prelevata la prima istruzione del codice di eccezione. Si noti che viene salvato l'indirizzo dell'istruzione *successiva* alla `add`: $4C_{esa} + 4 = 50_{esa}$.

Il Capitolo 5 elenca altre cause di eccezioni:

- Richieste da parte dei dispositivi di ingresso/uscita.
- Chiamata di un servizio del sistema operativo da parte di un programma utente.
- Utilizzo di un'istruzione non definita.
- Malfunzionamenti hardware.

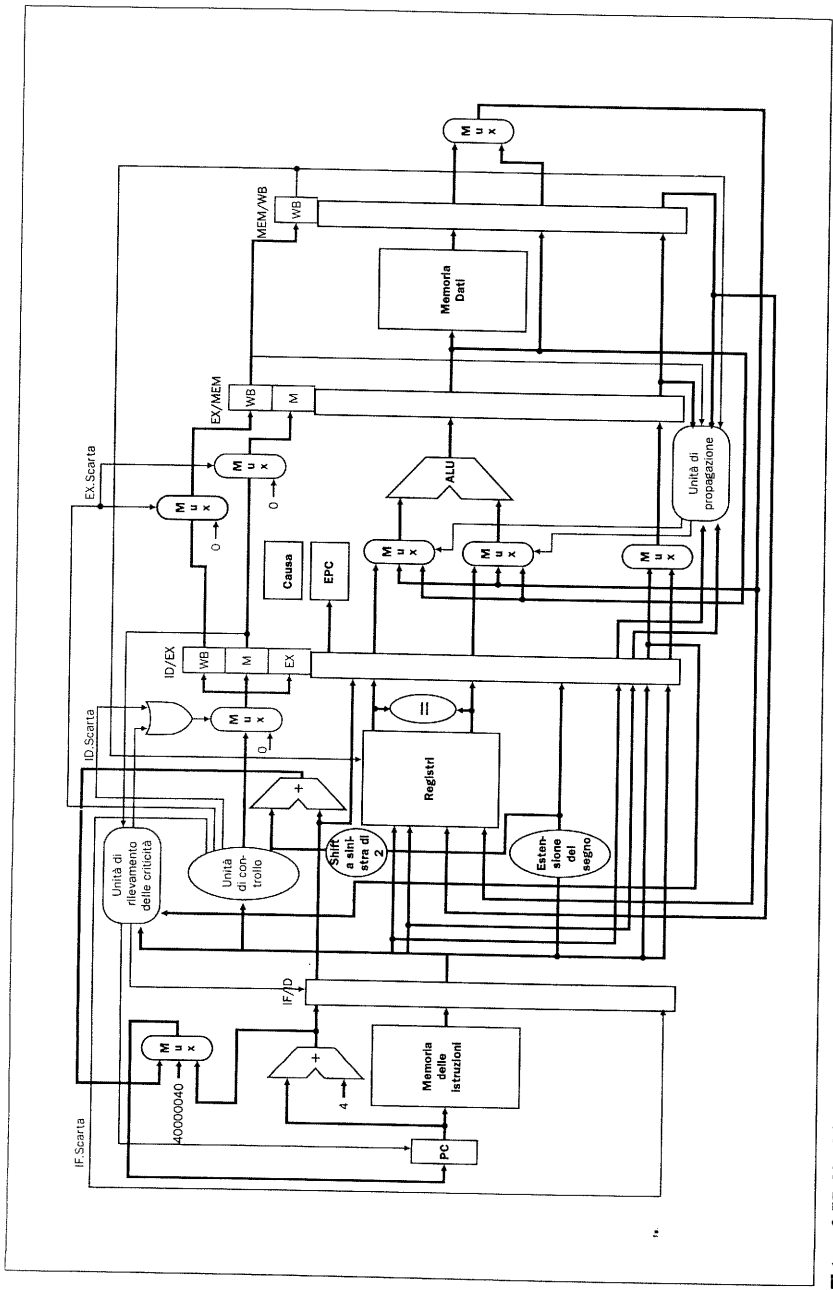


Figura 6.55 L'unità di elaborazione per la gestione delle eccezioni, con i relativi controlli. I cambiamenti rispetto alla figura 6.51 comprendono un nuovo ingresso, pari a $4000\ 0040_{esa}$, collegato al multiplexer che fornisce il nuovo valore al PC, un registro Causa che mantiene traccia del motivo per cui si è verificata l'eccezione, ed un registro Exception PC che salva l'indirizzo dell'istruzione che ha provocato l'eccezione. L'ingresso $4000\ 0040_{esa}$ del multiplexer rappresenta l'indirizzo da cui iniziare il prelievo delle istruzioni nel caso si verifichi un'eccezione. Anche se qui non è mostrato, il segnale di overflow della ALU è uno degli ingressi dell'unità di controllo.

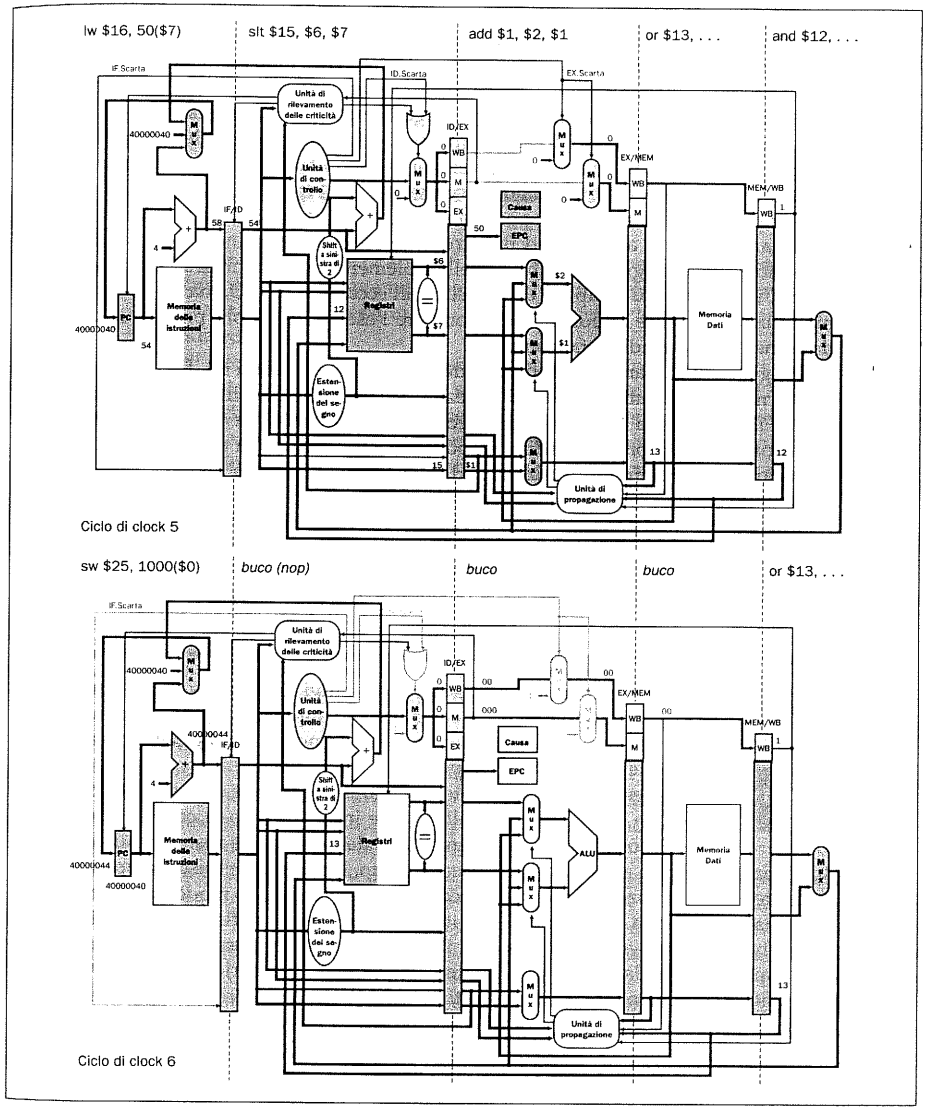


Figura 6.56 L'effetto di un'eccezione dovuta ad overflow aritmetico nell'istruzione di somma. L'overflow è riconosciuto durante lo stadio EX del ciclo di clock 5, provocando il salvataggio dell'indirizzo dell'istruzione che segue la somma nel registro EPC ($4C_{esa} + 4 = 50_{esa}$). L'overflow fa sì che tutti i segnali Scarta vengano attivati verso la fine di questo ciclo, azzerando i segnali di controllo dell'istruzione `add`. Il ciclo di clock 6 mostra le istruzioni convertite in buchi all'interno della pipeline ed il prelevamento della prima istruzione della procedura di gestione delle eccezioni (`sw $25, 1000($0)`) dalla locazione $4000\ 0040_{esa}$. Si noti che le istruzioni `and` e `or`, precedenti alla `add`, vengono completate. Anche se qui non è mostrato, il segnale di overflow della ALU è uno degli ingressi dell'unità di controllo.

Poiché si hanno cinque istruzioni attive in ogni ciclo di clock, il problema consiste nell'associare ogni eccezione all'istruzione che l'ha provocata; in più esiste la possibilità che nello stesso ciclo di clock più eccezioni si verifichino simultaneamente. La soluzione usata normalmente consiste nell'associare una priorità alle eccezioni, in modo da rendere facile determinare quale gestire per prima; questa strategia funziona anche per i calcolatori dotati di pipeline. Nella maggioranza delle implementazioni MIPS, i circuiti ordinano le eccezioni in modo tale che venga interrotta l'istruzione eseguita per prima fra quelle coinvolte.

Le richieste da parte dei dispositivi di ingresso/uscita e i malfunzionamenti hardware non sono associati ad un'istruzione specifica, e quindi in questi casi è consentita una maggiore flessibilità nell'implementazione. I circuiti possono scegliere l'istruzione più semplice da associare all'eccezione di I/O, ma nel caso di un malfunzionamento è opportuno interrompere l'esecuzione il più presto possibile, dato che l'hardware diventa instabile.

Il registro EPC mantiene l'indirizzo dell'istruzione interrotta ed il registro MIPS Causa memorizza tutte le possibili eccezioni che si verificano in un ciclo di clock; quindi il programma di gestione delle eccezioni deve trovare la corrispondenza fra istruzione ed eccezione. Un indizio importante è lo stadio della pipeline in cui può verificarsi un certo tipo di eccezione: ad esempio un'istruzione non definita è scoperta nello stadio ID, mentre una chiamata al sistema operativo avviene nello stadio EX. Le eccezioni sono collezionate nel registro Causa, in modo tale che i circuiti possano interrompere la normale esecuzione basandosi su eccezioni che si verificano dopo che eccezioni precedenti sono state risolte.

Interfaccia Hardware Software

Gli elaboratori ed i sistemi operativi devono lavorare in modo congiunto perché le eccezioni si comportino nel modo previsto. Il compito dell'hardware consiste normalmente nel bloccare a metà l'istruzione responsabile, nel far sì che le istruzioni precedenti siano completate, nell'eliminare tutte le istruzioni successive dalla pipeline, nell'inserire in un registro la causa dell'eccezione, nel salvare l'indirizzo dell'istruzione responsabile e nel trasferire il controllo ad un indirizzo preassegnato. Il compito del sistema operativo consiste nell'esaminare la causa dell'eccezione e nell'agire in modo appropriato. Per quanto riguarda le eccezioni dovute a istruzioni non definite, malfunzionamenti hardware o overflow aritmetico, il sistema operativo normalmente blocca il programma e segnala il motivo dell'interruzione. Nel caso di richieste di I/O o di chiamate al sistema operativo, quest'ultimo salva lo stato del programma, svolge il compito richiesto e quindi riprende l'esecuzione del programma.

La difficoltà nell'associare sempre correttamente un'eccezione all'istruzione che l'ha provocata nei calcolatori dotati di pipeline ha fatto sì che alcuni progettisti abbiano rilassato i requisiti nei casi non critici: si dice che questi calcolatori hanno *interruzioni imprecise* o *eccezioni imprecise*. Nell'esempio precedente in condizioni normali il PC conterrebbe il valore 58_{esa} all'inizio del ciclo di clock successivo al riconoscimento dell'eccezione, anche se l'istruzione che ha provocato l'eccezione si

trova all'indirizzo $4C_{esa}$. Un calcolatore con eccezioni imprecise metterebbe 58_{esa} nel registro EPC, lasciando al sistema operativo il compito di determinare quale istruzione abbia provocato l'eccezione. Il MIPS e la stragrande maggioranza dei calcolatori di oggi supportano *interruzioni precise* o *eccezioni precise* (una delle ragioni è dovuta alla gestione alla memoria virtuale, come si vedrà nel Capitolo 7).

6.8 Pipeline superscalari e dinamiche

Si avverte il lettore che i Paragrafi 6.8 e 6.9 contengono degli accenni ad argomenti affascinanti ma molto avanzati; per approfondirli si consiglia di consultare un libro più avanzato degli stessi autori, *Computer Architecture: A Quantitative Approach*, seconda edizione.

Nell'intento di realizzare processori sempre più veloci, le semplici pipeline descritte in questo capitolo sono evolute secondo tre direzioni principali. La prima è comunemente chiamata *superpipeline*, che significa semplicemente pipeline più lunga: dato che il massimo incremento di velocità di una pipeline ideale è legato al numero degli stadi, alcuni microprocessori recenti hanno pipeline con otto o più stadi.

Utilizzando l'analogia della lavanderia si può pensare di suddividere il processo di lavaggio tradizionale fra tre macchine separate, che si occupano rispettivamente di lavare, risciacquare e centrifugare; la pipeline passa quindi da quattro a sei stadi. Per ottenere il massimo aumento di velocità è necessario bilanciare nuovamente i restanti passi, in modo che abbiano la stessa durata (e questo vale per i processori come per il bucato).

La seconda tendenza consiste nel replicare i componenti interni del calcolatore in modo che sia possibile lanciare l'esecuzione di più istruzioni in ogni stadio della pipeline; questa tecnica in gergo informatico si chiama *superscalare*. Una lavanderia superscalare sostituirebbe la lavatrice e l'asciugatrice domestica con, ad esempio, tre lavatrici e tre asciugatrici; inoltre sarebbe necessario trovare altra mano d'opera per stirare e per riporre negli armadi una quantità di biancheria tre volte superiore nello stesso tempo. Lo svantaggio di questo approccio consiste nel lavoro addizionale necessario per mantenere impegnate tutte le macchine e per trasferire il carico di lavoro da uno stadio della pipeline al successivo.

Il fatto di lanciare più istruzioni nello stesso stadio fa sì che il tasso di esecuzione delle istruzioni sia maggiore della frequenza di clock o, detto in modo diverso, che il CPI sia minore di 1 (qualche buontempone ha persino invertito il nome della misura, chiamandola *IPC*, o *Instruction Per Clock*, ossia *numero di istruzioni per ciclo di clock*!). Quindi un microprocessore superscalare a quattro vie da 1000 MHz ha una velocità di esecuzione di picco pari a quattro miliardi di istruzioni al secondo, e nel caso migliore ha un CPI di 0,25. Gli attuali calcolatori superscalari cercano di far eseguire da due a sei istruzioni in ogni stadio della pipeline; se però le istruzioni all'interno del flusso hanno delle dipendenze o non soddisfano determinati criteri, soltanto le prime istruzioni vanno a buon fine (e in certi casi soltanto la prima istruzione).

La terza tendenza consiste nella *schedulazione dinamica della pipeline* mediante l'utilizzo di circuiti logici che evitano il verificarsi di condizioni critiche. Per quanto visto fino ad ora, la pipeline è in condizione di stallo quando aspetta che una criticità sia risolta, anche se le istruzioni successive sono già pronte. Ad esempio nella sequenza di codice

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

anche se `sub` e `slti` sono pronte per essere eseguite devono aspettare che `lw` ed `addu` siano state completate, il che può richiedere parecchi cicli di clock se la memoria è lenta (il Capitolo 7 spiega le cache e la ragione per cui a volte gli accessi alla memoria sono molto lenti). Le pipeline dinamiche sono normalmente affiancate da circuiti addizionali che fanno sì che istruzioni successive possano procedere in parallelo.

Il costo di questa soluzione consiste in un'unità di controllo delle pipeline molto più complessa ed in un modello dell'esecuzione delle istruzioni molto più complicato della semplice linea temporale riportata in figura 6.1.

MIPS superscalare

Come potrebbe essere fatto un calcolatore MIPS con implementazione superscalare? Si supponga che si vogliano eseguire due istruzioni per ciclo di clock, una delle quali potrebbe essere un'istruzione aritmetica o logica su numeri interi o un salto condizionato, e l'altra un'istruzione di trasferimento dati.

L'esecuzione di due istruzioni per ciclo di clock richiede il prelevamento e la decodifica di 64 bit di istruzioni. Per semplificare la fase di decodifica si può richiedere che le istruzioni siano accoppiate ed allineate a multipli di 64 bit, e che in ogni coppia di istruzioni la prima sia quella che esegue un'operazione con la ALU o un salto condizionato. In alternativa si potrebbero esaminare le istruzioni, ed eventualmente scambiarle prima che vengano inviate alla ALU o alla memoria; questo però introduce dei requisiti aggiuntivi per il rilevamento delle condizioni critiche. In entrambi i casi la seconda istruzione deve essere presa in considerazione solo se la prima può essere eseguita. La figura 6.57 mostra come appaiono le istruzioni nel momento in cui entrano a coppie nella pipeline; si ricordi che il circuito prende le decisioni in modo dinamico, eseguendo soltanto la prima istruzione se le condizioni non sono verificate.

Per eseguire in parallelo un'operazione logico-aritmetica ed una di trasferimento dati, il primo requisito hardware aggiuntivo (oltre alla solita logica di rilevamento delle condizioni critiche) riguarda la predisposizione di porte addizionali per il register file (vedere figura 6.58). In un solo ciclo di clock infatti l'operazione della ALU potrebbe dover leggere due registri, ed altrettanto potrebbe dover fare l'operazione di trasferimento dati; inoltre possono servire due porte in scrittura, una per

Tipo di istruzione	Stadi della pipeline							
Istruzione della ALU o salto condizionato	IF	ID	EX	MEM	WB			
Istruzione load o store	IF	ID	EX	MEM	WB			
Istruzione della ALU o salto condizionato		IF	ID	EX	MEM	WB		
Istruzione load o store		IF	ID	EX	MEM	WB		
Istruzione della ALU o salto condizionato			IF	ID	EX	MEM	WB	
Istruzione load o store			IF	ID	EX	MEM	WB	
Istruzione della ALU o salto condizionato				IF	ID	EX	MEM	WB
Istruzione load o store				IF	ID	EX	MEM	WB

Figura 6.57 Funzionamento di una pipeline superscalare. Le istruzioni della ALU e quelle di trasferimento dati sono eseguite contemporaneamente.

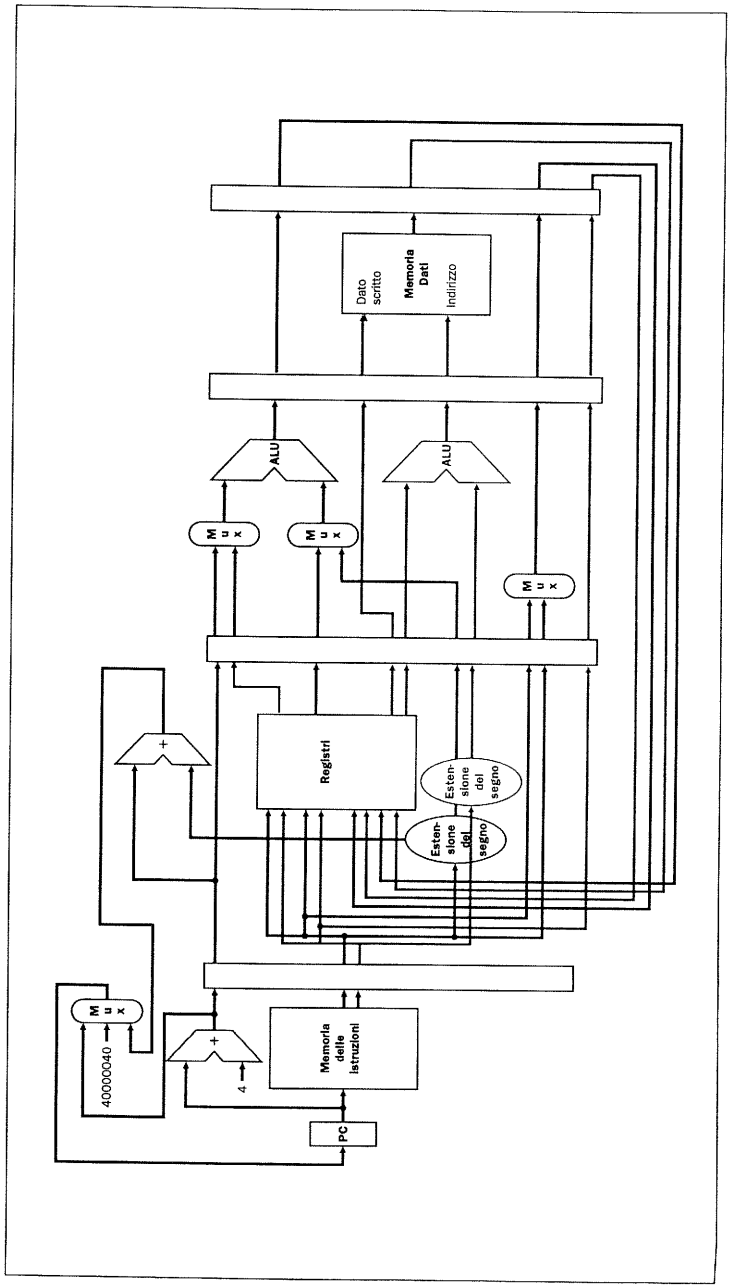


Figura 6.58 Unità di elaborazione superscalare. Sono evidenziate le aggiunte dovute all'implementazione superscalare: 32 bit addizionali dalla memoria delle istruzioni, due porte aggiuntive in lettura ed una in scrittura per il register file, ed un'altra ALU. Si assuma che la ALU in basso si occupi del calcolo degli indirizzi per il trasferimento dati, e che quella in alto si occupi di tutto il resto.

l'operazione della ALU e l'altra per l'operazione di trasferimento dati. Dato che la ALU è impegnata nell'esecuzione dell'operazione logica o aritmetica, occorre anche un sommatore separato per il calcolo dell'indirizzo per il trasferimento dati. Senza queste risorse addizionali il funzionamento della pipeline superscalare sarebbe ostacolato da criticità di tipo strutturale.

Un'altra difficoltà può limitare l'efficacia di una pipeline superscalare; nella semplice pipeline MIPS dell'esempio le istruzioni load hanno una latenza pari ad un ciclo di clock, per cui un'istruzione non può usare il risultato senza un'attesa. Nella pipeline superscalare il risultato di un'istruzione load non può essere usato nel *ciclo di clock* successivo; questo significa che le *due* istruzioni seguenti non possono usare il risultato della load senza un periodo di attesa. In un processore superscalare per sfruttare il parallelismo in modo efficace sono necessari compilatori più sofisticati o tecniche di schedulazione hardware, e tecniche più complesse per la decodifica delle istruzioni.

Esempio

Una semplice schedulazione superscalare del codice

Come sarebbe schedulato il seguente ciclo in una pipeline superscalare per il MIPS?

```
Ciclo: lw   $t0, 0($s1)        # $t0 = elemento di un vettore
       addu $t0, $t0, $s2      # somma con $s2
       lw   $t0, 0($s1)        # memorizzazione del risultato
       addi $s1, $s1, -4       # decremento del puntatore
       bne  $s1, $zero, Ciclo  # salto se $s1 != 0
```

Si riordinino le istruzioni in modo da evitare il più possibile condizioni di stallo della pipeline.

Soluzione

Le prime tre istruzioni, come le ultime due, hanno dati dipendenti fra loro; la figura 6.59 mostra la migliore schedulazione per queste istruzioni. Si noti che solo una coppia di istruzioni viene eseguita in modalità superscalare. Per ogni iterazione del ciclo occorrono 4 cicli di clock; con 4 cicli di clock per 5 istruzioni si ottiene un CPI pari a 0,8, decisamente deludente rispetto al caso più favorevole, cioè 0,5.

Una tecnica per ottenere prestazioni migliori da cicli che accedono a vettori è quella dell'*espansione dei cicli* (loop unrolling); come suggerisce il nome vengono inserite più copie del corpo del ciclo, e istruzioni appartenenti ad iterazioni diverse sono schedulate insieme.

	Istruzione della ALU o salto condizionato	Istruzione di trasferimento dati	Cicli di clock
Ciclo:		lw $t0, 0($s1)	1
	addi $s1, $s1, -4		2
	addu $t0, $t0, $s2		3
	bne $s1, $zero, Ciclo	sw $t0, 4($s1)	4

Figura 6.59 Il codice, come apparirebbe schedulato in un MIPS superscalare.

	Istruzione della ALU o salto condizionato	Istruzione di trasferimento dati	Cicli di clock
Ciclo:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
		lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 0($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1, $zero, Ciclo	sw $t3, 4($s1)	8

Figura 6.60 Il codice di figura 5.59 espanso, come apparirebbe schedulato in un MIPS superscalare. Dato che la prima coppia di istruzioni decrementa $s1 di 16, gli indirizzi delle istruzioni load sono rispettivamente il valore originale di $s1, il valore meno 4, meno 8 e meno

Esempio

Espansione di un ciclo per pipeline superscalari

Si mostri l'efficacia della tecnica dell'espansione dei cicli nel caso dell'esempio precedente; si assuma che l'indice del ciclo sia multiplo di quattro.

Soluzione

Per schedulare il ciclo in modo tale che non ci siano ritardi si scopre la necessità di fare quattro copie del corpo del ciclo. Dopo averlo espanso, il ciclo conterrà quattro copie delle istruzioni lw, addu e sw, più una addi e una bne. Il codice espanso e schedulato è mostrato in figura 6.60.

Si noti che 12 delle 14 istruzioni del ciclo sono eseguite in modalità superscalare. Occorrono 8 cicli di clock per quattro iterazioni del ciclo, cioè 2 colpi di clock per iterazione. L'espansione del ciclo e la schedulazione per l'esecuzione superscalare hanno portato a migliorare le prestazioni di un fattore pari a due, in parte per aver ridotto il numero di istruzioni di controllo del ciclo, ed in parte per via dell'esecuzione superscalare. Il costo di questo miglioramento consiste nell'uso di quattro registri temporanei invece di uno.

Quadro d'insieme

Sia le pipeline sia l'esecuzione superscalare aumentano il throughput di picco delle istruzioni. Pipeline più lunghe e versioni superscalari di maggiore ampiezza rendono il compito del compilatore ancora più complesso nel tentativo di sfruttare al massimo le potenzialità fornite dall'hardware. Purtroppo le dipendenze fra i dati ed i segnali di controllo all'interno dei programmi, insieme alla latenza delle istruzioni, costituiscono un limite superiore per le prestazioni effettive, dal momento che il processore talvolta è costretto ad aspettare per via di una dipendenza non ancora risolta o di un salto condizionato non previsto correttamente.

Sforzandosi di ottenere le migliori prestazioni, i progettisti dei circuiti devono inoltre assicurare la corretta esecuzione di tutte le sequenze di istruzioni. Agli autori dei compilatori può essere chiesto o no di limitare il tipo delle sequenze generate, ma in ogni caso essi devono sapere in che modo la pipeline ottiene le migliori prestazioni possibili per generare il codice di conseguenza.

Schedulazione dinamica della pipeline

La schedulazione dinamica della pipeline supera lo stallo cercando di eseguire istruzioni successive in attesa che lo stallo sia risolto. Tipicamente la pipeline è suddivisa in tre unità principali: un'unità per il prelevamento e l'invio dell'istruzione, un'unità di esecuzione e un'unità di consegna; la figura 6.61 mostra il modello corrispondente. La prima unità preleva le istruzioni, le decodifica e invia ciascuna di esse alla corrispondente unità funzionale dello stadio relativo all'esecuzione; possono esistere da 5 a 10 unità funzionali. Ciascuna unità funzionale ha dei buffer, chiamati *stazioni di prenotazione* (reservation stations) che mantengono operandi ed operazione. Non appena la stazione di prenotazione contiene tutti gli operandi richiesti e l'unità funzionale è pronta per l'esecuzione viene calcolato il risultato; a questo punto è compito dell'unità di consegna decidere se salvare il risultato nel register file o (nel caso di una store) in memoria.

Il modello base prevede la presenza di più macchine a stati che si occupano dell'esecuzione delle istruzioni: un'unità preleva e decodifica le istruzioni, diverse unità svolgono le operazioni richieste, ed un'unità decide quando le istruzioni sono completate e quando il risultato può essere consegnato. Per fare in modo che i programmi si comportino come se fossero in esecuzione su un semplice calcolatore senza pipeline, l'unità di prelevamento e decodifica deve trattare le istruzioni in ordine sequenziale, e quella di consegna deve scrivere i risultati nei registri o in memoria seguendo l'ordine di esecuzione del programma. Questo modello conservativo è detto *completamento in ordine*. Quindi nel caso si verifichi un'eccezione il calcolatore ha il riferimento all'ultima istruzione eseguita, e gli unici registri aggiornati sono quelli scritti da istruzioni che precedono l'evento eccezionale. Le unità funzionali, invece, sono libere di iniziare e finire l'esecuzione a piacere. Un approccio più radicale, che nel caso tipico introduce interruzioni imprecise, consiste nel consentire al calcolatore di fornire i risultati anche in ordine diverso rispetto a quello di esecuzione del programma; questo metodo è detto *completamento fuori ordine*.

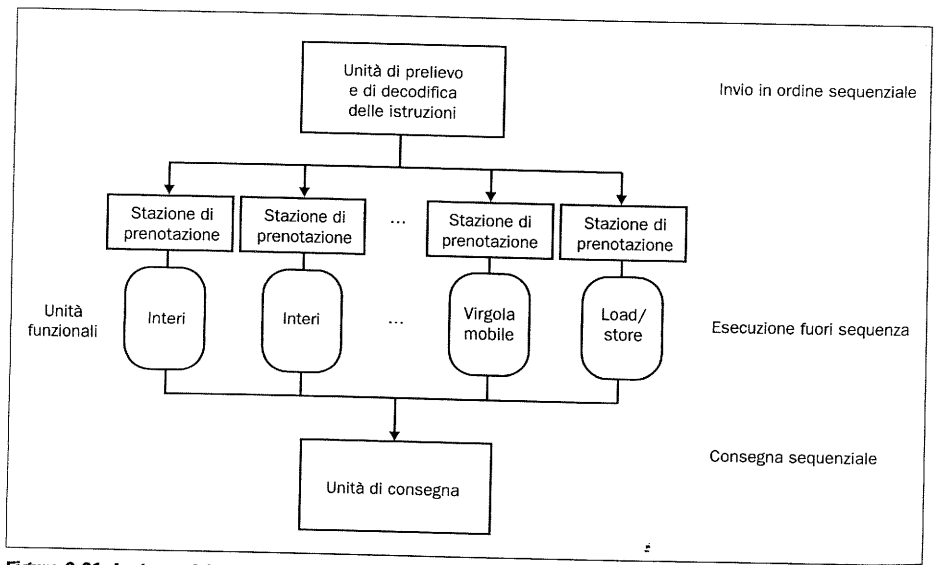


Figura 6.61 Le tre unità fondamentali di una pipeline con schedulazione dinamica.

Le pipeline a schedulazione dinamica sono più complesse delle pipeline tradizionali, o *pipeline statiche*, e trovare gli eventuali errori di progetto nei circuiti che le realizzano è una vera impresa. Parte della difficoltà è dovuta al fatto che la schedulazione dinamica è solitamente abbinata alla predizione dei salti, e quindi l'unità di consegna deve essere in grado di scartare tutti i risultati nelle unità di esecuzione dovuti ad istruzioni eseguite in seguito ad un salto erroneamente previsto. La combinazione della schedulazione dinamica e della predizione dei salti è detta *esecuzione speculativa*. Una seconda ragione che ne spiega la complessità sta nel fatto che la schedulazione dinamica è solitamente combinata con l'esecuzione superscalare, così che ciascuna unità può inviare o consegnare da quattro a sei istruzioni per ciclo di clock.

In sintesi i calcolatori dinamici predicono il flusso delle istruzioni, considerando le istruzioni a blocchi per capire quale verrà eseguita prima, ed eseguendo le istruzioni in base a congetture fondate sulla predizione e sulle dipendenze esistenti fra le istruzioni stesse. Le motivazioni alla base dell'esecuzione dinamica sono essenzialmente tre:

1. Nascondere la latenza della memoria, un problema chiave per i calcolatori, descritto nel Capitolo 7.
2. Evitare le situazioni di stallo che il compilatore non è in grado di risolvere, dovute a dipendenze potenziali fra istruzioni load e store.
3. Eseguire delle istruzioni, in base a congetture sul flusso futuro, in attesa che le situazioni critiche vengano risolte.

Un microprocessore che realizza tutte e tre le possibilità discusse (pipeline più lunga, superscalare e schedulata dinamicamente) è il DEC Alpha 21264. Questo dispositivo superscalare esegue quattro istruzioni per ciclo di clock (può però inserire fino a sei istruzioni nella pipeline), utilizza la modalità di esecuzione non sequenziale e la modalità di completamento ordinato. La pipeline è composta da nove stadi per le operazioni semplici su interi e su numeri in virgola mobile, consentendo al calcolatore di raggiungere una frequenza di clock pari a 600 MHz nel 1997.

Per mettere questo valore nella giusta prospettiva, si noti che nel 1997 la frequenza di clock del supercalcolatore Cray T-90 è solo 445 MHz; il Capitolo 2 ha mostrato che la frequenza di clock è solo uno dei parametri chiave che definiscono le prestazioni di un calcolatore, ma questo dato è ugualmente impressionante.

Approfondimento: L'unità di consegna controlla l'aggiornamento del register file e della memoria. Alcuni calcolatori con schedulazione dinamica aggiornano il register file direttamente durante la fase di esecuzione, mentre altri mantengono una copia del register file e svolgono l'aggiornamento vero e proprio del register file più avanti durante la fase di consegna dei risultati. Per quanto riguarda la memoria, di solito esiste un *buffer di memorizzazione*, chiamato anche *buffer delle scritture* (vedere Capitolo 7); l'unità di consegna consente la scrittura in memoria dal buffer quando sia l'indirizzo che il dato sono validi, e quando la memorizzazione non dipende più dalla predizione dei salti.

Approfondimento: Gli accessi alla memoria traggono beneficio dalle *cache non bloccanti*, che continuano a fornire accesso alla cache anche quando manca un dato (vedere Capitolo 7). I processori con esecuzione non sequenziale necessitano di cache non bloccanti per permettere l'esecuzione di istruzioni quando la cache non contiene il dato richiesto.

6.9 Un caso reale: le pipeline del PowerPC 604 e del Pentium Pro

Sia il PowerPC 604 che il Pentium Pro utilizzano pipeline schedulate dinamicamente; essi hanno un'organizzazione della pipeline talmente simile da poter essere rappresentata con lo stesso schema generale, riportato in figura 6.62. La figura 1.18 mostrava l'area di silicio richiesta dalla pipeline dinamica del Pentium Pro.

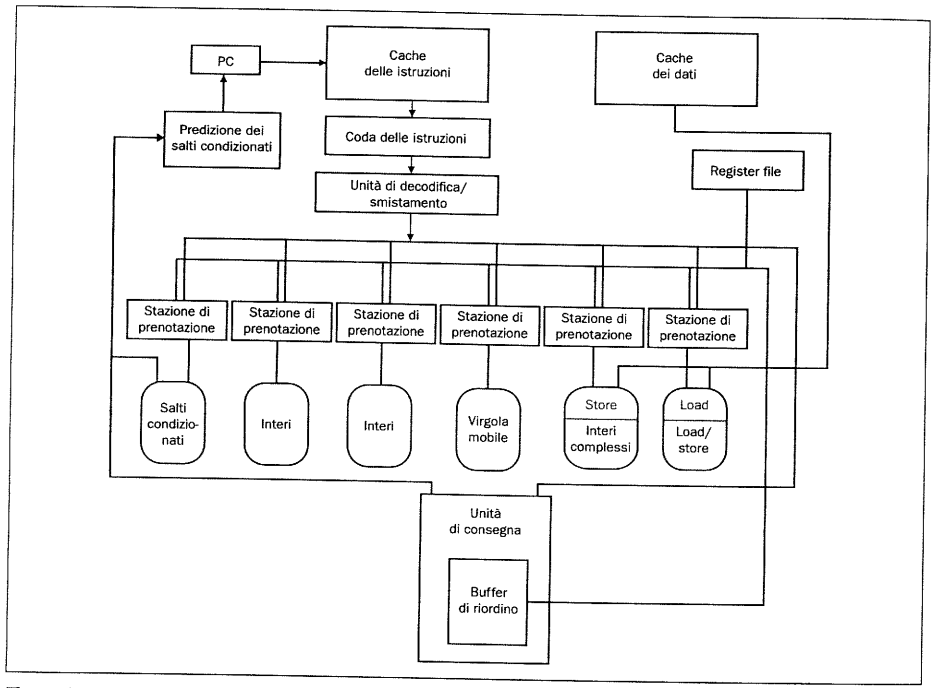


Figura 6.62 L'organizzazione generale della pipeline dell'Intel Pentium Pro e del PowerPC 604. Entrambe hanno sei unità funzionali, e le prime quattro svolgono gli stessi compiti. Nel Pentium Pro le ultime due unità funzionali sono quelle di load e di store (evidenziate in figura), mentre nel PowerPC 604 sono rispettivamente quelle per le operazioni complesse su interi e quelle per le operazioni di trasferimento dati. La figura mostra che ogni unità funzionale ha il proprio percorso che la collega ad una stazione di prenotazione, ma in realtà il Pentium Pro ha una sola stazione centrale di prenotazione che può essere usata da tutte le unità funzionali, attraverso un bus condiviso dall'unità per i salti e da una delle unità per le operazioni su interi, un bus condiviso dall'altra unità per gli interi e da quella per la virgola mobile, e un bus separato per le restanti unità funzionali. I nomi delle principali unità del Pentium Pro sono Unità di prelevamento/decodifica (Fetch/Decode Unit), Unità di smistamento/esecuzione (Dispatch/Execute Unit) e Unità di rientro (Retire Unit); le unità funzionali si chiamano Unità di esecuzione dei salti (Jump Execution Unit o EU), Unità di esecuzione per gli interi (Integer EU), Unità di esecuzione per la virgola mobile (Floating Point EU), Unità di generazione degli indirizzi di store (Store Address Generation Unit o AGU), e Unità di generazione degli indirizzi di load (Load AGU). I nomi reali delle principali unità del PowerPC sono Unità per le istruzioni (Instruction Unit), Unità funzionali (Functional Units) e Unità di completamento (Completion Unit); le unità funzionali si chiamano Unità di elaborazione dei salti (Branch Processing Unit), Unità a singolo ciclo per gli interi (Single-Cycle Integer Unit), Unità per la virgola mobile (Floating Point Unit), Unità multi-ciclo per gli interi (Multi-Cycle Integer Unit) e Unità di load / store (Load/Store Unit).

La cache delle istruzioni preleva 16 byte di istruzioni e le invia ad una coda di istruzioni; per il PowerPC questi byte rappresentano 4 istruzioni, mentre per il Pentium Pro sono un numero variabile. In seguito un certo numero di istruzioni sono prelevate e decodificate; per prevedere i salti ed eseguire le istruzioni successive (facendo le ipotesi opportune), entrambi i processori usano una tabella da 512 elementi contenente le informazioni sui salti eseguiti in tempi precedenti. L'unità di smistamento invia ciascuna istruzione, con i relativi operandi, alla stazione di prenotazione corrispondente all'unità funzionale prescelta, e in più inserisce un riferimento a quella istruzione all'interno del buffer di riordino dell'unità di consegna. Un'istruzione non può quindi essere considerata se non c'è spazio disponibile sia nell'appropriata stazione di prenotazione che nel buffer di riordino.

Con così tante istruzioni eseguite contemporaneamente è possibile esaurire lo spazio per mantenere i risultati: entrambi i processori hanno registri interni addizionali, chiamati *registri replicati* (rename registers o rename buffers), usati per memorizzare i risultati in attesa che l'unità di consegna dia il permesso per scrivere i registri effettivi. I registri replicati sono assegnati dall'unità di decodifica, riducendo quindi le criticità sul numero dei registri. L'istruzione viene eseguita quando il corrispondente elemento all'interno della stazione di prenotazione possiede tutti gli operandi, e la relativa unità funzionale è libera.

L'unità di consegna tiene traccia di tutte le istruzioni pendenti all'interno del buffer di riordino. Dato che entrambi i calcolatori usano la predizione dei salti, un'istruzione non è terminata fino a quando non lo decide l'unità di consegna; quando l'unità funzionale che si occupa dei salti decide se un salto deve o non deve essere effettuato, comunica questa informazione sia all'unità di predizione dei salti, che aggiorna lo stato del processore, sia all'unità di consegna, che decide la sorte delle istruzioni pendenti. Se la predizione è stata accurata, i risultati delle istruzioni che seguono il salto sono confermati e possono essere inseriti nei registri visibili all'utente e in memoria; in caso contrario i risultati sono invalidati e scartati dalle stazioni di prenotazione e dal buffer di riordino.

L'unità di consegna può fornire in uscita più istruzioni in un ciclo di clock; per garantire un comportamento corretto in caso si verifichino eccezioni, essa fa in modo che le istruzioni siano consegnate nello stesso ordine in cui sono state inserite nella pipeline. Tale unità quindi non può consegnare un'istruzione fino a che l'unità funzionale non ha finito l'operazione corrispondente, tutti i salti condizionati da cui dipende non sono stati risolti, e tutte le istruzioni precedenti non sono state consegnate.

La figura 6.63 elenca i parametri specifici delle pipeline del PowerPC 604 e del Pentium Pro. Molte differenze fra i due processori sono puramente di facciata; non sorprendente che la differenza principale sia relativa alla decodifica ed allo smistamento delle istruzioni. Come descritto nel Paragrafo 5.7, invece di cercare di inserire direttamente nella pipeline le istruzioni 80x86 di lunghezza variabile, l'unità di decodifica del Pentium Pro traduce le istruzioni Intel in micro-operazioni di lunghezza fissa pari a 72 bit, e manda queste ultime al buffer di riordino ed alle stazioni di prenotazione. Questo processo di traduzione richiede 1 ciclo di clock per determinare la lunghezza dell'istruzione 80x86, e 2 cicli aggiuntivi per creare le micro-operazioni.

Come discusso nel Capitolo 5, queste micro-operazioni hanno due registri sorgente ed un registro destinazione, e sono simili alle istruzioni MIPS. La maggior parte delle istruzioni 80x86 sono convertite in un numero di micro-operazioni variabile fra uno e quattro, ma le istruzioni 80x86 veramente complesse sono ese-

Nome del parametro	PowerPC 604	Pentium Pro
Massimo numero di istruzioni inserite per ciclo di clock	4	3
Massimo numero di istruzioni che completano l'esecuzione per ciclo di clock	6	5
Massimo numero di istruzioni consegnate per ciclo di clock	6	3
Numero di byte prelevati dalla cache delle istruzioni	16	16
Numero di byte nella coda delle istruzioni	32	32
Numero di istruzioni nel buffer di riordino	16	40
Numero di entrate nella tabella dei salti	512	512
Numero di bit di informazione sul passato per ciascuna entrata della tabella dei salti	2	4
Numero dei registri replicati	12 interi + 8 in virgola mobile	40
Numero totale di stazioni di prenotazione	12	20
Numero totale di unità funzionali	6	6
Numero di unità funzionali per gli interi	2	2
Numero di unità funzionali per operazioni complesse sugli interi	1	0
Numero di unità funzionali per i numeri in virgola mobile	1	1
Numero di unità funzionali per i salti condizionati	1	1
Numero di unità funzionali per la memoria	1 sia per load che per store	1 per load + 1 per store

Figura 6.63 I parametri specifici del PowerPC 604 e del Pentium Pro, in riferimento alla figura 6.62.

guite tramite un microprogramma convenzionale che contiene lunghe sequenze di micro-operazioni.

6.10 Errori e trabocchetti

Errore: realizzare le pipeline è facile.

Libri precedenti scritti dagli autori di questo testo testimoniano la difficoltà di costruire una pipeline: la prima edizione del libro rivolto agli specialisti conteneva un errore sulla pipeline, nonostante fosse stato rivisto da più di 100 persone ed usato sperimentalmente durante i corsi in otto università. L'errore saltò fuori soltanto quando qualcuno cercò di costruire il calcolatore descritto nel libro. Allo stesso modo la versione preliminare della prima edizione di questo libro conteneva un errore riguardante le istruzioni di propagazione e memorizzazione, errore sfuggito all'attenzione di molti fra revisori e studenti. Meglio stare molto attenti!

Errore: le pipeline possono essere implementate in modo indipendente dalla tecnologia.

Quando il numero di transistor per chip e la loro velocità facevano sì che la soluzione più conveniente per una pipeline fosse quella a cinque stadi, i salti ritardati (vedere gli approfondimenti dei Paragrafi 6.1 e 6.6) rappresentavano una soluzione semplice per controllare le criticità. Con pipeline più lunghe, esecuzione di tipo superscalare e predizione dinamica dei salti tale soluzione è ora ridondante. Nei primi anni '90, la schedulazione dinamica delle pipeline richiedeva troppe risorse, e non era un requisito per ottenere prestazioni elevate; oggi, con il numero di transistor che continua a raddoppiare e con il fatto che la logica è diventata più veloce

della memoria, unità funzionali multiple e pipeline dinamiche hanno molto più significato.

Trabocchetto: non tenere in adeguata considerazione il progetto del set di istruzioni può influire in modo negativo sulle pipeline.

Molte delle difficoltà nel progettare le pipeline nascono dalle complicazioni del set di istruzioni. Ecco alcuni esempi:

- Lunghezza e tempo di esecuzione fortemente variabili da istruzione a istruzione possono condurre a sbilanciamenti negli stadi della pipeline. Possono inoltre complicare anche di molto il riconoscimento delle criticità e la gestione precisa delle eccezioni.
- Modi di indirizzamento sofisticati possono condurre a diversi tipi di problemi. I modi di indirizzamento che aggiornano i registri, come l'indirizzamento di tipo update (vedere Paragrafo 3.12) complicano il riconoscimento delle criticità. Altri modi di indirizzamento che richiedono accessi multipli alla memoria complicano di molto l'unità di controllo della pipeline, e rendono difficile ottenere un flusso costante nella pipeline.

L'esempio migliore è forse quello del DEC Alpha e del DEC NVAX. Usando tecnologie confrontabili, il nuovo set di istruzioni Alpha consentì un'implementazione le cui prestazioni raddoppiavano in velocità rispetto a quelle del NVAX. In un altro esempio, Bhandarkar e Clark [1991] confrontarono il MIPS M/2000 ed il VAX 8700 contando i cicli di clock relativi ai benchmark SPEC; essi conclusero che, benché il MIPS eseguisse un maggior numero di istruzioni, in media il VAX eseguiva un numero di cicli di clock pari a 2,7 volte quelli del MIPS, e che quindi il MIPS era più veloce.

Errore: aumentare la profondità delle pipeline fa sempre crescere le prestazioni.

Esistono tre fattori che, combinati, possono limitare il miglioramento delle prestazioni ottenuto con l'utilizzo delle pipeline. In primo luogo le criticità sui dati del codice fanno sì che una profondità maggiore della pipeline renda l'esecuzione più lenta, dal momento che una percentuale maggiore di cicli di clock entra in condizione di stallo. Secondariamente, le criticità sui controlli fanno sì che una profondità maggiore della pipeline renda i salti più lenti da eseguire, aumentando il numero di cicli di clock del programma. Infine l'overhead dei registri di pipeline può limitare la riduzione del ciclo di clock dovuta alla maggiore profondità. La figura 6.64 mostra il rapporto fra profondità della pipeline e prestazioni nel caso di una pipeline per la virgola mobile.

6.11 Note conclusive

La saggezza consiste per nove decimi nell'essere saggi al momento giusto.

Proverbio popolare

Questo capitolo inizia parlando di una lavanderia, allo scopo di mostrare il principio alla base delle pipeline in un contesto quotidiano. Usando tale analogia come guida, è stato illustrato passo-passo come le istruzioni entrano nella pipeline, par-

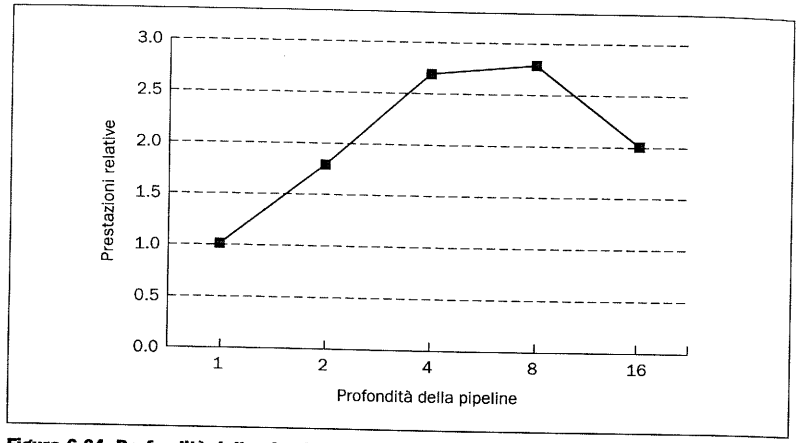


Figura 6.64 Profondità della pipeline in rapporto alle prestazioni ottenute. L'asse x mostra il numero di stadi nella parte EX della pipeline per la virgola mobile. Una pipeline ad un solo stadio corrisponde ad una logica a 32 livelli, che può essere appropriata per una singola operazione in virgola mobile. Questi dati si basano sulla Tabella 2 in S. R. Kunkell e J. E. Smith, "Optimal pipelining in supercomputers," *Proc. 13th Symposium on Computer Architecture* (June 1986), pag. 404-414.

tendo con il caso dell'unità di elaborazione a singolo ciclo e quindi aggiungendo i registri di pipeline, i cammini di propagazione, la rilevazione delle criticità sui dati, la predizione dei salti e l'eliminazione delle istruzioni dalla pipeline in caso di eccezione. La figura 6.65 mostra l'evoluzione finale di unità di elaborazione e unità di controllo.

L'utilizzo delle pipeline migliora il tempo medio di esecuzione delle istruzioni. A seconda che l'unità di controllo sia a singolo ciclo o multi-ciclo, si può pensare a questo come alla diminuzione della durata del ciclo di clock o alla diminuzione del numero di cicli di clock per istruzione (CPI). Avendo cominciato a trattare il caso di una semplice unità di elaborazione a singolo ciclo, la pipeline è stata introdotta come un modo per ridurre la durata del ciclo di clock dell'unità di elaborazione. La figura 6.66 mostra l'effetto della pipeline su CPI e frequenza di clock per ciascuna delle unità di elaborazione trattate nei Capitoli 5 e 6: la pipeline consente di ottenere sia un CPI inferiore che una frequenza di clock superiore.

L'utilizzo delle pipeline aumenta il throughput, ma non il tempo di esecuzione intrinseco delle istruzioni, o *latenza*; tale latenza ha un valore simile al caso dell'approccio multi-ciclo. A differenza di tale approccio, che utilizza ripetutamente lo stesso hardware durante l'esecuzione delle istruzioni, la pipeline inizia l'esecuzione di un'istruzione ad ogni ciclo di clock utilizzando dei circuiti dedicati. La figura 6.67 mostra le unità di elaborazione della figura 6.66 disposte a seconda del grado di condivisione dei circuiti e della latenza delle istruzioni.

Il diverso tempo di esecuzione delle istruzioni genera criticità a causa delle dipendenze delle istruzioni all'interno di un programma, dal momento che l'esistenza di una situazione di dipendenza costringe il calcolatore ad attendere per un periodo pari all'intera latenza dell'istruzione prima di risolvere la criticità. Il costo dovuto alle dipendenze fra i dati può essere ridotto con l'utilizzo di circuiti di propagazione, e la frequenza delle dipendenze sui segnali di controllo può migliorare attraverso la predizione dei salti e la schedulazione fatta dal compilatore.

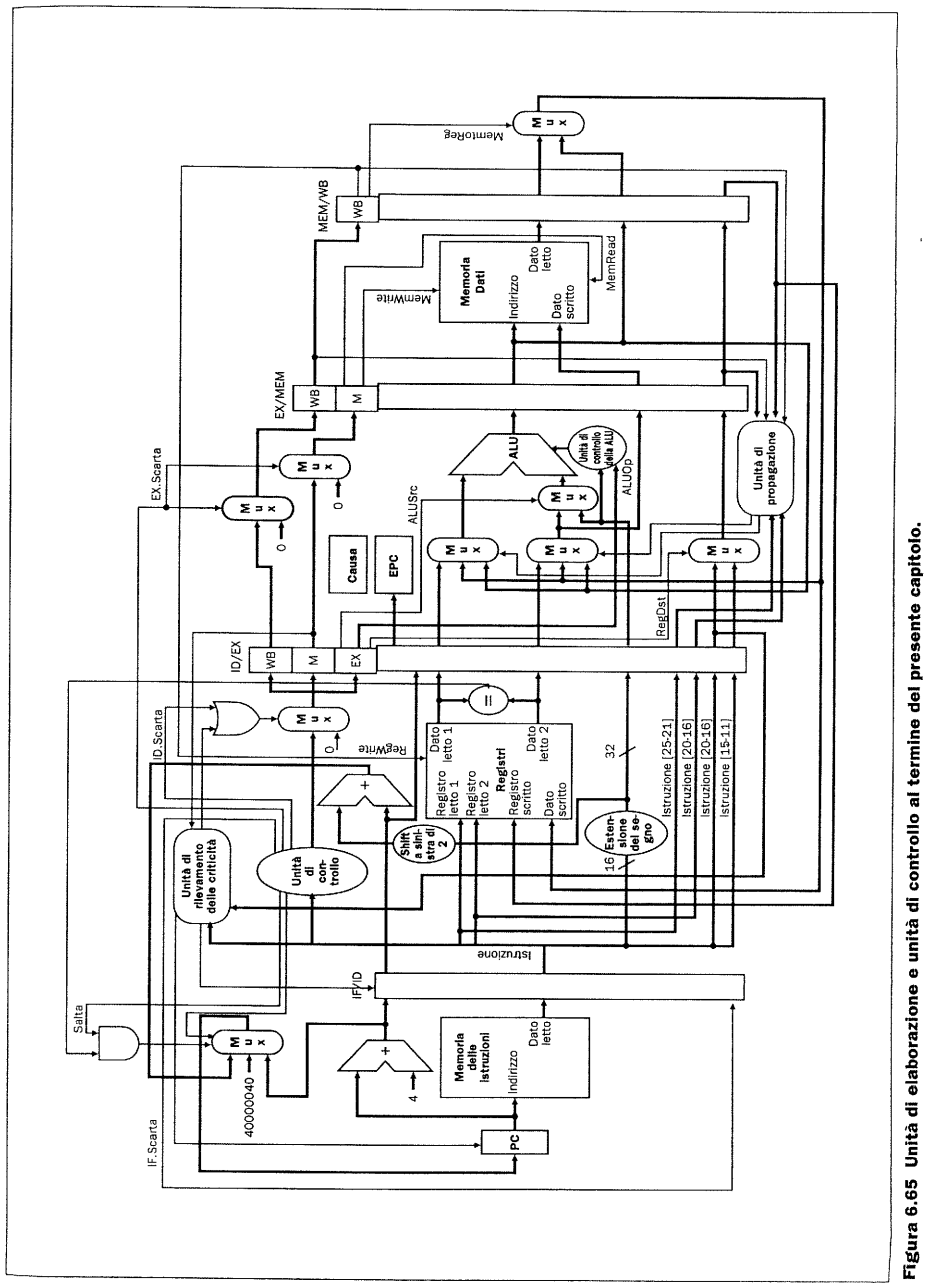


Figura 6.65 Unità di elaborazione e unità di controllo al termine del presente capitolo.

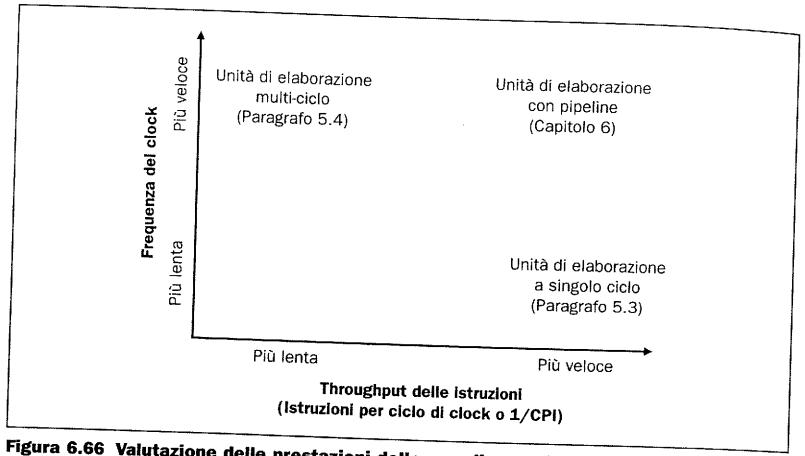


Figura 6.66 Valutazione delle prestazioni della semplice unità di elaborazione (a singolo ciclo), dell'unità di elaborazione multi-ciclo del Capitolo 5 e del modello di esecuzione con pipeline del Capitolo 6. Benché il numero di istruzioni per ciclo di clock (throughput delle istruzioni) sia leggermente maggiore nel caso di unità di controllo a singolo ciclo, l'unità di elaborazione con pipeline si avvicina a questo valore e raggiunge una frequenza di clock pari a quella dell'unità di elaborazione multi-ciclo.

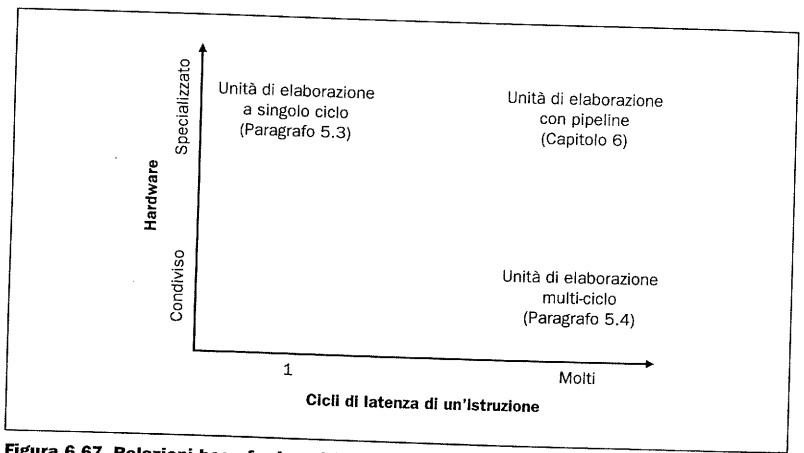


Figura 6.67 Relazioni base fra le unità di elaborazione di figura 6.66. L'unità di elaborazione con pipeline figura nella categoria con latenza delle istruzioni pari a quella dell'unità di elaborazione multi-ciclo, dal momento che il tempo di esecuzione della singola istruzione non è inferiore: è il throughput delle istruzioni ad essere migliore.

La recente introduzione di pipeline più lunghe, superscalari e con schedulazione dinamica ha contribuito ad un miglioramento annuale delle prestazioni pari al 60% sin dal 1986. Nel passato sembrava che l'unica scelta fosse fra i processori ad elevata frequenza di clock e i più sofisticati processori superscalari. L'Alpha 21264, che lancia sei istruzioni per ciclo di clock, ha modalità di esecuzione fuori sequenza e nel 1997 raggiungeva la frequenza di clock di 600 MHz, è la dimostrazione che si

possono ottenere entrambe le cose. Non sorprende che questo chip da 15 milioni di transistor avesse le migliori prestazioni con il benchmark SPEC95.

Avendo ottenuto questi straordinari risultati per quanto riguarda l'elaborazione, la legge di Amdahl suggerisce che un'altra parte del sistema diventerà il collo di bottiglia; il collo di bottiglia in questione è l'argomento del prossimo capitolo: la memoria di sistema.

Un'alternativa alla spasmodica ricerca di aumentare il parallelismo all'interno di un singolo processore consiste nel provare la soluzione multiprocessore, che sfrutta un parallelismo a livello più macroscopico. L'elaborazione parallela è l'argomento del Capitolo 9.

6.12 Inquadramento storico e approfondimenti

supercomputer: Qualunque calcolatore disegnato solo sulla lavagna.

Stan Kelly-Bootle, *The Devil's DP Dictionary*, 1981.

Questo paragrafo descrive alcuni importanti sviluppi nella progettazione delle pipeline. È riconosciuto quasi universalmente che il primo calcolatore general-purpose dotato di pipeline fu Stretch, l'IBM 7030 (figura 6.68). Stretch fu rilasciato dopo l'IBM 704, con l'obiettivo di essere 100 volte più veloce del suo predecessore; tale obiettivo rappresentava un grande sforzo (in inglese "stretch") rispetto allo stato dell'arte di quegli anni, e da qui deriva il soprannome del calcolatore. Il progetto prevedeva di ottenere un fattore di miglioramento pari a 1,6 sovrapponendo le fasi di prelevamento, decodifica ed esecuzione delle istruzioni grazie ad una pipeline a quattro stadi; il resto ovviamente doveva essere ottenuto con hardware addizionale e logica più veloce. Stretch inoltre fu la palestra di addestramento sia dei progettisti dell'IBM 360, Gerrit Blaauw e Fred Brooks Jr., che del progettista dell'IBM RS/3600, John Cocke.

Figura 6.68 Il calcolatore Stretch, uno dei primi dotati di pipeline. Fotografia gentilmente concessa da IBM.

Figura 6.69 Il CDC 6600, il primo supercomputer. Fotografia gentilmente concessa dall'Istituto Charles Babbage, University of Minnesota.

La Control Data Corporation (CDC) nel 1964 rilasciò quello che è considerato il primo supercomputer, il CDC 6600 (vedere figura 6.69). Le istruzioni base dei successivi calcolatori Cray hanno molta somiglianza con quelle del CDC 6600 originale. Il CDC 6600 era unico sotto molti aspetti; l'interazione fra il progetto della pipeline e quello del set di istruzioni era stata ben compresa, ed il set di istruzioni era volutamente semplice per rendere più efficiente l'utilizzo della pipeline. Il CDC 6600 usava inoltre una tecnologia avanzata di realizzazione di chip (packaging). Il libro di Thornton [1970] fornisce un'eccellente descrizione dell'intero elaboratore, dalla tecnologia all'architettura, e comprende un'introduzione di Seymour Cray (sfortunatamente il libro è fuori stampa). Sia Cray che Thornton hanno vinto l'ACM Eckert-Mauchly Award.

L'IBM 360/91 introdusse molti concetti nuovi, compreso il rilevamento dinamico delle criticità, la propagazione generalizzata e le stazioni di prenotazione (vedere figura 6.70). L'approccio seguito è usualmente definito *algoritmo di Tomasulo*, dal

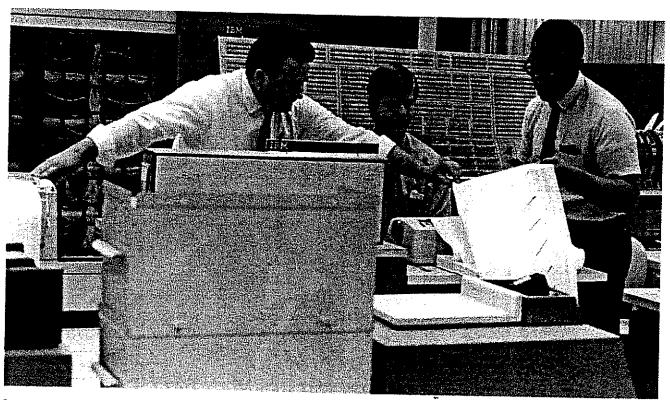


Figura 6.70 L'IBM 360/91, rilasciato nel 1966, diede una svolta decisiva allo stato dell'arte dell'esecuzione delle istruzioni in pipeline. Fotografia gentilmente concessa dall'IBM.

nome di un ingegnere che lavorò al progetto. Il gruppo che creò il 360/91 era guidato da Michael Flynn, che nel 1992 ottenne l'ACM Eckert-Mauchly Award anche per il suo contributo sull'IBM 360/91; nel 1997 lo stesso premio andò a Robert Tomasulo per il suo lavoro pionieristico sull'elaborazione non sequenziale.

L'organizzazione interna dell'IBM 360/91 ha molto in comune con il PowerPC 604 e con il Pentium Pro. Una differenza sostanziale è che non era implementata la predizione dei salti, e quindi neanche l'esecuzione speculativa; un'altra importante differenza era la mancanza dell'unità di consegna, e quindi quando un'istruzione terminava aggiornava direttamente i registri. La consegna non sequenziale delle istruzioni rendeva le *interruzioni imprecise*, il che dimostrò essere poco apprezzato e fece sì che da quel momento in poi negli elaboratori con pipeline a schedulazione dinamica fossero introdotte le unità di consegna.

Nei primi anni '80 gli elaboratori RISC resero più raffinata la nozione di pipeline schedulata dal compilatore. I concetti di salti ritardati e di caricamento ritardato, comuni nella microprogrammazione, furono estesi alle architetture ad alto livello; infatti il calcolatore Stanford, che pose le basi all'architettura commerciale MIPS, fu chiamato Microprocessore senza Stadi di Pipeline Concatenati (Microprocessor without Interlocked Pipeline Stages) perché risolvere le criticità sui dati era compito dell'assemblatore o del compilatore.

Il lavoro dell'IBM fu pionieristico in molti campi. Negli anni '70 fu lanciato un progetto chiamato ACS, che comprendeva concetti di esecuzione multipla di istruzioni, ma che non raggiunse mai lo stadio di produzione. La prima proposta di un processore superscalare che prendesse decisioni dinamiche sull'esecuzione delle istruzioni fu di John Cocke; egli descrisse le idee di base durante molti seminari a metà degli anni '80, ed inventò il termine *superscalare*. Il progetto originale fu chiamato America. L'architettura IBM Power-1, usata nella linea RS/6000, si basa su queste idee, e il PowerPC è una variante dell'architettura Power-1. Cocke vinse il Turing Award per questo suo lavoro di progetto, il più alto riconoscimento nel campo dell'informatica e dell'ingegneria.

Un approccio precedente a quello superscalare che si basa su una tecnologia di compilazione dello stesso tipo è chiamato *parola di istruzione lunga* (LIW, long instruction word) o talvolta *parola di istruzione molto lunga* (VLIW, very long instruction word). In questo approccio, come nel caso superscalare, durante ogni ciclo di clock viene iniziata l'esecuzione di più istruzioni, ma in LIW il compilatore garantisce che non esistano dipendenze fra le istruzioni da eseguire nello stesso periodo e che ci siano risorse hardware sufficienti per eseguirle, semplificando quindi la logica di decodifica ed esecuzione delle istruzioni. Un vantaggio molto pratico dell'approccio superscalare rispetto ai progetti LIW è che i processori superscalari possono eseguire programmi senza modificare il programma in codice macchina che funziona sulle architetture più tradizionali; l'approccio LIW funziona bene quando è disponibile il codice sorgente dei programmi ed è possibile ricompilarlo.

Letture consigliate

Bhandarkar, D., e D. W. Clark [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (Aprile), Palo Alto, 310-19.

Un confronto quantitativo fra RISC e CISC scritto da specialisti che sostennero la superiorità dei CISC, e che li progettarono; la loro conclusione è che il MIPS è da 2 a 4 volte più veloce di un VAX costruito con tecnologia simile, con una media di 2,7.

Hennessy, J. L., e D. A. Patterson [1996]. *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann, San Francisco.
I Capitoli 4 e 5 entrano molto più in dettaglio sui calcolatori con pipeline, inclusi i concetti di schedulazione dinamica dell'hardware e degli elaboratori superscalari.

Jouppi, N. P., e D. W. Wall [1989]. "Available instruction-level parallelism for superscalar and superpipelined machines," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (Aprile), Boston, 272-82.
Un confronto fra sistemi superscalari e con superpipeline.

Kogge, P. M. [1981]. *The Architecture of Pipelined Computers*, Mc-Graw Hill, New York.
Un testo chiaro sul controllo delle pipeline, con enfasi sui principi fondamentali.

Russel, R. M. [1978]. "The CRAY-1 computer system," *Comm. of the ACM* 21:1 (Gennaio), 63-72.
Una breve presentazione di un calcolatore classico, che usa vettori di operazioni per eliminare le condizioni di stallo della pipeline.

Smith, A., e J. Lee [1984]. "Branch prediction strategies and branch target buffer design," *Computer* 17:1 (Gennaio), 6-22.
Una delle prime rassegne sulla predizione dei salti.

Smith, J. E., e A. R. Plezkun [1988]. "Implementing precise interrupts in pipelined processors," *IEEE Trans. on Computers* 37:5 (Maggio), 562-73.
Tratta delle difficoltà nel gestire le interruzioni nei calcolatori dotati di pipeline.

Thornton, J. E. [1970]. *Design of a Computer: The Control Data 6600*, Scott, Foresman, Glenview, IL.
Un libro classico che descrive un elaboratore classico, considerato il primo supercomputer.

6.13 Parole chiave

I termini elencati nel seguito riflettono le idee chiave discusse in questo capitolo. Coloro che non sono sicuri del significato dei termini sono invitati a fare riferimento al Glossario che si trova al fondo del testo.

branch delay slot
buffer di previsione dei salti, o tabella di storia dei salti
buffer di riordino
buffer o registro replicato
consegna fuori sequenza
consegna sequenziale
criticità dovuta a salti o segnali di controllo
criticità strutturale
criticità sui dati
criticità sui dati usati dalle load
dipendenze sui dati
esecuzione fuori sequenza
esecuzione multipla di istruzioni
esecuzione sequenziale
esecuzione speculativa
espansione dei cicli
interruzione o eccezione imprecisa
interruzioni o eccezioni imprecise
latenza (pipeline)
latenza dell'istruzione
load ritardata
nop
pipeline superscalare
predizione dei salti
propagazione
scaricamento (delle istruzioni)
schedulazione dinamica della pipeline
stadio della pipeline
stallo della pipeline
stazione di prenotazione
superpipeline
unità di consegna

6.14 Esercizi

6.1 [5] <§6.1> Se il tempo effettivo di esecuzione per una operazione della ALU fosse di 4 ns invece di 2 ns (come descritto in figura 6.2), quali sarebbero le conseguenze sull'aumento di velocità ottenuto aggiungendo la pipeline all'implementazione a singolo ciclo?

6.2 [5] <§6.1> Usando una rappresentazione grafica simile a quella di figura 6.8, mostrare i cammini di propagazione necessari ad eseguire le seguenti tre istruzioni:

```
add   $2, $3, $4
add   $4, $5, $6
add   $5, $3, $4
```

6.3 [5] <§6.1> Come potrebbe essere modificato il codice seguente per sfruttare un delayed branch slot?

```
Ciclo: lw   $2, 100($3)
       add  $3, $3, 4
       beq  $3, $4, Ciclo
```

6.4 [10] <§6.1> Identificare tutte le dipendenze dei dati all'interno del codice seguente. Quali dipendenze sono criticità risolvibili con la propagazione?

```
add   $2, $5, $4
add   $4, $2, $5
sw    $5, 100($2)
add   $3, $2, $4
```

6.5 [5] <§6.2> Etichettare ciascuna porzione di tutti i registri di pipeline di figura 6.25 con il nome relativo al valore caricato nel registro. Determinare la lunghezza in bit di ciascun campo. Ad esempio il registro di pipeline IF/ID contiene due campi, uno dei quali è un campo istruzione lungo 32 bit.

6.6 [15] <§§4.8, 6.2> Usando come base la figura 4.42, pensare ad una struttura ragionevole per un'unità di elaborazione per l'esecuzione di somme in virgola mobile dotata di pipeline, ed integrare nello schema i registri per i numeri in virgola mobile e la memoria dati, producendo una figura simile a quella di figura 6.12 (a parte l'aggiunta di una pipeline alternativa per le istruzioni di somma in virgola mobile). Non è necessario preoccuparsi eccessivamente dei ritardi associati a ciascuna delle funzioni di figura 4.42, basta assumere che la somma dei ritardi sia abbastanza grande, e che quindi richieda 2 o 3 cicli per eseguire le somme in virgola mobile. L'istruzione di somma in virgola mobile deve essere scomposta in stadi in modo che in ogni ciclo sia possibile iniziarne una nuova, assumendo che non esistano dipendenze fra i dati (in questo esercizio si può ignorare il problema delle dipendenze fra i dati).

6.7 [10] <§6.2> Usando come base la figura 6.25, evidenziare utilizzando delle penne colorate quali porzioni dell'unità di elaborazione sono attive, e quali inattive, in ciascuno dei cinque stadi dell'istruzione `add`. Per risolvere questo esercizio si consiglia di usare quattro fotocopie della figura 6.25 (gli autori concedono il permesso di violare il diritto di copyright risolvendo gli esercizi dei Capitoli 5 e 6!). È inoltre possibile ottenere la figura all'indirizzo *www.mkp.com/cod2e.htm*. È importante includere una legenda per spiegare l'uso dei colori.

6.8 [15] <§6.3> Per assicurarsi di avere compreso la relazione fra i due stili di progettazione delle pipeline, riportare le informazioni delle figure da 6.31 a 6.35 nello stile di figura 6.36. Evidenziare le porzioni attive dell'unità di elaborazione in questa figura più semplice.

6.9 [20] <§6.3> La figura 6.71 è simile alla figura 6.33, ma le istruzioni non sono identificate. In questo esercizio il compito consiste nel capire il più possibile riguar-

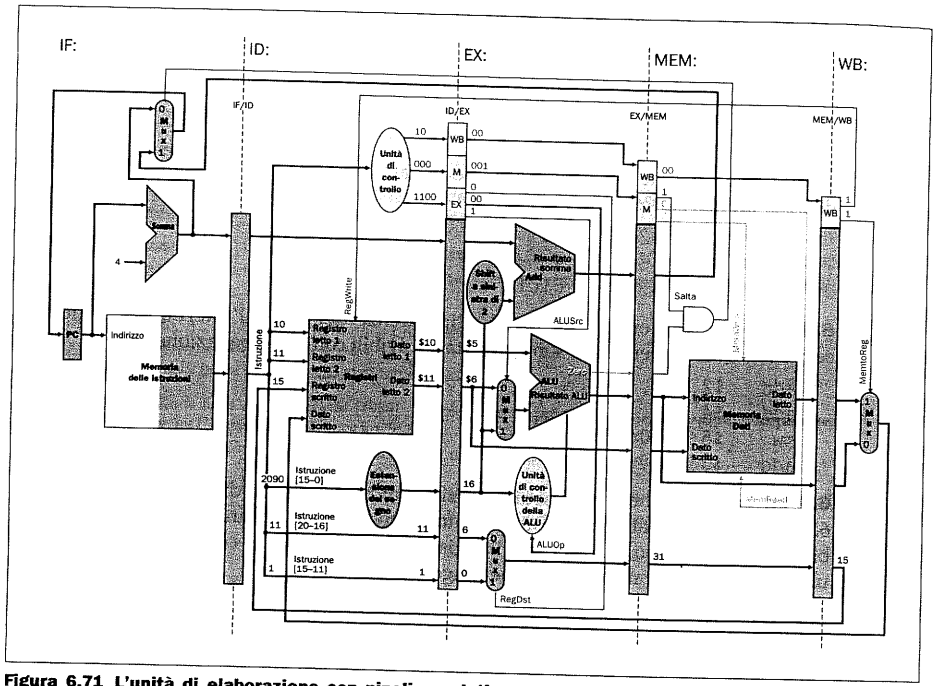


Figura 6.71 L'unità di elaborazione con pipeline relativa all'esercizio 6.9. Usare etichette numeriche per determinare il maggior numero possibile dei campi delle cinque istruzioni all'interno della pipeline.

do le cinque istruzioni contenute nei cinque stadi della pipeline. Se non si riesce a riempire un campo di una istruzione spiegarne il motivo. Per alcuni campi risulta più semplice decodificare le istruzioni macchina in linguaggio assembler, usando come riferimento le figure 3.18 e A.18. Nel caso di altri campi è più facile considerare i valori dei segnali di controllo, usando come riferimento le figure da 6.26 a 6.28. Può essere necessario esaminare attentamente le figure da 6.31 a 6.35 per capire come sono presentati gli insiemi dei valori di controllo (cioè il bit più a sinistra in un ciclo diventa il bit superiore in un altro ciclo). Ad esempio il valore di controllo EX dell'istruzione di sottrazione (1100), calcolato durante lo stadio ID del ciclo 3 in figura 6.32, nel ciclo 4 si trasforma in tre valori separati specificando RegDst (1), ALUOp (10) e ALUSrc (0).

6.10 [40] <§6.3> Usando la figura 6.33 determinare il valore di *tutti* i campi dei quattro registri di pipeline nel ciclo di clock 5 (cioè i valori all'inizio di ciascun ciclo di clock). Si assuma che prima dell'esecuzione delle istruzioni lo stato della macchina sia il seguente:

- Il PC contiene il valore 500_{dieci}, cioè l'indirizzo dell'istruzione `lw`.
- Ciascun registro ha come valore iniziale 10_{dieci} più il numero del registro (ad esempio il registro `$8` ha come valore iniziale 18_{dieci}).

- Ciascuna parola di memoria contenente dati a cui è stato fatto accesso ha come valore iniziale 1000_{dieci} più l'indirizzo in byte della parola (ad esempio Memoria[8] ha come valore iniziale 1008_{dieci}).

Determinare il valore di ciascun campo, inclusi quelli non identificati nella figura e quelli non necessari in una specifica istruzione. Nel caso si ritenga che il valore di un dato campo sia impossibile da determinare a partire dalle informazioni fornite se ne spieghi la ragione.

6.11 [5] <§§6.4, 6.5> Considerare l'esecuzione del seguente codice nell'unità di elaborazione con pipeline di figura 6.46:

```
add   $1, $2, $3
add   $4, $5, $6
add   $7, $8, $9
add   $10, $11, $12
add   $13, $14, $15
```

Al termine del quinto ciclo di esecuzione quali registri sono in lettura e quali saranno scritti?

6.12 [5] <§§6.4, 6.5> {Es. 6.11} Con riferimento al programma dell'esercizio 6.11 spiegare che cosa fa l'unità di propagazione durante il quinto ciclo dell'esecuzione. Citare eventuali paragoni.

6.13 [5] <§§6.4, 6.5> {Es. 6.11} Con riferimento al programma dell'esercizio 6.11 spiegare che cosa fa l'unità di rilevazione delle condizioni critiche durante il quinto ciclo dell'esecuzione. Citare gli eventuali paragoni.

6.14 [5] <§6.5> Considerare un programma che consiste di 100 istruzioni `lw` (senza preoccuparsi se sia o meno un buon codice) e in cui ogni istruzione dipende dall'istruzione precedente. Quale sarebbe l'effettivo CPI se il programma fosse eseguito sull'unità di elaborazione con pipeline di figura 6.45?

6.15 [5] <§§6.4, 6.5> Considerare l'esecuzione del seguente codice nell'unità di elaborazione con pipeline di figura 6.46:

```
add   $5, $6, $7
lw    $6, 100($7)
sub   $7, $6, $8
```

Quanti cicli sono necessari per eseguire questo codice? Disegnare un diagramma analogo a quello di figura 6.44 che illustri le dipendenze da risolvere, e un diagramma analogo a quello di figura 6.45 che illustri come il codice venga effettivamente eseguito (comprese condizioni di stallo o propagazione) per risolvere i problemi identificati.

6.16 [15] <§6.5> Elencare tutti gli ingressi e tutte le uscite dell'unità di propagazione di figura 6.42. Indicare nome e numero di bit di ciascun ingresso e ciascuna uscita.

6.17 [30] <§6.5, Appendice C> {Es. 6.16} Usando l'Appendice C e la soluzione dell'esercizio 6.16 progettare l'hardware che implementa l'unità di propagazione. Suggerimento: per decidere se i numeri dei registri sono uguali, si provi ad usare una porta Exor (vedere l'approfondimento nel Paragrafo 4.5 o il dettaglio del Capitolo 4 relativo alle istruzioni logiche).

6.18 [20] <§6.5> L'unità di propagazione potrebbe essere spostata allo stadio ID e le decisioni sulla propagazione potrebbero essere prese prima. Il risultato di queste

decisioni dovrebbe essere passato insieme all'istruzione ed usato nello stadio EX, in cui viene eseguita l'effettiva propagazione. Questa modifica renderebbe più veloce lo stadio EX e potrebbe migliorare il tempo di ciclo. Eseguire la modifica, e fornire la versione rivista dell'unità di elaborazione con la descrizione dei cambiamenti necessari. Come è cambiato il registro ID/EX? Fornire delle nuove equazioni di propagazione per sostituire quelle riportate nel Paragrafo 6.4.

6.19 [10] <§§6.4, 6.5> Il seguente codice contiene una criticità sui dati del tipo "lettura dopo scrittura", risolvibile tramite propagazione:

```
add   $2, $3, $4
add   $5, $2, $6
```

Considerare una situazione analoga, in cui una lettura in memoria segue una scrittura in memoria:

```
sw    $7, 100($2)
lw    $8, 100($2)
```

Descrivere in che cosa questa situazione differisce da quella precedente e come può essere risolto in questo caso il potenziale problema della "lettura dopo scrittura".

6.20 [20] <§§6.4, 6.5> Si consideri una sequenza di istruzioni usata per copiare dati da una locazione di memoria ad un'altra:

```
lw    $2, 100($5)
sw    $2, 200($6)
```

L'ultimo approfondimento riportato nel Paragrafo 6.4 prende in considerazione questa situazione e suggerisce che dei circuiti di propagazione aggiuntivi possano migliorare le prestazioni. Mostrare le aggiunte necessarie all'unità di elaborazione di Figura 6.43 perché codice di questo tipo possa essere eseguito senza il verificarsi di situazioni di stallo. Riportare anche le equazioni di propagazione (come quelle che compaiono nel Paragrafo 6.4) per tutti i segnali di controllo riguardanti i multiplexer dell'unità di elaborazione aggiunti o modificati. Riscrivere infine la formula di stallo riportata nel Paragrafo 6.5 per far sì che non si verifichino condizioni di questo tipo.

6.21 [15] <§§6.2-6.5> Nell'Esercizio 5.5 la possibilità di specificare uno spiazzamento per le istruzioni di load e di store era stata eliminata. Come influirebbe questa modifica del set di istruzioni sull'implementazione della pipeline? Descrivere i cambiamenti all'unità di elaborazione e l'impatto sulle prestazioni. Assicurarsi di inserire nella risposta anche una discussione sulla propagazione.

6.22 [10] <§§6.2-6.5> L'Esercizio 3.23 ha descritto un'istruzione, `addm`, che consente alle istruzioni aritmetiche di accedere direttamente in memoria, come nel caso dell'80x86. Spiegare perché sarebbe difficile aggiungere questa istruzione alla pipeline MIPS descritta in questo capitolo.

6.23 [10] <§§6.4-6.6> L'esempio sul riordino del codice all'interno del Paragrafo 6.1 mostra come *massimizzare* le prestazioni di un'unità di elaborazione dotata di pipeline con unità di propagazione e di stallo per l'utilizzo successivo ad una lettura in memoria. Riscrivere il seguente codice per *minimizzare* le prestazioni in un'unità di elaborazione di questo tipo, riordinare cioè le istruzioni in modo tale che per essere eseguita la sequenza impieghi il *maggior* numero di cicli di clock ottenendo lo stesso risultato.


```
lw    $3, 0($5)
lw    $4, 4($5)
add   $7, $7, $3
add   $8, $8, $4
add   $10, $7, $8
sw    $6, 0($5)
beq   $10, $11, Ciclo
```

6.24 [15] <§6.6> Usando l'esempio sulla predizione dei salti del Paragrafo 6.6, riscrivere il codice in modo che sia eseguito il più rapidamente possibile usando una nuova istruzione `beqd`, cioè un'istruzione di branch equal ritardata con un single-branch delay slot.

6.25 [10] <§6.6> {Es. 6.24} Usando la risposta all'Esercizio 6.24, schematizzare l'esecuzione dell'istruzione come nella figura 6.52. Fotocopiare la figura, o cercarla all'indirizzo *www.mkp.com/cod2e.htm* permette di risparmiare tempo.

6.26 [20] <§6.6> Considerare l'unità di elaborazione con pipeline di figura 6.51. Può accadere che un tentativo di scaricamento e di stallo accadano contemporaneamente? Se ciò può avvenire, il risultato sono operazioni in conflitto o in cooperazione? Se ci sono azioni in cooperazione, come lavorano insieme? Se ci sono operazioni in conflitto, quale dovrebbe avere la priorità? Esiste un semplice cambiamento all'unità di elaborazione che consenta di assicurare le priorità necessarie? Per rispondere a questa domanda può essere utile considerare il seguente codice:

```
                beq $1, $2, Destinazione # si assuma che il
                                         # salto venga eseguito
                lw  $3, 40($4)
                add $3, $3, $3
                sw  $3, 40($4)
Destinazione:   or  $10, $11, $12
```

6.27 [30] <§6.6> In quale stadio deve essere effettuata la decisione sui salti per ridurre ad una singola istruzione il ritardo dovuto ai salti? Ridisegnare l'unità di elaborazione usando del nuovo hardware che riduca ad un solo ciclo di clock il ritardo dovuto ai salti.

6.28 [10] <§6.6> Un'estensione del set di istruzioni MIPS ha due nuove istruzioni chiamate `movn` (sposta se diverso da zero) e `movz` (sposta se uguale a zero). Ad esempio l'istruzione

```
movn  $8, $11, $4
```

copia il contenuto del registro 11 nel registro 8, purché il valore nel registro 4 sia diverso da zero (nel caso contrario non fa nulla). L'istruzione `movz` è simile, ma la copia avviene solo se il valore nel registro è uguale a zero. Mostrare l'utilizzo delle nuove istruzioni per mettere nel registro 10 il valore più elevato fra quello contenuto nel registro 8 e quello contenuto nel registro 9. Se i valori sono uguali copiare uno qualsiasi dei due nel registro 10. È possibile usare il registro 1 come registro aggiuntivo temporaneo. Non usare salti condizionati.

6.29 [10] <§6.6> {Es. 6.28} La soluzione dell'Esercizio 6.28 dovrebbe contenere meno istruzioni rispetto all'utilizzo dei salti condizionati. Qualche volta però la riscrittura del codice usando `movn` e `movz` invece dei salti condizionati non riduce il numero di istruzioni eseguite. Nonostante ciò anche nel secondo caso l'uso di `movn`

e `movz` può comunque rendere il programma più veloce da eseguire con un'unità di elaborazione dotata di pipeline: spiegarne il motivo.

6.30 [10] <§6.6> In questo esercizio, simile all'esempio del Paragrafo 6.8, vengono analizzati i benefici dell'uso dell'espansione dei cicli su codice eseguito sulla pipeline MIPS standard descritta nei Paragrafi da 6.2 a 6.6. Nel caso specifico si consideri il seguente codice, espanso ma non ancora schedulato. In questo caso il codice è stato espanso una volta, assumendo che l'indice che controlla il ciclo sia multiplo di due (che cioè `$s1` sia multiplo di otto):

```
Ciclo: lw     $t0, 0($s1)
       addu   $t0, $t0, $s2
       sw     $t0, 0($s1)
       lw     $t1, -4($s1)
       addu   $t1, $t1, $s2
       sw     $t1, -4($s1)
       addi   $s1, $s1, -8
       bne    $s1, $zero, Ciclo
```

Per prima cosa schedulare il codice per farlo eseguire rapidamente con la pipeline MIPS standard (assumendo che questa supporti le istruzioni `addi` e `addu`). Tenendo conto di tutte le situazioni di stallo necessarie, confrontare le prestazioni del codice originale, non espanso, e del nuovo codice, espanso e schedulato.

6.31 [20] <§6.8> Questo esercizio è simile all'Esercizio 6.30, eccetto che questa volta il codice deve essere espanso due volte (creando tre copie del codice). Non è però noto se l'indice che controlla il ciclo sia multiplo di tre, e quindi deve essere trovato un modo per assicurare che il codice venga ugualmente eseguito in modo corretto. Si consiglia di aggiungere del codice all'inizio o alla fine del ciclo per tenere conto dei casi non gestiti dal ciclo.

6.32 [20] <§6.9> Modificare processori complessi come il PowerPC 604 o il Pentium Pro per migliorare le prestazioni implica decisioni complesse che possono essere prese solo con l'aiuto di simulazioni estensive. Si assuma che i miglioramenti tecnologici abbiano aumentato la disponibilità di transistor e che sia possibile aumentare alcuni dei parametri riportati in figura 6.63. Quali assunzioni possono far scegliere specifici parametri da migliorare? Come sono legati alcuni di questi parametri?

6.33 [30] <§6.9> Nuovi processori vengono introdotti sul mercato più frequentemente di nuovi libri. Per mantenere il libro aggiornato, investigare alcuni dei recenti sviluppi in quest'area, e scrivere un approfondimento di una pagina alla fine del Paragrafo 6.9. Come partenza può essere interessante consultare il sito MIPS tramite il collegamento all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*.

6.34 [1 settimana] <§§6.4, 6,5> Usando il simulatore allegato al libro raccogliere le statistiche sulle criticità sui dati all'interno di un programma C (fornito dal docente o con il software). Scrivere una procedura a cui viene passata l'istruzione che deve essere eseguita; tale procedura deve modellare la pipeline a cinque stadi descritta in questo capitolo. Il programma deve raccogliere le seguenti statistiche:

- Numero di istruzioni eseguite.
- Numero di criticità sui dati.
- Numero di criticità che provocano situazioni di stallo.

- Se il compilatore C MIPS usato genera delle istruzioni `nop` per evitare le criticità, contare anche il numero di `nop`.

Assumendo che gli accessi in memoria richiedano sempre 1 ciclo di clock, calcolare il numero medio di istruzioni per ciclo. Classificare le istruzioni `nop` come condizioni di stallo inserite dal programma, e quindi sottrarle dal numero di istruzioni eseguite nel conteggio del CPI.

6.35 [1 mese] <§§5.3, 6.3-6.7> Avendo l'accesso ad un simulatore come Verilog o ViewLogic, per prima cosa progettare l'unità di elaborazione a singolo ciclo e l'unità di controllo a partire dal Capitolo 5. Espandere quindi il progetto con un'organizzazione dotata di pipeline, come in questo capitolo. Assicurarsi di eseguire programmi MIPS ad ogni passo per verificare che il progetto modificato continui ad operare in modo corretto.

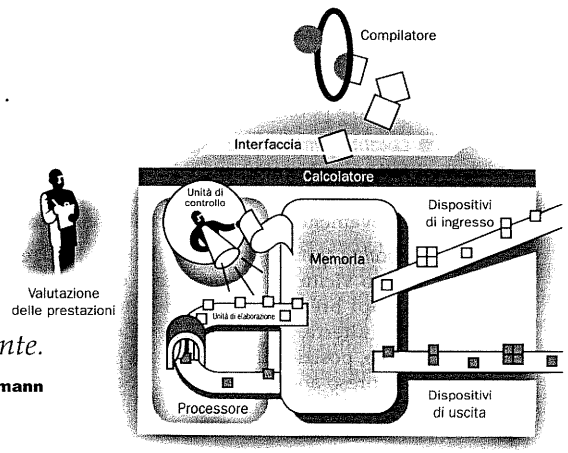
7

Capacità e velocità: sfruttare la gerarchia delle memorie

L'ideale sarebbe avere una memoria con capacità indefinitamente grande in modo che ciascuna specifica ... parola fosse immediatamente disponibile... Siamo ... costretti a riconoscere la possibilità di costruire una gerarchia di memorie, ciascuna delle quali dotata di maggior capacità della precedente, ma accessibile meno velocemente.

A. W. Burks, H. H. Goldstine e J. von Neumann
Discussione preliminare sul progetto logico di un dispositivo elettronico di calcolo, 1946

Le cinque componenti classiche del calcolatore



7.1 Introduzione

Sin dai primissimi tempi dell'era informatica, i programmatori hanno desiderato quantità illimitate di memoria. Il contenuto di questo capitolo si concentrerà su un meccanismo creato per dare ai programmatori l'illusione di disporre di una memoria veloce ed illimitata. Prima di esaminare come tale illusione venga effettivamente creata, si introdurrà una semplice analogia per meglio illustrare i principi fondamentali ed i meccanismi su cui si basa la realizzazione dei sistemi di memoria.

Si prenda in considerazione uno studente che deve scrivere una relazione sugli sviluppi storici più importanti degli aspetti hardware dei calcolatori. Lo studente siede ad una scrivania nella biblioteca di ingegneria con alcuni libri che ha preso dagli scaffali e che sta esaminando. Ha scoperto che molti dei calcolatori importanti che deve trattare nella relazione sono descritti in quei libri, ma non vi è nulla a proposito del calcolatore EDSAC. Lo studente si alza e va a cercare un altro libro: ne trova uno sui primi calcolatori britannici che tratta anche EDSAC. Una volta che lo studente avrà davanti a sé sulla scrivania una buona selezione di libri è probabile che molti degli argomenti di cui avrà bisogno saranno in essi contenuti, e potrà spendere la maggior parte del tempo consultando i testi sulla scrivania senza dover ritornare agli scaffali. La possibilità di avere parecchi libri davanti a sé fa risparmiare tempo rispetto a poterne avere uno solo, in quanto si dovrebbe continuamente tornare agli scaffali per riportare il libro e prenderne un altro.

Lo stesso principio permette di creare l'illusione di una memoria di grandi dimensioni a cui si può accedere con la stessa velocità di una piccola memoria: come non era necessario accedere a tutti i testi della biblioteca con la stessa probabilità, così un programma non ha la necessità di accedere a tutto il suo codice ed a tutti i suoi dati con egual probabilità. In caso contrario non sarebbe possibile rendere veloci la maggior parte degli accessi in memoria e contemporaneamente avere grandi quantità di memoria nei calcolatori, come sarebbe impossibile tenere tutti i libri della biblioteca sulla scrivania ed avere qualche possibilità di trovare velocemente l'argomento cercato.

Questo *principio di località* sta alla base sia della metodologia con cui si lavora in biblioteca, sia del modo in cui funzionano i programmi. Il principio di località dice che in un dato istante di tempo i programmi fanno accesso ad una porzione relativamente piccola del loro spazio di indirizzamento, così come lo studente faceva accesso ad una piccola porzione dei libri della biblioteca. Vi sono due tipi diversi di località:

- *Località temporale* (o località nel tempo): è probabile che un oggetto a cui si è fatto riferimento venga nuovamente richiesto in tempo brevi. Se lo studente ha portato un libro sulla scrivania per guardarlo, probabilmente ne avrà nuovamente bisogno non molto tempo dopo.
- *Località spaziale* (o località nello spazio): è probabile che gli oggetti che si trovano vicini ad un oggetto a cui si è fatto riferimento vengano richiesti in tempi brevi. Per esempio, nel prendere il libro sui primi calcolatori in Inghilterra per trovare notizie su EDSAC, lo studente ha notato un altro libro, posizionato a fianco del primo, sui primi elaboratori meccanici, per cui ha preso anche tale libro e, successivamente, vi ha trovato qualcosa di utile. Nelle biblioteche i libri sullo stesso argomento sono posti insieme per aumentare la località spaziale; un po' più avanti nel capitolo si vedrà come è utilizzata la località spaziale nelle gerarchie di memorie.

Così come gli accessi ai libri godono spontaneamente delle proprietà di località, nei programmi la località nasce da strutture di programmazione semplici e naturali. Per esempio molti programmi contengono dei cicli, per cui le istruzioni ed i dati saranno probabilmente richiesti ripetutamente, portando ad un buon grado di località temporale. Siccome durante l'esecuzione di un programma le istruzioni sono normalmente richieste in modo sequenziale, i programmi hanno un'elevata località spaziale. Anche gli accessi ai dati mostrano una località spaziale spontanea; ad esempio gli accessi agli elementi di un vettore o ai campi di una struttura hanno per loro natura un'alta località spaziale.

Il principio di località viene messo a frutto implementando la memoria di un calcolatore sotto forma di *gerarchia di memorie*. La gerarchia delle memorie consiste di più livelli di memoria con velocità e dimensioni diverse. Le memorie più veloci hanno un costo per bit maggiore delle memorie più lente, per cui sono generalmente più piccole.

Oggigiorno vi sono tre tecnologie principali per la costruzione delle gerarchie di memorie: la memoria principale è realizzata con DRAM (memoria dinamica ad accesso casuale, dynamic random access memory), mentre i livelli più vicini alla CPU (le cosiddette cache) usano SRAM (memoria statica ad accesso casuale, static random access memory). La DRAM ha un costo per bit inferiore alla SRAM, ma è decisamente più lenta. La differenza di prezzo è determinata dal fatto che nelle memorie DRAM si usa un'area notevolmente minore per ciascun bit di memoria, e quindi le DRAM hanno una capacità maggiore a pari quantità di silicio; la differenza di velocità deriva da diversi fattori, descritti nel Paragrafo B.5 dell'Appendice B. L'ultima tecnologia, usata per implementare il livello più capiente e più lento della gerarchia, è rappresentata dai dischi magnetici. Il tempo di accesso ed il costo per bit variano notevolmente tra queste tecnologie, come mostrato dalla tabella seguente, che riporta valori tipici per il 1997 espressi in dollari e in modo da non risentire delle fluttuazioni del tasso di cambio:

Tecnologia di memorizzazione	Tempo di accesso tipico	$ per MByte nel 1997
SRAM	5-25 ns	$100-$250
DRAM	60-120 ns	$5-$10
Dischi magnetici	10-20 milioni ns	$0,10-$0,20

Viste le differenze di costo e di tempo di accesso, è vantaggioso costruire il sottosistema di memoria come una gerarchia di livelli, con la memoria più veloce vicina al processore e quella più lenta e meno costosa ai livelli inferiori, come mostrato in figura 7.1. L'obiettivo è di presentare all'utente tanta memoria quanta ne sia disponibile nella tecnologia meno costosa con una velocità di accesso pari a quella offerta dalla tecnologia più veloce.

Il sottosistema di memoria è organizzato come una gerarchia: dato un livello, il suo contenuto è un sottoinsieme di quello di qualunque altro livello che sia più distante dal processore, mentre al livello più basso sono memorizzati tutti i dati. In modo analogo, i libri sulla scrivania sono un sottoinsieme di quelli della biblioteca in cui ci si trova, che a loro volta sono un sottoinsieme di tutte le biblioteche dell'università. Inoltre, man mano che ci si allontana dal processore, i livelli richiedono un tempo sempre maggiore per l'accesso, come ci si potrebbe aspettare in una gerarchia di biblioteche.

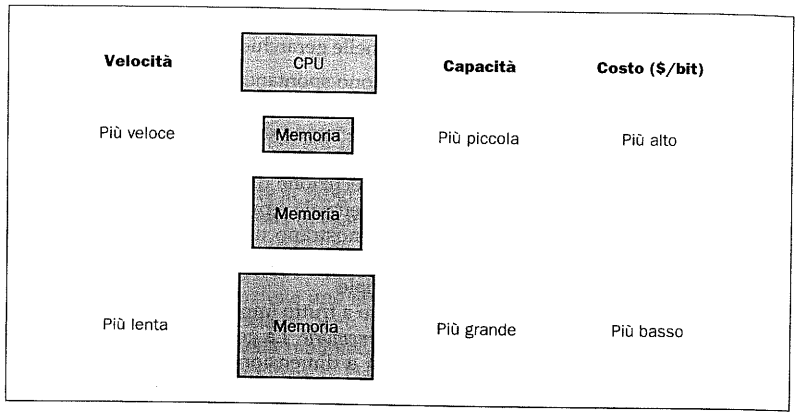


Figura 7.1 Struttura fondamentale della gerarchia delle memorie. Implementando il sottosistema di memoria come una gerarchia, l'utente ha l'illusione di una memoria ampia quanto il livello più ampio nella gerarchia, a cui si fa comunque accesso alla velocità della memoria più veloce.

La gerarchia delle memorie può consistere di più livelli, ma in ogni istante di tempo i dati sono copiati solamente tra ciascuna coppia di livelli adiacenti, per cui ci si può concentrare su due soli livelli. Il livello superiore (quello più vicino al processore) è più piccolo e veloce (poiché usa una tecnologia più costosa) di quello inferiore. L'unità minima di informazione che può essere presente o assente in una gerarchia a due livelli è detto *blocco*, come indicato in figura 7.2; nell'analogia della biblioteca, un blocco di informazioni è costituito da un libro.

Se i dati richiesti dal processore compaiono in qualche blocco nel livello superiore, si dice che si è verificato un *successo nell'accesso*, detto nel seguito *hit* (ossia il dato è stato "colpito", come quando si trova l'informazione cercata in un libro già sulla scrivania). Se il dato non è presente al livello superiore, la richiesta viene classificata come *fallimento nell'accesso*, o *miss* (ossia il dato "manca") e si fa accesso al

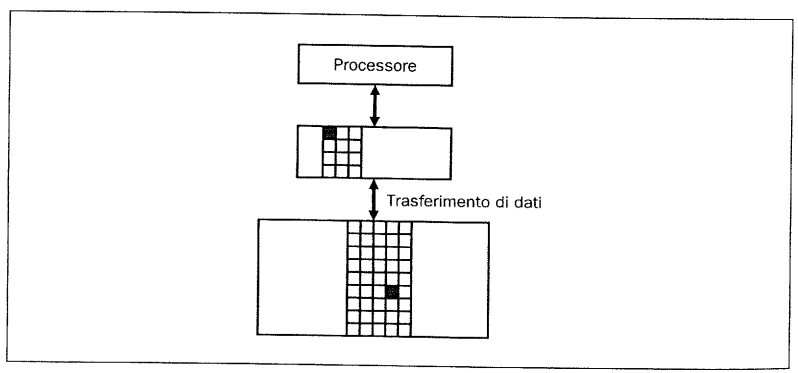


Figura 7.2 Ciascuna coppia di livelli nella gerarchia delle memorie è costituita da un livello superiore ed uno inferiore. All'interno di ciascun livello l'unità di informazione che può essere presente o no è detta blocco. Solitamente quando si copia qualcosa tra due livelli si trasferisce un intero blocco.

livello inferiore nella gerarchia per recuperare il blocco contenente il dato richiesto (proseguendo nell'analogia, ci si alza dal tavolo per cercare l'informazione richiesta tra gli scaffali). Il parametro detto *hit rate* (*frequenza di hit*) o *hit ratio* (*frazione degli hit*) è la frazione degli accessi alla memoria che si sono risolti al livello superiore; esso è spesso usato per misurare le prestazioni della gerarchia delle memorie. Il parametro detto *miss rate* (*frequenza di miss*, pari a 1 – *frequenza di hit*) è la frazione degli accessi in memoria che non sono stati soddisfatti al livello superiore.

Poiché il motivo principale per avere una gerarchia di memorie è l'incremento delle prestazioni, la velocità con cui si gestiscono hit e miss è importante: il *tempo di hit* è il tempo necessario ad accedere al livello superiore della gerarchia delle memorie, comprendendo il tempo necessario a determinare se l'accesso è un hit o un miss (ossia il tempo necessario a sfogliare i libri sulla scrivania). La *penalità di miss* è il tempo necessario per sostituire un blocco nel livello superiore con il blocco corrispondente del livello inferiore, più il tempo necessario per consegnare il dato al processore (o il tempo necessario per prendere un altro libro dagli scaffali e portarlo sulla scrivania). Poiché il livello superiore è più piccolo ed è composto di componenti più veloci, il tempo di hit sarà molto minore del tempo necessario ad accedere al livello successivo nella gerarchia, che corrisponde alla componente preponderante del tempo di miss (il tempo necessario ad esaminare i libri sulla scrivania è molto minore di quello che serve per alzarsi ed andare a cercare qualcosa sugli scaffali).

Come si vedrà in questo capitolo, i concetti utilizzati nella costruzione dei sottosistemi di memoria influiscono su molti altri aspetti di un calcolatore, tra cui la modalità di gestione della memoria e dei dispositivi di ingresso/uscita da parte del sistema operativo, la generazione del codice nei compilatori, e addirittura il modo di usare l'elaboratore da parte delle applicazioni. Naturalmente, poiché tutti i programmi spendono la maggior parte del loro tempo accedendo alla memoria, il sottosistema di memoria è necessariamente un fattore determinante ai fini delle prestazioni. L'utilizzo delle gerarchie di memorie per ottenere prestazioni migliori richiede che i programmatori, abituati a pensare alla memoria come ad un dispositivo piatto e dotato di accesso casuale, debbano comprendere il funzionamento delle gerarchie di memorie per ottenere buone prestazioni. Si vedrà quanto sia importante questa comprensione in un esempio del Paragrafo 7.7.

Essendo il sottosistema di memoria così critico ai fini delle prestazioni, i progettisti di calcolatori vi hanno dedicato molta attenzione ed hanno sviluppato meccanismi sofisticati per migliorarne le prestazioni. In questo capitolo si vedranno le idee concettuali più importanti, sebbene si siano usate molte semplificazioni ed astrazioni per poter offrire un materiale non eccessivo in termini di lunghezza e complessità. Si sarebbero potute facilmente scrivere centinaia di pagine sui sottosistemi di memoria, come hanno dimostrato molte recenti tesi di dottorato.

Quadro d'insieme

I programmi sono caratterizzati sia da una località temporale (la tendenza a riusare i dati a cui hanno fatto accesso di recente), sia da una località spaziale (la tendenza a far riferimento a dati vicini a quelli a cui hanno fatto accesso di recente). Le gerarchie di memorie traggono vantaggio dalla località temporale mantenendo i dati a cui si è fatto accesso più di recente in posizioni più vicine al processore. Le gerarchie di memorie

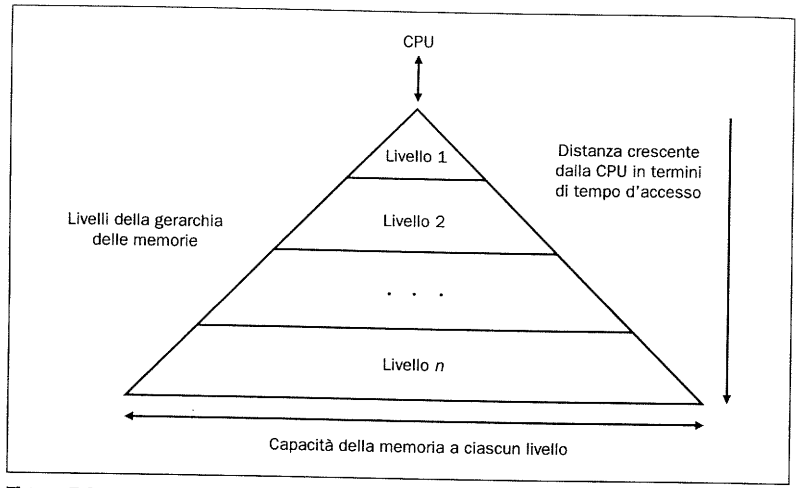


Figura 7.3 Il diagramma mostra la struttura generale della gerarchia delle memorie: al crescere della distanza dalla CPU, crescono le dimensioni. Questa struttura, grazie agli opportuni meccanismi operativi, permette alla CPU di avere un tempo di accesso determinato essenzialmente dal livello 1 della gerarchia e di disporre tuttavia di una memoria di ampiezza pari a quella del livello *n*. L'obiettivo di questo capitolo è la realizzazione di questa illusione.

traggono vantaggio dalla località spaziale spostando verso i livelli superiori della gerarchia blocchi composti di molte parole consecutive.

Una gerarchia di memorie usa memorie più piccole e di tecnologia più veloce vicino al processore, come mostrato in figura 7.3: gli accessi che hanno successo al livello superiore (hit) vengono quindi elaborati velocemente. Gli accessi non soddisfacibili (miss) devono ricorrere ai livelli inferiori della gerarchia, che sono più capienti ma più lenti. Se la frequenza di hit (hit rate) è sufficientemente elevata, la gerarchia delle memorie ha un tempo di accesso effettivo vicino a quello del livello più alto (e più veloce) ed una capacità pari a quella del livello più basso (e più capiente).

7.2 Fondamenti sulle cache

Cache: un posto sicuro in cui nascondere o immagazzinare oggetti.

Webster's New World Dictionary of the American Language, Third College Edition (1988)

Nell'esempio della biblioteca, la scrivania svolgeva il ruolo di una *cache* (termine inglese che significa nascondiglio): un posto sicuro per immagazzinare gli oggetti (cioè i libri) che si devono esaminare. Il termine *cache* è stato usato per la prima

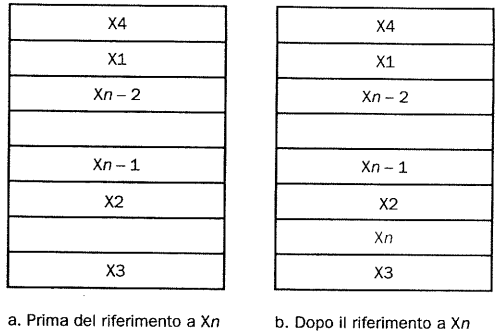


Figura 7.4 La cache subito prima e subito dopo il riferimento ad una parola X*n* inizialmente non presente. Il riferimento fa sì che si debba prelevare X*n* dalla memoria ed inserirla nella cache.

volta per indicare il livello della gerarchia tra la CPU e la memoria principale nel primo calcolatore commerciale dotato di tale livello; benché oggigiorno il termine sia ancora utilizzato principalmente in tal senso, esso indica anche qualsiasi tipo di gestione della memoria che tragga vantaggio dalla località degli accessi. Le cache sono comparse all'inizio degli anni '60 sui calcolatori destinati alla ricerca e si sono estese ai sistemi commerciali verso la fine del decennio: attualmente, praticamente tutti gli elaboratori, dal più veloce al più lento, contengono una cache.

In questo paragrafo si esaminerà una cache semplificata in cui il processore richiede una parola per volta e ciascun blocco contiene una sola parola. La figura 7.4 mostra una cache di questo tipo, prima e dopo la richiesta di un dato che inizialmente non era nella cache. Inizialmente la cache contiene un insieme di dati richiesti di recente (X1, X2, ..., X*n*-1); ad un certo punto il processore richiede la parola X*n* non presente nella cache: questo tipo di richiesta è classificato come miss e la parola X*n* viene trasferita dalla memoria alla cache.

Esaminando lo scenario di figura 7.4 emergono due domande legittime: Come si può sapere se un dato è nella cache? Se il dato è presente, come lo si trova? In effetti le risposte a queste due domande sono correlate: se ciascuna parola potesse risiedere in un solo posto nella cache, si saprebbe come determinare se la parola è presente. Il modo più semplice per associare una posizione della cache a ciascuna parola di memoria è di determinare la posizione nella cache sulla base dell'indirizzo della parola in memoria. Le cache con questa struttura sono dette *a corrispondenza diretta* (in inglese *direct mapped*), poiché a ciascuna parola di memoria corrisponde esattamente una locazione della cache. Solitamente la regola di traduzione degli indirizzi per le cache a corrispondenza diretta è abbastanza semplice; ad esempio, quasi tutte le cache a corrispondenza diretta usano la regola seguente:

(indirizzo del blocco) modulo (numero di blocchi nella cache)

Questa corrispondenza è interessante perché, se il numero di elementi della cache è una potenza di 2, l'operazione di modulo si può calcolare semplicemente utilizzando alcuni dei bit meno significativi dell'indirizzo, in particolare si usano

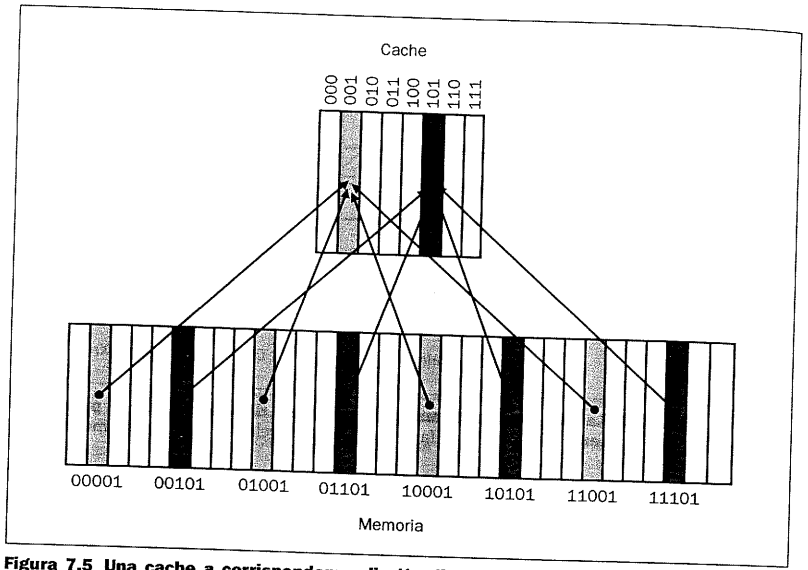


Figura 7.5 Una cache a corrispondenza diretta di otto elementi con indicati gli indirizzi delle parole di memoria tra 0 e 31 che corrispondono alle stesse locazioni della cache. Essendoci otto parole nella cache, un indirizzo X corrisponde alla parola X modulo 8 nella cache. In altri termini, i $\log_2(8)=3$ bit meno significativi sono usati come indice per la cache. Di conseguenza, gli indirizzi 00001_{due}, 01001_{due}, 10001_{due} e 11001_{due} corrispondono tutti alla cella 001_{due} della cache, mentre gli indirizzi 00101_{due}, 01101_{due}, 10101_{due} e 11101_{due} corrispondono alla cella 101_{due}.

$\log_2$(dimensione della cache in blocchi) bit; in altre parole si può accedere alla cache direttamente usando i bit meno significativi. Per esempio, la figura 7.5 riporta una cache a corrispondenza diretta di otto elementi e mostra gli indirizzi di memoria compresi tra 1_{dieci} (00001_{due}) e 29_{dieci} (11101_{due}) che corrispondono alle locazioni 1_{dieci} (001_{due}) e 5_{dieci} (101_{due}) nella cache.

Poiché ciascuna locazione della cache può contenere il valore di diverse locazioni di memoria, come si può sapere se il dato presente nella cache corrisponde alla parola richiesta? Ossia, come si può sapere se una data parola è nella cache? Si può risolvere questo problema aggiungendo alla cache un insieme di *tag* (ossia di indicatori di proprietà). I campi tag contengono l'informazione sull'indirizzo necessaria ad determinare se una parola nella cache corrisponde a quella richiesta; il tag deve contenere solo la parte più significativa dell'indirizzo, quella che non corrisponde ai bit utilizzati come indice nella cache. Questa regola è giustificata dal fatto che i bit corrispondenti all'indice sono utilizzati per scegliere un'unica cella corrispondente all'indirizzo di partenza, per cui è solamente necessario garantire che la parte più significativa dell'indirizzo corrisponda al tag. Per esempio, in figura 7.5 servono solo 2 dei 5 bit come tag, poiché i 3 bit meno significativi dell'indirizzo determinano il blocco.

Sarà anche necessario un meccanismo per riconoscere i blocchi della cache che non contengono informazioni valide; ad esempio quando il processore viene avviato la cache sarà vuota ed il valore dei campi tag non sarà significativo. Anche dopo l'esecuzione di molte istruzioni alcuni degli elementi della cache potrebbero essere

ancora vuoti, come in figura 7.4. È necessario sapere che il valore dei tag deve essere ignorato per tali celle: il metodo più comune è quello di aggiungere un *bit di validità* per indicare se una cella contiene un indirizzo valido. Se il bit non vale 1, il blocco non sarà considerato corrispondente ad alcun indirizzo.

Nel seguito di questo paragrafo ci si concentrerà sull'esame del funzionamento delle operazioni di lettura e di scrittura nella cache e della logica di controllo per la lettura. In generale la gestione della lettura è lievemente più semplice di quella della scrittura perché in lettura non occorre modificare il contenuto della memoria. Dopo un esame dei meccanismi fondamentali per la lettura e per la gestione dei miss, si esamineranno i progetti delle cache per due calcolatori reali e si descriverà come tali cache trattano le scritture.

Accesso ad una cache

La figura 7.6 mostra il contenuto di una cache a corrispondenza diretta di otto parole mentre risponde ad una serie di richieste da parte del processore. Avendo la cache otto blocchi, i 3 bit meno significativi dell'indirizzo forniscono il numero del blocco. La tabella seguente riporta l'azione intrapresa in seguito a ciascun accesso alla memoria:

Indirizzo decimale del riferimento	Indirizzo binario del riferimento	Hit o miss nella cache	Blocco della cache corrispondente (in cui il dato è stato trovato o posto)
22	10110_{due}	miss (7.6b)	(10110_{due} mod 8) = 110_{due}
26	11010_{due}	miss (7.6c)	(11010_{due} mod 8) = 010_{due}
22	10110_{due}	hit	(10110_{due} mod 8) = 110_{due}
26	11010_{due}	hit	(11010_{due} mod 8) = 010_{due}
16	10000_{due}	miss (7.6d)	(10000_{due} mod 8) = 000_{due}
3	00011_{due}	miss (7.6e)	(00011_{due} mod 8) = 011_{due}
16	10000_{due}	hit	(10000_{due} mod 8) = 000_{due}
18	10010_{due}	miss (7.6f)	(10010_{due} mod 8) = 010_{due}

Quando la parola di indirizzo 18 (10010_{due}) è portata nel blocco 2 (010_{due}) della cache, la parola dell'indirizzo 26 (11010_{due}), che si trovava nel blocco 2 (010_{due}), deve essere sostituita dai dati appena richiesti. Questo comportamento permette alla cache di trarre beneficio dalla località temporale: le parole richieste più di recente sostituiscono quelle a cui si è fatto riferimento meno di recente. La situazione è del tutto analoga all'aver bisogno di un libro dagli scaffali quando non c'è più spazio sulla scrivania: uno dei libri sulla scrivania deve essere riposto negli scaffali. In una cache a corrispondenza diretta vi è un solo posto in cui porre il dato appena richiesto, quindi vi è una sola possibile scelta a riguardo di che cosa debba essere sostituito.

A questo punto è chiaro dove deve essere ricercato nella cache un generico indirizzo: i bit di ordine inferiore dell'indirizzo sono utilizzabili per trovare l'unica cella della cache a cui tale indirizzo può corrispondere. La figura 7.7 mostra come un indirizzo richiesto sia suddiviso in:

- un indice della cache, usato per selezionare il blocco
- un campo tag, usato per confrontare il valore del campo tag presente nella cache.

Indice	V	Tag	Dato
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. Lo stato iniziale della cache all'accensione

Indice	V	Tag	Dato
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	S	10_{due}	Memoria(10110_{due})
111	N		

b. Dopo aver risolto il miss dell'indirizzo 10110_{due}

Indice	V	Tag	Dato
000	N		
001	N		
010	S	11_{due}	Memoria(11010_{due})
011	N		
100	N		
101	N		
110	S	10_{due}	Memoria(10110_{due})
111	N		

c. Dopo aver risolto il miss dell'indirizzo 11010_{due}

Indice	V	Tag	Dato
000	S	10_{due}	Memoria(10000_{due})
001	N		
010	S	11_{due}	Memoria(11010_{due})
011	N		
100	N		
101	N		
110	S	10_{due}	Memoria(10110_{due})
111	N		

d. Dopo aver risolto il miss dell'indirizzo 10000_{due}

Indice	V	Tag	Dato
000	S	10_{due}	Memoria(10000_{due})
001	N		
010	S	11_{due}	Memoria(11010_{due})
011	S	00_{due}	Memoria(00011_{due})
100	N		
101	N		
110	S	10_{due}	Memoria(10110_{due})
111	N		

e. Dopo aver risolto il miss dell'indirizzo 00011_{due}

Indice	V	Tag	Dato
000	S	10_{due}	Memoria(10000_{due})
001	N		
010	S	10_{due}	Memoria(10010_{due})
011	S	00_{due}	Memoria(00011_{due})
100	N		
101	N		
110	S	10_{due}	Memoria(10110_{due})
111	N		

f. Dopo aver risolto il miss dell'indirizzo 10010_{due}

Figura 7.6 Viene riportato il contenuto della cache dopo ciascuna richiesta che causa un miss, dove i campi indice e tag sono indicati in binario. La cache inizialmente è vuota, con tutti i bit di validità (V) non attivi (N). Il processore richiede i seguenti indirizzi: 10110_{due} (miss), 11010_{due} (miss), 10110_{due} (hit), 11010_{due} (hit), 10000_{due} (miss), 00011_{due} (miss), 10000_{due} (hit), 10010_{due} (miss). Le figure mostrano il contenuto della cache dopo che ciascun miss nella sequenza è stato risolto. Quando si fa riferimento all'indirizzo 10010_{due} (18) occorre sostituire la cella che si riferiva all'indirizzo 11010_{due} (26), per cui un successivo riferimento a 11010_{due} causerà un miss. Il campo tag contiene solo la parte più significativa dell'indirizzo, per cui in questa cache l'indirizzo completo di una parola contenuta nel blocco i della cache con campo tag pari a j sarà pari a $j \times 8 + i$, o equivalentemente alla concatenazione del campo tag j con l'indice i. Per esempio nella cache f della figura la posizione di indice 010 ha tag pari a 10 e quindi corrisponde all'indirizzo 10110.

Poiché un dato indirizzo può comparire in una sola locazione, il campo tag può corrispondere solo alla parte più significativa dell'indirizzo, non usata per indirizzare la cache. Quindi l'indice di un blocco della cache, unitamente al contenuto del campo tag di tale blocco, specifica in modo univoco l'indirizzo di memoria della parola contenuta nel blocco della cache. Poiché il campo indice è utilizzato come indirizzo per accedere alla cache e poiché un numero di n bit può avere 2^n diversi valori, il numero totale di celle nella cache deve essere una potenza di due.

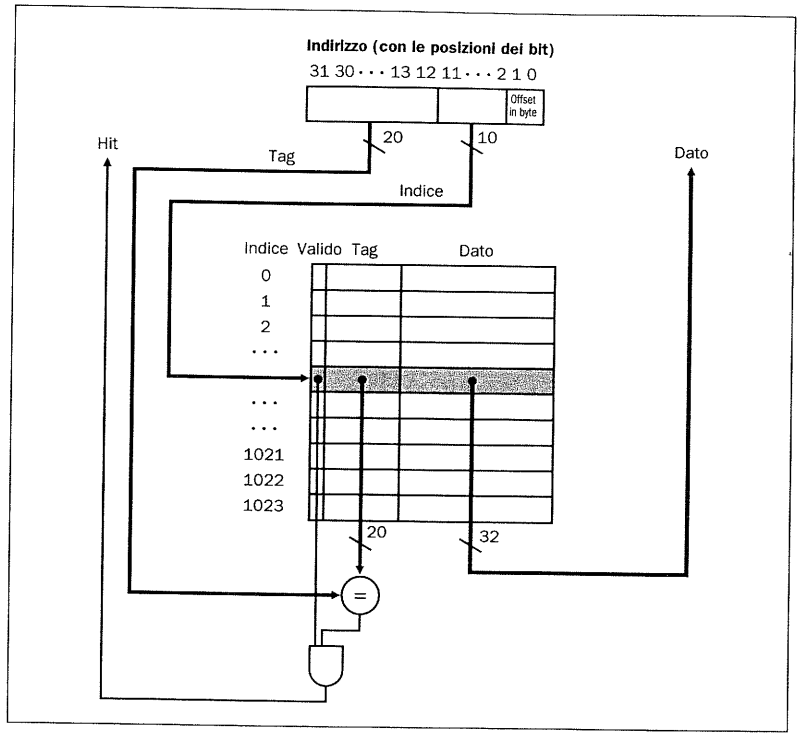


Figura 7.7 In questa cache la parte inferiore dell'indirizzo è utilizzata per selezionare una cella della cache, che contiene una parola di dato ed un campo tag. Il campo tag della cache è confrontato con la parte superiore dell'indirizzo per determinare se i dati nella cache corrispondono all'indirizzo richiesto. Poiché la cache ha 2^{10} (o 1024) parole ed una dimensione di blocco pari ad una parola, sono necessari 10 bit per indirizzare la cache, lasciando 32 – 10 – 2 = 20 bit per i confronti con il campo tag. Se il tag ed i 20 bit più significativi dell'indirizzo coincidono e se il bit di validità vale 1, allora la richiesta è un hit ed il dato viene prelevato dalla cache, fornendo la parola al processore. In caso contrario si verifica un miss.

Nell'architettura MIPS i due bit meno significativi di un indirizzo specificano il byte all'interno della parola e non sono usati per determinare una parola nella cache.

Il numero totale di bit richiesti per una cache è funzione della dimensione della cache e della lunghezza degli indirizzi, poiché la cache memorizza sia i dati che i campi tag. Assumendo di lavorare con indirizzi di 32 bit, una cache a corrispondenza diretta di capacità pari a 2^n parole e con blocchi di una parola (4 byte) richiederà un campo tag di ampiezza pari a $32 - (n + 2)$ bit, poiché 2 bit sono usati per l'offset del byte e n bit per l'indice. Il numero totale di bit in una cache a corrispondenza diretta è quindi $2^n \times$ (dimensione del blocco + dimensione del tag + dimensione del bit di validità). Con una dimensione di blocco pari ad una parola (32 bit) ed avendo indirizzi di 32 bit, il numero di bit nella cache è pari a $2^n \times (32 + (32 - n - 2) + 1) = 2^n \times (63 - n)$.

Numero di bit in una cache

Esempio Qual è il numero totale di bit richiesti per una cache a corrispondenza diretta contenente 64 KB di dati e blocchi di una parola, ipotizzando indirizzi di 32 bit?

Soluzione Sapendo che 64 KB equivalgono a 16K parole, ossia 2^{14} parole, essendo la dimensione del blocco pari ad una parola, si avranno 2^{14} blocchi. Ciascun blocco contiene 32 bit per il dato più il campo tag, che è di 32 – 14 – 2 bit, oltre ad un bit di validità. La dimensione totale della cache è quindi pari a:

$$2^{14} \times (32 + (32 - 14 - 2) + 1) = 2^{14} \times 49 = 784 \times 2^{10} = 784 \text{ Kbit}$$

corrispondente a 98 KB per una cache di 64 KB. In questa cache il numero totale di bit è pari ad 1,5 volte quello necessario alla memorizzazione dei soli dati.

Gestione dei fallimenti di accesso (miss) alla cache

Prima di passare ad esaminare il funzionamento della cache in un sistema reale, si esaminerà come l'unità di controllo deve trattare le situazioni di fallimento di negli accessi (miss): l'unità di controllo deve rilevare il fallimento e gestirlo prelevando i dati dalla memoria (o da una cache di livello inferiore). Se la cache segnala che si è verificato un hit, il sistema continua utilizzando il dato come se nulla fosse successo. Di conseguenza, si potrà usare la stessa unità di controllo elementare sviluppata nel Capitolo 5 ed estesa nel Capitolo 6 per gestire le pipeline. Le memorie presenti nell'unità di elaborazione vengono semplicemente sostituite dalle corrispondenti cache.

La modifica dell'unità di controllo per gestire le situazioni di hit è banale, mentre quelle di miss richiedono del lavoro aggiuntivo; l'approccio più semplice prevede di mettere in stallo la CPU, congelando il contenuto di tutti i registri. Un'unità di controllo separata gestisce i miss nella cache, prelevando i dati dalla memoria e portandoli nella cache: appena i dati sono presenti, si riprende l'esecuzione a partire esattamente dal ciclo che aveva causato la condizione di miss. La gestione dei miss nella cache avviene grazie all'unità di controllo e ad un'unità di controllo aggiuntiva che attiva l'accesso alla memoria e riempie la cache. Il verificarsi di un miss crea uno stallo, simile agli stalli della pipeline visti nel Capitolo 6, diversamente da un'interruzione, che richiederebbe di salvare lo stato di tutti i registri. In seguito ad un miss nella cache si può mandare in stallo l'intero processore, in pratica congelando il contenuto di tutti i registri temporanei e di quelli visibili al programmatore, in attesa della risposta da parte della memoria. Se si confronta questa situazione con gli stalli della pipeline discussi nel Capitolo 6, si vede che quelli erano più complessi poiché occorreva continuare l'esecuzione di alcune istruzioni mentre altre venivano poste in stallo.

Si esaminerà ora in maggior dettaglio la gestione dei casi di miss nella lettura delle istruzioni da parte dell'unità di elaborazione multi-ciclo o da quella con pipeline; è possibile estendere lo stesso approccio anche alla gestione dei miss nella cache dei dati. Se l'accesso ad un'istruzione causa il verificarsi di un miss, il valore caricato nel registro delle istruzioni non è quello corretto; per portare all'interno della cache l'istruzione corretta si deve chiedere al livello inferiore nella gerarchia

delle memorie di compiere una lettura. Poiché sia nell'unità di elaborazione dotata di pipeline che in quella multi-ciclo il program counter viene incrementato durante il primo ciclo di clock, l'indirizzo dell'istruzione responsabile del miss nella cache delle istruzioni è pari al valore del contatore di programma diminuito di quattro. Tale valore può essere calcolato facendo uso della ALU, anche se ciò richiederà dei registri temporanei aggiuntivi nell'implementazione con pipeline. Non appena l'indirizzo sarà disponibile, si potrà avviare una lettura dalla memoria principale; si attenderà la risposta della memoria (in quanto l'accesso richiederà diversi cicli) ed infine si potrà scrivere la parola all'interno della cache.

I passi da intraprendere al verificarsi di un miss nella cache delle istruzioni sono quindi definiti come segue:

1. Inviare il valore originario del registro PC (ossia l'attuale PC meno 4) alla memoria.
2. Ordinare alla memoria principale di effettuare un'operazione di lettura ed attendere fino a che la memoria non avrà completato l'operazione di accesso.
3. Scrivere l'elemento nella cache, ponendo i dati provenienti dalla memoria nella parte dati dell'elemento ed i bit più significativi dell'indirizzo (provenienti dalla ALU) nel campo tag, e portando ad uno il bit di validità.
4. Riavviare l'esecuzione dell'istruzione a partire dal primo passo, in cui verrà ripetuto il prelievo dell'istruzione, questa volta presente nella cache.

Il controllo della cache nel caso degli accessi ai dati è essenzialmente identico: quando avviene un miss si pone semplicemente in stallo il processore finché la memoria non risponde con il dato richiesto. Nel seguito di questo paragrafo verranno descritte due diverse cache di elaboratori reali e si esaminerà come esse gestiscano le operazioni di lettura e scrittura. Nel Paragrafo 7.5 verranno forniti maggiori dettagli sulla gestione delle operazioni di scrittura.

Approfondimento: Per ridurre il costo dei miss nella cache i progettisti possono adottare due tecniche, la prima delle quali viene discussa ora, mentre la seconda verrà trattata più avanti. Per ridurre il numero dei cicli nei quali il processore è in stato di stallo a causa di un miss nella cache, gli si può permettere di continuare ad eseguire altre istruzioni mentre viene gestito il miss. Questa strategia non è adatta ai miss delle istruzioni in quanto non si possono prelevare nuove istruzioni da eseguire; tuttavia, nel caso dei miss dei dati, si può permettere al processore di continuare il prelievo e l'esecuzione di nuove istruzioni finché non si renderà necessario utilizzare il valore della parola di cui si era tentato il caricamento; tale tecnica è quindi denominata *stallo all'utilizzo* (*stall on use*). Questo sforzo aggiuntivo può far risparmiare dei cicli, ma probabilmente non molti poiché il dato caricato sarà molto verosimilmente richiesto quasi subito e poiché anche le altre istruzioni dovranno fare accesso alla cache. I processori moderni utilizzano una tecnica più sofisticata che verrà descritta nel Paragrafo 7.6.

Un esempio di cache: la DECStation 3100

La DECStation 3100 era una workstation che usava un processore MIPS R2000 ed un'implementazione della cache molto semplice. Verso la fine del capitolo si esaminerà il progetto di una cache per un elaboratore più recente, ma per ragioni didattiche conviene partire da questo esempio che, pur essendo reale, è più semplice.

Il processore è dotato di un'architettura a pipeline simile a quella esaminata nel Capitolo 6: quando lavora alla massima velocità, ad ogni ciclo di clock richiede una

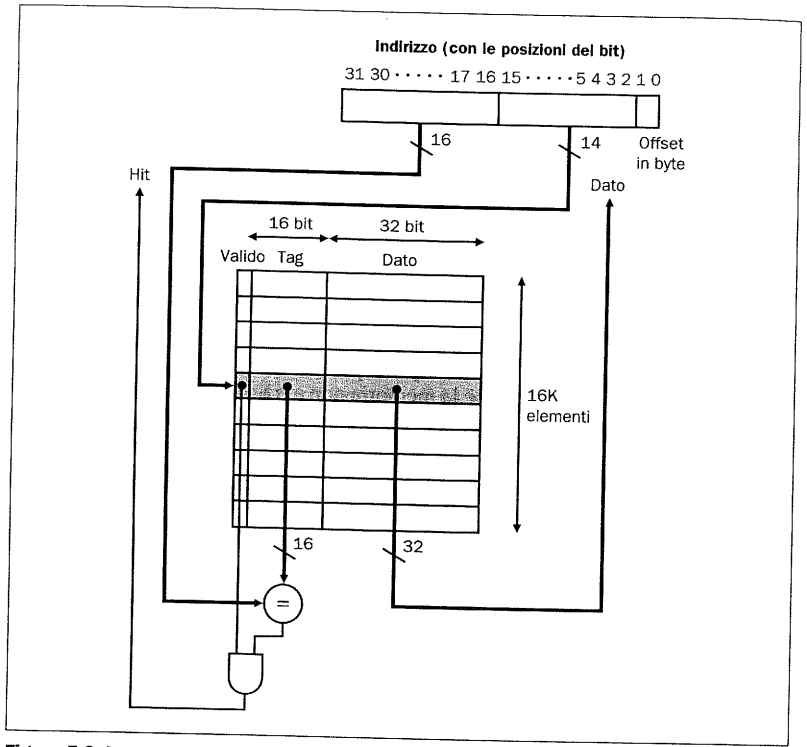


Figura 7.8 Le cache della DECStation 3100 contengono 16K blocchi ciascuna, con una parola per blocco. Di conseguenza l'indice ha dimensione pari a 14 bit ed il campo tag contiene 16 bit.

parola relativa ad un'istruzione e una parola di dato. Per soddisfare le richieste della pipeline senza introdurre stalli sono utilizzate due cache separate per le istruzioni e per i dati. Ciascuna cache è ha dimensione pari a 64 KB, ossia 16K parole, organizzata in blocchi di una parola. La figura 7.8 mostra l'organizzazione della cache dei dati della DECStation 3100.

Le richieste di lettura dalla cache sono gestite in modo lineare: essendoci cache separate per istruzioni e dati serviranno segnali di controllo distinti per leggere e scrivere da ciascuna cache (si ricordi che occorre scrivere anche nella cache delle istruzioni, ad esempio a seguito di un miss). I passi da seguire per una richiesta di lettura in una delle cache sono quindi i seguenti:

1. Inviare l'indirizzo alla cache opportuna. L'indirizzo proviene dal registro PC (per la lettura di un'istruzione) o dalla ALU (negli accessi ai dati).
2. Se la cache indica che si è verificato un hit la parola richiesta è disponibile sulle linee di dato, mentre se indica che si è verificato un miss si invia l'indirizzo alla memoria principale. Quando la memoria restituisce il dato, questo viene scritto nella cache.

Le operazioni di scrittura procedono in un modo leggermente diverso. Si supponga che a seguito di un'istruzione store il dato venga scritto solamente nella cache dei dati (senza modificare la memoria principale): in tal caso, dopo la scrittura nella cache la memoria conterrebbe un valore diverso da quello della cache, e la cache e la memoria sarebbero *incoerenti*. Il modo più semplice per mantenere la coerenza tra la memoria principale e la cache è di scrivere sempre i dati sia nella memoria che nella cache. Questa tecnica, utilizzata dalla DECStation 3100, è detta *write-through* (ossia scrittura attraverso [la cache], in quanto il dato "attraversa" la cache e viene scritto anche in memoria).

Il secondo aspetto importante nelle operazioni di scrittura riguarda ciò che avviene quando si verifica un miss durante una scrittura. Essendo la parola di dato scritta dal processore non vi è ragione per leggere una parola dalla memoria: verrebbe immediatamente sovrascritta dal processore. Infatti in questa semplice cache si può sempre semplicemente scrivere la parola nella cache, aggiornando opportunamente i campi di dato e tag. Non è necessario distinguere se il dato fosse presente o meno nella cache; questa osservazione porta al seguente semplice metodo di gestione delle operazioni di scrittura, usato anche sulla DECStation 3100:

1. Accedere alla cache usando come indice i bit 15-2 dell'indirizzo.
2. Scrivere i bit 31-16 dell'indirizzo nel campo tag, scrivere la parola di dato nella porzione di dato e forzare ad uno il bit di validità.
3. Scrivere altresì la parola nella memoria principale utilizzando l'indirizzo completo.

Questo progetto tratta le operazioni di scrittura in un modo molto semplice, ma non fornisce buone prestazioni: con una tecnica di tipo write-through tutte le operazioni di scrittura implicano la scrittura dei dati nella memoria principale, il che richiede molto tempo e potrebbe rallentare notevolmente il sistema. Ad esempio nel programma gcc il 13% delle istruzioni sono di tipo store. Nella DECStation 3100, se non vi fossero dei miss nella cache, il parametro CPI per un programma come gcc varrebbe circa 1,2. Se fossero richiesti 10 cicli per ciascuna scrittura il valore di CPI risulterebbe pari a $1{,}2 + 10 \times 13\% = 2{,}5$: le prestazioni peggiorerebbero di un fattore maggiore di due.

Una possibile soluzione a questo problema è l'adozione di un *buffer delle scritture*: esso memorizza i dati in attesa di essere scritti nella memoria. Dopo aver scritto il dato nella cache e nel buffer delle scritture, il processore può continuare immediatamente l'esecuzione; al completamento di un'operazione di scrittura nella memoria centrale viene liberata una posizione del buffer delle scritture. Se tale buffer è già pieno quando il processore giunge ad un'istruzione di scrittura occorrerà porre il processore in stallo finché non si libererà almeno una posizione nel buffer. Chiaramente se la velocità di completamento delle operazioni di scrittura da parte della memoria fosse minore del tasso a cui il processore genera le richieste di scrittura, non esisterebbe un buffer tanto grande da evitare gli stalli, poiché le scritture verrebbero generate più velocemente di quanto il sottosistema di memoria sia in grado di accettare.

Se anche la frequenza di generazione delle scritture fosse *minore* della frequenza con la quale la memoria le può accettare, si potrebbero ancora verificare alcuni stalli: ciò avviene quando le scritture sono richieste in rapida successione. Per ridurre la frequenza di tali stalli, si aumenta la capienza del buffer delle scritture, rendendolo capace di memorizzare più di un dato da scrivere. Ad esempio, il buffer delle scrit-

Programma	Frequenza di miss per le istruzioni	Frequenza di miss per i dati	Frequenza di miss combinata
gcc	6,1%	2,1%	5,4%
spice	1,2%	1,3%	1,2%

Figura 7.9 Frequenze di miss per le istruzioni e per i dati nella DECStation 3100 durante l'esecuzione di due programmi diversi. La frequenza di miss combinata è la frequenza effettiva dei miss relativa alla combinazione della cache delle istruzioni di 64 KB e della cache dei dati di 64 KB. Il suo valore è ottenuto pesando le frequenze di miss delle istruzioni e dei dati con la frequenza delle istruzioni e dei riferimenti ai dati. Si ricordi che i miss nei dati tengono conto solo delle operazioni di lettura dei dati, poiché nella cache della DECStation 3100 le operazioni di scrittura non possono causare dei miss.

ture della DECStation 3100 può contenere quattro parole. All'aumentare della differenza tra la frequenza a cui i programmi generano le operazioni di scrittura e la frequenza a cui il sottosistema di memoria è in grado di accettarle, potrebbero essere opportuni anche buffer delle scritture più capienti (ad esempio di 10 elementi).

Un'alternativa alla metodologia del write-through prende il nome di *write-back* (scrittura a posteriori): secondo tale tecnica, quando avviene una scrittura il nuovo valore viene scritto solamente nel blocco della cache. Il blocco modificato verrà ricopiato nel livello gerarchico inferiore solo quando dovrà essere sostituito. Gli schemi di tipo write-back possono migliorare le prestazioni soprattutto quando i processori sono in grado di generare le operazioni di scrittura ad una velocità pari o superiore a quella richiesta dalla memoria per gestirle; tuttavia la tecnica write-back presenta una maggior complessità implementativa rispetto alla write-through.

Quali valori di frequenza di miss ci si può attendere da una struttura delle cache come quella usata dalla DECStation 3100? La figura 7.9 mostra le frequenze di miss per le cache dei dati e delle istruzioni in due programmi già noti dai capitoli precedenti. La frequenza di miss combinata è la frequenza di miss effettiva per i riferimenti alla memoria tenuto conto della diversa frequenza delle istruzioni e dei riferimenti ai dati.

Si ricordi che, sebbene la frequenza dei miss sia un parametro importante nel progetto delle cache, l'unica misura definitiva è l'impatto del sottosistema di memoria sul tempo di esecuzione del programma; tra breve si analizzerà la relazione tra frequenza di miss e tempo di esecuzione, ma prima di ciò occorre valutare come il sottosistema di memoria possa trarre vantaggio dalla località spaziale.

Approfondimento: Una cache unica di dimensioni totali pari alla somma delle due cache separate ha generalmente una migliore frequenza di hit. Tale maggior frequenza è dovuta al fatto che la cache non divide in modo rigido i propri elementi tra quelli che possono essere usati dalle istruzioni e quelli per i dati. Nonostante ciò molti sistemi usano cache delle istruzioni e dei dati divise per aumentare la *larghezza di banda* verso la cache.

I seguenti dati derivano da misurazioni eseguite sul il programma gcc in esecuzione sulla DECStation 3100, nel caso di una cache combinata di dimensione pari alla somma delle due cache della 3100:

- Dimensione totale della cache: 128 KB
- Frequenza di miss effettiva delle cache separate: 5,4%
- Frequenza di miss della cache combinata: 4,8%

La frequenza di miss delle cache separate è solo lievemente peggiore.

In molti elaboratori il vantaggio di raddoppiare la larghezza di banda verso le cache, permettendo l'accesso simultaneo ad un'istruzione ed a un dato, supera decisamente lo svantaggio di una frequenza di miss lievemente maggiore. Questa osservazione è un ulteriore monito a non utilizzare la frequenza di miss come unico indicatore delle prestazioni delle cache.

Come trarre vantaggio dalla località spaziale

La cache descritta finora, nella sua semplicità, non faceva nulla per trarre vantaggio dalla località spaziale delle richieste, poiché ciascuna parola apparteneva ad un blocco diverso. Come si era notato nel Paragrafo 7.1 la località spaziale emerge spontaneamente nei programmi; per trarne vantaggio è però necessario che i blocchi della cache siano più ampi di una sola parola. Quando si verifica un miss, si leggeranno quindi diverse parole adiacenti, che hanno un'elevata probabilità di essere richieste nel prossimo futuro. La figura 7.10 riporta una cache che contiene 64 KB di dati organizzata in blocchi di quattro parole (16 byte) ciascuno. Confrontata con la figura 7.8, che mostra una cache delle stesse dimensioni con blocchi di una parola, nella cache di figura 7.10 l'indirizzo ha un campo aggiuntivo per indicizzare il blocco. Il campo "Offset nel blocco" viene utilizzato per controllare il multiplexer (in basso nella figura) che seleziona la parola richiesta tra le quattro del blocco prescelto. Il numero totale di campi tag e di bit di validità è inferiore nella cache con blocchi di più parole, poiché ciascuno di essi viene usato per quattro parole. La condivisione dei tag migliora l'efficienza di utilizzo della memoria nelle cache.

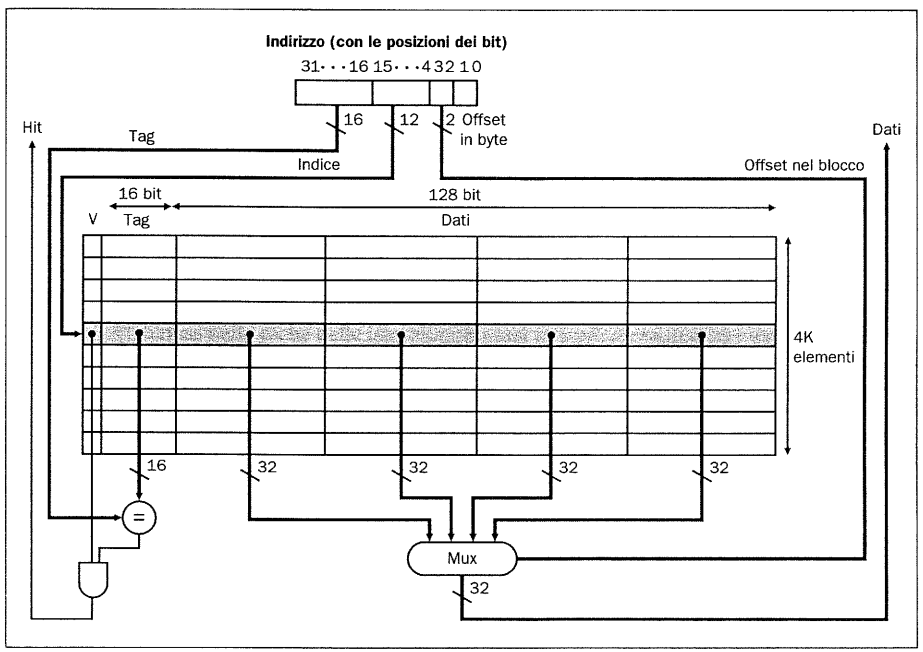


Figura 7.10 Una cache di 64 KB che utilizza blocchi di quattro parole (16 byte). Il campo tag è ampio 16 bit ed il campo indice è ampio 12 bit, mentre si usa un campo di 2 bit (i bit 3 e 2) per indicizzare il blocco e per selezionare la parola all'interno del blocco usando un multiplexer a 4 vie. In pratica i bit meno significativi dell'indirizzo (i bit 2 e 3 in questo caso) sono usati per abilitare solo quelle RAM che contengono la parola voluta, evitando così di dover utilizzare un multiplexer. Un secondo modo di eliminare il multiplexer è disporre di un'ampia memoria RAM per i dati (memorizzando i campi tag altrove) ed usare l'offset nel blocco per fornire alla RAM 2 bit di indirizzo; la RAM deve avere ampiezza pari a 32 bit ed un numero di parole pari al quadruplo del numero di blocchi della cache.

Come si ricava il blocco di cache relativo ad un dato indirizzo? Si può usare lo stesso metodo che si era utilizzato nella cache con blocchi di una parola:

(indirizzo del blocco) modulo (numero di blocchi nella cache)

L'indirizzo del blocco è semplicemente l'indirizzo espresso in parole diviso per il numero di parole nel blocco (o, equivalentemente, l'indirizzo espresso in byte diviso per il numero di byte in ciascun blocco).

Corrispondenza tra indirizzi e blocchi nelle cache a più parole per blocco

Esempio Si consideri una cache con 64 blocchi ed una dimensione dei blocchi di 16 byte. A quale numero di blocco corrisponde l'indirizzo del byte 1200?

Soluzione Il numero del blocco è fornito da

(indirizzo del blocco) modulo (numero di blocchi nella cache)

L'indirizzo del blocco è invece pari a

$$\frac{\text{indirizzo del byte}}{\text{byte per blocco}}$$

Si noti che il blocco con tale indirizzo contiene tutti gli indirizzi compresi tra

$$\left\lfloor \frac{\text{indirizzo del byte}}{\text{byte per blocco}} \right\rfloor \times \text{byte per blocco}$$

e

$$\left\lfloor \frac{\text{indirizzo del byte}}{\text{byte per blocco}} \right\rfloor \times \text{byte per blocco} + (\text{byte per blocco} - 1)$$

Disponendo di 16 byte per blocco, il byte di indirizzo 1200 si trova pertanto nel blocco di indirizzo

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

corrispondente al blocco della cache numero (75 modulo 64) = 11.

Le condizioni di miss nel caso delle letture sono trattate nello stesso modo sia per blocchi di più parole che per blocchi di una sola parola: quando avviene un miss si preleva sempre l'intero blocco. Durante le scritture, invece, gli hit ed i miss devono essere gestiti diversamente da quanto avveniva nella cache della DECStation 3100. Poiché il blocco contiene più di una parola, non è possibile scrivervi solamente il tag ed il dato. Per capirne il motivo, si ipotizzi che vi siano due

indirizzi di memoria, X ed Y, entrambi corrispondenti al blocco C della cache, il quale sia un blocco di quattro parole che attualmente contiene Y. Si consideri ora una scrittura all'indirizzo X che sovrascriva semplicemente il dato ed il campo tag all'interno del blocco della cache C: dopo la scrittura, il blocco C avrà il tag corrispondente ad X, ma i dati contenuti in C corrisponderanno ad una parola di X ed a tre parole di Y!

Nel caso di una cache di tipo write-through il problema può essere risolto compiendo un confronto dei tag mentre la scrittura è in corso, esattamente come avviene per le letture: se il tag dell'indirizzo coincide con quello della cache, si ha un hit nella scrittura, e si può procedere. Se i tag sono diversi, la scrittura ha provocato un miss e si deve prelevare il blocco dalla memoria. Dopo che il blocco è stato prelevato e posto nella cache, si può riscrivere la parola che ha causato il miss all'interno del blocco della cache. Diversamente dalla situazione che si presenta con blocchi di una parola, i miss nelle scritture richiedono di leggere la memoria.

La motivazione che spinge ad aumentare la dimensione del blocco è trarre vantaggio dalla località spaziale per aumentare le prestazioni: come valutare quindi l'impatto di un blocco più ampio sulle prestazioni? In generale la frequenza di miss si riduce all'aumentare della dimensione dei blocchi; tale andamento è più facilmente analizzabile mediante un esempio. Si ipotizzi che un programma richieda dei byte ai seguenti indirizzi: 16, ..., 24, ..., 20 e che nessuno di essi sia nella cache. Il principio di località spaziale ci dice che una sequenza di questo tipo è molto probabile, anche se l'ordine dei riferimenti potrebbe essere diverso. Le la cache avesse un blocco di quattro parole, il miss che avviene all'indirizzo 16 causerebbe il caricamento nella cache del blocco corrispondente alle parole di indirizzo 16, 20, 24 e 28. Per questi tre riferimenti si verificherà un solo miss, purché nessun riferimento intermedio elimini il blocco dalla cache. Se si fosse utilizzata una cache con blocchi di una parola, si sarebbero verificati altri due miss poiché ciascun miss poteva caricare nella cache una sola parola.

La figura 7.11 riporta le frequenze di miss per i programmi gcc e spice nel caso di cache con blocchi di una e di quattro parole. Il fattore di riduzione della frequenza di miss nella cache delle istruzioni è quasi pari al fattore di incremento nella dimensione del blocco; il motivo per cui la riduzione è più marcata per le istruzioni che per i dati è che i riferimenti alle istruzioni hanno una località spaziale maggiore.

La frequenza di miss potrebbe in realtà aumentare se la dimensione del blocco divenisse una frazione significativa della dimensione della cache, poiché si ridurrebbe il numero di blocchi presenti nella cache e nascerebbe una forte competizione tra i pochi blocchi. I blocchi tenderebbero quindi ad essere estromessi dalla cache

Programma	Dimensione del blocco in parole	Frequenza di miss delle istruzioni	Frequenza di miss dei dati	Frequenza di miss combinata
gcc	1	6,1%	2,1%	5,4%
	4	2,0%	1,7%	1,9%
spice	1	1,2%	1,3%	1,2%
	4	0,3%	0,6%	0,4%

Figura 7.11 Le frequenze di miss per i programmi gcc e spice in una cache simile a quella della DECStation 3100, con una dimensione dei blocchi di una e di quattro parole. Nel caso dei blocchi di quattro parole si sono conteggiati anche i miss nelle scritture, mentre con blocchi di una parola questi non introducono alcun rallentamento e quindi non sono stati conteggiati. Se si fossero conteggiati i miss nelle scritture per entrambi i casi, la differenza nelle frequenze di miss sarebbe stata leggermente più accentuata.

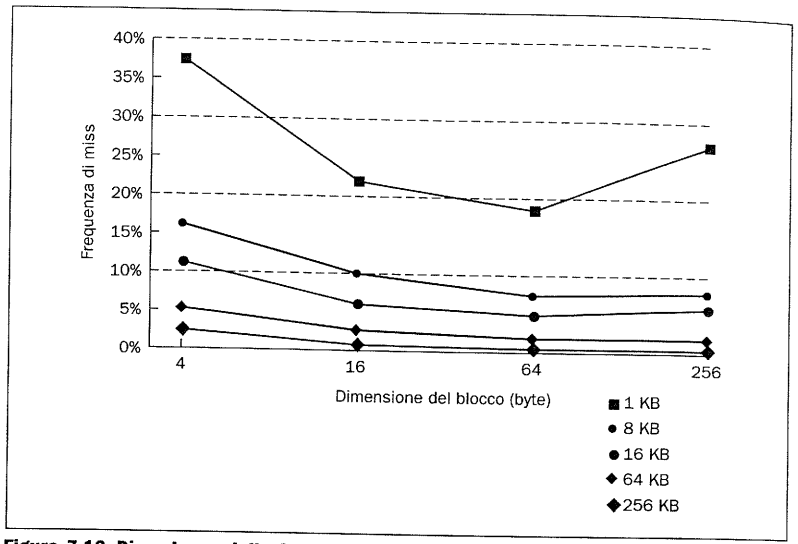


Figura 7.12 Dipendenza della frequenza di miss dalla dimensione dei blocchi. Per una piccola cache di 1 KB, una grande dimensione dei blocchi di 256 byte causa una frequenza di miss maggiore rispetto a blocchi più piccoli. I dati per la cache a corrispondenza diretta sono stati raccolti mediante l'analisi delle tracce di esecuzione (SAVE0) raccolta da Agarwal per il VAX. Per maggiori dettagli si faccia riferimento a A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Ph.D. thesis, Stanford Univ., Tech. Rep. No. CSL-TR-87-332 (Maggio 1987).

prima che avvenga un numero significativo di accessi alle parole in essi contenute. Come indicato in figura 7.12, l'aumento della dimensione dei blocchi tendenzialmente riduce la frequenza di miss; tuttavia la località spaziale tra le parole dello stesso blocco si riduce all'aumentare della sua dimensione, per cui il miglioramento della frequenza di miss diviene sempre più piccolo e prima o poi la frequenza di miss tenderà ad aumentare.

Un problema ancora più grave associato all'aumento della dimensione dei blocchi è l'aumento del costo dei miss: la penalità dovuta ad un miss è determinata dal tempo necessario a caricare il blocco dal livello immediatamente inferiore nella gerarchia delle memorie e scriverlo nella cache. Tale tempo è composto da due parti: la latenza per leggere la prima parola ed il tempo di trasferimento per il resto del blocco. È evidente che, a meno di modificare il sottosistema di memoria, il tempo di trasferimento (e quindi il costo di un miss) aumenta al crescere della dimensione del blocco. Inoltre il miglioramento della frequenza di miss diviene meno significativo quando i blocchi sono troppo grandi. Il risultato di tutto ciò è che l'aumento della penalità introdotta da un miss supera la riduzione della frequenza di miss quando i blocchi sono grandi, facendo così ridurre le prestazioni della cache. Come è ovvio, se si progettasse la memoria in modo da poter trasferire con maggior efficienza grandi blocchi, si potrebbe aumentare la dimensione dei blocchi e si potrebbero ottenere dei miglioramenti ulteriori nelle prestazioni della cache: questo aspetto sarà discusso nel prossimo paragrafo.

Approfondimento: Lo svantaggio maggiore dell'incremento della dimensione dei blocchi è l'aumento della penalità nel caso di miss nella cache. Mentre è difficile limitare le criticità dovute alla latenza, si può riuscire a nascondere una parte del tempo di trasferimento il modo da ridurre la penalità associata ai miss. Il modo più semplice per far ciò, denominato *ripresa immediata* (*early restart*), consiste nel riprendere l'esecuzione non appena nel blocco compare la parola richiesta, senza attendere che il caricamento del blocco venga completato. Questa tecnica viene utilizzata in molti sistemi per l'accesso alle istruzioni, dove fornisce risultati migliori vista la natura prevalentemente sequenziale delle istruzioni: se il sottosistema di memoria può fornire una nuova parola ad ogni ciclo di clock, il processore potrà riprendere l'esecuzione non appena è disponibile la parola richiesta, mentre le parole successive saranno fornite dal sottosistema di memoria nel momento esatto un cui serviranno. La stessa tecnica si rivela meno efficace nelle cache dei dati perché probabilmente le parole saranno richieste dal blocco in un ordine meno prevedibile e perché vi è un'alta probabilità che il processore richieda una parola da un diverso blocco della cache prima della fine del caricamento. Se il processore non può accedere alla cache dei dati perché vi è un trasferimento in corso, è costretto ad entrare in stallo.

Uno schema ancora più sofisticato prevede di organizzare la memoria in modo che la parola richiesta sia trasferita per prima dalla memoria alla cache. Il resto del blocco viene trasferito subito dopo, partendo dall'indirizzo che segue la parola richiesta e ricominciando poi dall'inizio del blocco. Questa tecnica, denominata *requested word first* o *critical word first* (la parola richiesta, o critica, per prima), può rivelarsi lievemente più veloce della ripresa immediata, ma presenta le stesse limitazioni.

Progetto di un sottosistema di memoria che supporti le cache

Le situazioni di fallimento negli accessi alla cache (miss) implicano un accesso alla memoria principale, costituita da DRAM; nel Paragrafo 7.1 si era visto che le DRAM sono progettate ponendo maggior attenzione alla densità rispetto al tempo di accesso. Anche se è difficile ridurre la latenza nella lettura della prima parola dalla memoria, si può ridurre la penalità associata ai miss aumentando la larghezza di banda dalla memoria alla cache. Si potranno così utilizzare blocchi più grandi lavorando con una penalità ancora accettabile, simile a quella che si riscontrerebbe con blocchi più piccoli.

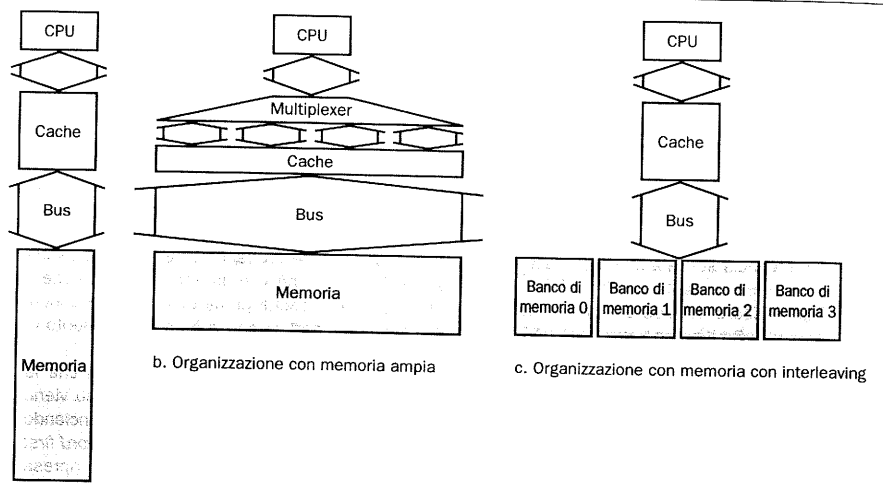
Per esaminare le implicazioni delle diverse organizzazioni della memoria, si definiscano dei tempi di accesso ipotetici:

- 1 ciclo di clock per inviare l'indirizzo
- 15 cicli di clock per ciascuna attivazione di un accesso alla DRAM
- 1 ciclo di clock per trasferire una parola di dato.

Con un blocco di cache di quattro parole ed un banco di DRAM ampio una parola, la penalità dei miss sarebbe pari a $1 + 4 \times 15 + 4 \times 1 = 65$ cicli di clock. Il numero di byte trasferiti per ciclo di clock nel caso si un singolo miss sarà pari a

$$\frac{4 \times 4}{65} = 0,25$$

La figura 7.13 riporta tre opzioni per il progetto del sottosistema di memoria: la prima opzione riassume quanto era stato ipotizzato finora, ossia la larghezza della memoria è pari ad una parola e tutti gli accessi avvengono sequenzialmente. La seconda opzione aumenta la larghezza di banda ampliando la memoria ed i bus tra il processore e la memoria; tale soluzione permette l'accesso in parallelo a tutte le parole di un blocco. La terza opzione aumenta la larghezza di banda ampliando la



a. Organizzazione con memoria ampia una parola

b. Organizzazione con memoria ampia

c. Organizzazione con memoria con interleaving

Figura 7.13 Il metodo principale per aumentare la larghezza di banda verso la memoria consiste nell'aumentare l'ampiezza fisica o logica del sottosistema di memoria. La figura riporta due diversi modi di aumentare la larghezza di banda verso la memoria. Il progetto più semplice (a) usa una memoria in cui tutti i componenti sono ampi una parola; in (b) sono stati ampliati la memoria, il bus e la cache; infine (c) mostra un bus ed una cache più stretti ma una memoria con interleaving. In (b) la logica tra la cache e la CPU consta di un multiplexer usato per le letture e di una logica di controllo per aggiornare le parole opportune della cache durante le operazioni di scrittura.

larghezza della memoria ma non quella del bus di interconnessione: si paga ancora un costo per trasmettere ciascuna parola, ma si evita di pagare più volte il costo della latenza. Si determinerà ora di quanto le ultime due opzioni migliorino la penalità di miss di 65 cicli vista per la prima opzione (figura 7.13a).

Aumentare l'ampiezza della memoria e del bus fa aumentare proporzionalmente la larghezza di banda, riducendo contemporaneamente il contributo del tempo di accesso e del tempo di trasferimento nel calcolo della penalità nel caso di miss. Con una memoria principale organizzata su un'ampiezza di due parole, la penalità di miss si ridurrebbe da 65 cicli di clock a $1 + 2 \times 15 + 2 \times 1 = 33$ cicli, mentre con un'ampiezza di quattro parole la penalità si ridurrebbe a 17 cicli. La larghezza di banda per un singolo miss è 0,48 byte per ciclo di clock in una memoria organizzata su due parole (quasi il doppio) e 0,94 byte per ciclo di clock con una memoria di ampiezza pari a quattro parole (quasi il quadruplo). I principali costi associati a questo miglioramento sono il bus più ampio ed il possibile aumento nel tempo di accesso alla cache dovuto al multiplexer ed alla logica di controllo tra la CPU e la cache.

Anziché rendere più ampio l'intero cammino tra la memoria e la cache, si potrebbero organizzare i chip di memoria in *banchi*, capaci di leggere o scrivere più parole nel tempo di un singolo accesso invece di leggere o scrivere una sola parola per volta. Ciascun banco potrebbe essere ampio una sola parola, in modo che non debba cambiare l'ampiezza del bus e della cache, ma l'invio di un indirizzo a più

banchi fa sì che questi effettuino contemporaneamente le operazioni di lettura. Questo schema, che prende il nome di *interleaving* (termine che ricorda l'alternanza degli indirizzi e che viene solitamente tradotto con *interlacciamento*), ha il vantaggio di incorrere una sola volta nella penalità dovuta alla latenza. Ad esempio con quattro banchi il tempo necessario a caricare un blocco di quattro parole consisterebbe in 1 ciclo per trasmettere l'indirizzo a tutti i banchi, 15 cicli affinché i banchi accedano alla memoria e 4 cicli per trasferire le 4 parole alla cache. Tutto ciò porta la penalità dei miss a $1 + 1 \times 15 + 4 \times 1 = 20$ cicli di clock, corrispondenti ad una larghezza di banda effettiva di 0,80 byte per clock, ossia circa il triplo della larghezza di banda di una memoria ed un bus ampi una sola parola. L'organizzazione a banchi è anche utile nelle operazioni di scrittura: ciascun banco può essere scritto indipendentemente, quadruplicando così la larghezza di banda in scrittura e portando ad un minore numero di stalli nelle cache di tipo write-through. Come si vedrà vi è una strategia alternativa per le scritture che rende ancora più interessanti le memorie con interleaving.

Approfondimento: Al crescere della capienza dei chip di memoria, il numero di chip richiesto decresce rapidamente a parità di dimensioni del sottosistema di memoria. I chip di memoria sono organizzati in modo da avere un piccolo numero di bit di uscita, solitamente da 1 a 16, dove 4 ed 8 erano le ampiezze più diffuse nel 1997. L'organizzazione di una DRAM è spesso descritta con la notazione $d \times w$, dove d è il numero di locazioni indirizzabili (la profondità, o depth) e w è il numero di uscite (l'ampiezza, o width, di ciascuna locazione). Le DRAM da 16 Mbit più diffuse sono di tipo 4M × 4. Al crescere della densità dei chip l'ampiezza rimane costante (o cresce molto lentamente), ma la profondità aumenta (si veda l'Appendice B per una discussione più approfondita sulle DRAM). Per questo motivo l'organizzazione a più banchi sta divenendo meno interessante, a causa dell'aumento della configurazione minima di memoria. Ad esempio una memoria principale di 64 MB costruita con chip di 4 Mbit × 1 può essere organizzata in quattro banchi, ciascuno ampio 32 bit, usando in totale 128 DRAM. Usando chip da 16 Mbit × 1 al posto di quelli da 4 Mbit × 1, la memoria di 64 MB potrà contenere un solo banco di ampiezza 32 bit, poiché l'intera memoria richiede solo 32 DRAM. Si sarebbero potuti ottenere i quattro banchi usando chip da 4 Mbit × 4 anziché 16 Mbit × 1; analogamente una memoria da 64 MB divisa in quattro banchi disponendo di chip di 64 Mbit richiederebbe DRAM da 4 Mbit × 16. La necessità di chip di memoria DRAM di ampiezza maggiore è il principale svantaggio dei banchi di memoria con interleaving. Una seconda possibilità per migliorare la velocità di trasferimento dalla memoria alla cache è di trarre vantaggio dalla struttura delle DRAM: esse sono organizzate logicamente come matrici quadrate ed il tempo di accesso è ripartito tra accesso alle righe ed accesso alle colonne. Ad ogni accesso le DRAM selezionano internamente un'intera riga di bit garantendo l'accesso a tutte le colonne; esse dispongono altresì di segnali di temporizzazione aggiuntivi che permettono accessi ripetuti alla riga memorizzata senza richiedere il tempo di accesso alle righe. Una versione diffusa di questa possibilità è detta *page mode* (modalità a pagine), che ha subito diversi miglioramenti, il più recente dei quali è detto EDO (Extended Data Out) RAM. Nella modalità a pagine, la riga memorizzata agisce esattamente come una SRAM: variando l'indirizzo di colonna si può accedere ai bit in posizioni arbitrarie, fino al prossimo accesso ad una riga successiva. Questa possibilità modifica significativamente il tempo di accesso, poiché il tempo per accedere ai bit nella stessa riga è molto minore. Ad esempio il tempo totale di accesso alle righe ed alle colonne senza modalità a pagine è pari a circa 120 ns, la modalità a pagine richiede 60 ns per accedere ai dati nella pagina, mentre la modalità EDO permette un accesso di 25 ns ai dati nella pagina. La figura 7.14 riporta le variazioni di densità, costo e tempo di accesso per le DRAM nel corso degli anni.

Gli sviluppi più recenti sono costituiti dalle SDRAM (*synchronous DRAM*, o DRAM sincrone): esse forniscono una modalità di accesso ai dati contenuti in una serie di locazioni consecutive della DRAM; tale modalità prende il nome di *burst*, termine inglese che ricorda uno

Anno di introduzione	Capacità del chip	$ per MB	Tempo di accesso totale ad una nuova riga/colonna	Tempo di accesso ad una colonna della riga corrente
1980	64 Kbit	1500	250 ns	150 ns
1983	256 Kbit	500	185 ns	100 ns
1985	1 Mbit	200	135 ns	40 ns
1989	4 Mbit	50	110 ns	40 ns
1992	16 Mbit	15	90 ns	30 ns
1996	64 Mbit	10	60 ns	20 ns

Figura 7.14 La capacità delle DRAM aumenta di circa quattro volte ogni tre anni. I miglioramenti nel tempo di accesso sono stati lenti ma costanti, mentre il costo tiene traccia dei miglioramenti nella densità, anche se il costo è talvolta legato ad altri parametri, come la disponibilità e la domanda di mercato. Il tempo di accesso alle colonne determina solitamente il tempo necessario per un accesso in modalità a pagine. Le DRAM sono quasi sempre disponibili dapprima nelle configurazioni di minor ampiezza (ad esempio 4 Mbit × 4), mentre le configurazioni più ampie (ad esempio 1 Mbit × 16) solitamente sono disponibili più tardi rispetto al prodotto di ampiezza minore, e costano di più. Le ragioni di ciò sono che il prodotto di ampiezza inferiore ha costi del collaudo e del package leggermente inferiori, è usato in sistemi a basse prestazioni ed ha una dimensione del die inferiore.

scoppio improvviso. Le SDRAM sono pilotate fornendo l'indirizzo iniziale e l'ampiezza del burst: il trasferimento dei dati interni al burst è controllato da un segnale di clock, che nel 1997 poteva arrivare a 100 MHz. Il due vantaggi principali delle SDRAM sono l'uso del clock, che semplifica la sincronizzazione, e l'eliminazione della necessità di fornire gli indirizzi successivi nel burst. La combinazione di questi vantaggi permette di abbassare il ritardo tra due bit successivi dai 25 ns delle EDO RAM agli 8-10 ns delle SDRAM.

Il vantaggio di queste ottimizzazioni è di sfruttare in larga misura i circuiti già presenti nella DRAM, incidendo di poco sul costo del sistema ma fornendo un significativo incremento nella larghezza di banda. Inoltre queste opzioni per le DRAM permettono di aumentare la larghezza di banda senza incorrere negli svantaggi a livello di sistema in termini di espandibilità e di dimensione minima della memoria associati a memorie più ampie o all'uso di memorie con interleaving. L'architettura interna delle DRAM e l'implementazione di queste ottimizzazioni sono descritte nel Paragrafo B.5 dell'Appendice B.

Riepilogo

Questo paragrafo ha inizialmente esaminato la più semplice organizzazione delle cache: una cache a corrispondenza diretta con blocchi di una parola. In tale cache sia le situazioni di hit che quelle di miss sono facilmente gestibili, poiché ciascuna parola può andare in una sola locazione e vi è un campo tag separato per ogni parola. Per garantire la coerenza tra cache e memoria si può usare la tecnica write-through, in modo tale che ogni scrittura nella cache causi anche un aggiornamento della memoria. In alternativa si può optare per uno schema write-back che copia il blocco nella memoria solo quando questo viene sostituito; si discuterà meglio quest'ultimo schema nei prossimi paragrafi.

Per trarre vantaggio dalla località spaziale i blocchi di una cache devono avere dimensioni maggiori di una parola; l'uso di un blocco più ampio riduce la frequenza dei miss e migliora l'efficienza delle cache riducendo la quantità di spazio dedicata alla memorizzazione dei tag rispetto alla memorizzazione dei dati. Anche se una maggiore dimensione del blocco può ridurre la frequenza dei miss, essa causa un aumento della penalità per i miss: se la penalità di miss aumentasse linearmente con la dimensione del blocco, blocchi più ampi potrebbero portare a prestazioni inferiori. Per evitare ciò si aumenta la larghezza di banda della memoria principale

per trasferire più efficientemente i blocchi della cache; i due metodi più utilizzati prevedono di rendere la memoria più ampia o di sfruttare la tecnica dell'interleaving. In entrambi i casi viene ridotto il tempo necessario a prelevare il blocco riducendo il numero di volte che occorre iniziare un accesso alla memoria durante il trasferimento del blocco; con un bus più ampio si riesce anche a ridurre il tempo necessario ad inviare il blocco dalla memoria alla cache.

7.3 Valutazione e miglioramento delle prestazioni delle cache

In questo paragrafo si inizierà ad analizzare come misurare ed analizzare le prestazioni delle cache; si procederà poi ad esplorare due diverse tecniche per il miglioramento delle prestazioni. La prima mira a ridurre la frequenza di miss riducendo la probabilità che due blocchi di memoria si contendano la stessa locazione nella cache; la seconda tecnica riduce la penalità associata ai miss aggiungendo un livello addizionale alla gerarchia. Tale tecnica, detta delle *cache a più livelli*, apparve la prima volta nei sistemi ad alte prestazioni che nel 1990 si vendevano a più di 200 milioni di lire e da allora è divenuta comune anche sui calcolatori del costo di meno di cinque milioni!

Il tempo di CPU può essere ripartito nei cicli di clock che la CPU passa ad eseguire il programma e nei cicli di clock che la CPU spende attendendo una risposta dal sottosistema di memoria. Solitamente si ipotizza che il costo degli accessi alla cache che si risolvono in un hit faccia parte dei cicli di esecuzione normali della CPU. Vale quindi la relazione seguente:

$$\text{tempo di CPU} = (\text{cicli di esecuzione della CPU} + \text{cicli di clock di stallo per la memoria}) \text{ periodo del clock}$$

I cicli di clock di stallo per la memoria derivano principalmente dalle situazioni di miss nella cache, e nel seguito dell'analisi lo si assumerà come ipotesi. Si restringerà inoltre l'attenzione ad un modello semplificato del sottosistema di memoria; nei processori reali gli stalli generati dalle letture e dalle scritture possono essere decisamente complessi, per cui una predizione accurata delle prestazioni richiede solitamente una simulazione molto dettagliata del processore e del sottosistema di memoria.

I cicli di clock di stallo per la memoria si possono definire come somma dei cicli di stallo derivanti dalle letture e di quelli causati dalle scritture:

$$\text{cicli di clock di stallo per la memoria} = \text{cicli di stallo in lettura} + \text{cicli di stallo in scrittura}$$

I cicli di stallo in lettura possono essere definiti in termini del numero di accessi in lettura del programma, della penalità per i miss durante le letture espressa in cicli di clock e della frequenza di miss in lettura:

$$\text{cicli di stallo in lettura} = \frac{\text{numero di letture}}{\text{programma}} \times \text{frequenza di miss in lettura} \times \text{penalità per i miss in lettura}$$

Le scritture sono più complesse; per uno schema write-through si hanno due tipi di stallo: i miss in scrittura, che solitamente richiedono di prelevare il blocco prima di continuare l'operazione di scrittura (si veda l'approfondimento del Paragrafo 7.5 per ulteriori dettagli sulla gestione delle scritture), e gli stalli sul buffer delle scritture, che avvengono quando il buffer delle scritture è pieno nel momento in cui dovrebbe avvenire la scrittura. Quindi i cicli di clock attribuibili agli stalli in scrittura sono la somma dei due contributi:

$$\text{cicli di stallo in scrittura} = \frac{\text{numero di scritture}}{\text{programma}} \times \text{frequenza di miss in scrittura}$$
$$\times \text{penalità per i miss in scrittura} + \text{stalli sul buffer delle scritture}$$

Poiché gli stalli sul buffer delle scritture dipendono dalla temporizzazione delle operazioni di scrittura e non solamente dalla loro frequenza, non è possibile fornire una semplice equazione per quantificarli. Fortunatamente in sistemi dotati di un'adeguata profondità del buffer delle scritture (ad esempio quattro o più parole) e di una memoria capace di accettare le scritture ad una frequenza significativamente superiore alla frequenza media con cui il programma effettua le operazioni di scrittura (ad esempio, di almeno un fattore due), gli stalli sul buffer delle scritture saranno pochi e brevi, e si potranno ignorare senza conseguenze. Un elaboratore che non dovesse soddisfare questi criteri non sarebbe ben progettato: il progettista avrebbe dovuto usare un buffer delle scritture più profondo oppure un'organizzazione di tipo write-back.

Anche gli schemi di tipo write-back sono fonte di potenziali stalli a causa della necessità di scrivere un blocco di cache nella memoria quando tale blocco viene sostituito; si discuterà questo fatto in maggior dettaglio nel Paragrafo 7.5.

Nella maggior parte dei sistemi con organizzazione di tipo write-through le penalità per i miss in lettura e per quelli in scrittura sono uguali (pari al tempo necessario a prelevare il blocco dalla memoria). Se si ipotizza che gli stalli sul buffer delle scritture siano trascurabili si possono combinare letture e scritture usando un'unica frequenza di miss e un'unica penalità di miss:

$$\text{cicli di clock di stallo per la memoria} = \frac{\text{numero di accessi in memoria}}{\text{programma}}$$
$$\times \text{frequenza di miss} \times \text{penalità per i miss}$$

La formula precedente si può anche scrivere come

$$\text{cicli di clock di stallo per la memoria} = \frac{\text{istruzioni}}{\text{programma}} \times \frac{\text{numero di miss}}{\text{istruzione}}$$
$$\times \text{penalità per i miss}$$

Verrà ora considerato un semplice esempio per facilitare la comprensione dell'impatto delle prestazioni della cache sulle prestazioni del calcolatore.

Calcolo delle prestazioni della cache

Esempio

Per il programma gcc si ipotizzi una frequenza di miss nella cache istruzioni pari al 2% ed una frequenza di miss nella cache dati pari al 4%. Se un elaboratore ha un valore di CPI pari a 2 in assenza di stalli per la memoria e la penalità di miss è pari a 40 cicli per tutti i tipi di miss, si determini di quanto sarebbe più veloce lo stesso calcolatore con una cache ideale in cui non si verificassero mai dei miss. Si usino le frequenze delle istruzioni del programma gcc riportate nella figura 4.54 del Capitolo 4.

Soluzione

Il numero di cicli dovuti ai miss per le istruzioni, in funzione del numero di istruzioni (I) è pari a

$$\text{cicli per i miss sulle istruzioni} = I \times 2\% \times 40 = 0{,}80 \times I$$

In gcc la frequenza complessiva delle letture e delle scritture è 36%, per cui si può calcolare il numero di cicli dovuti ai miss per gli accessi ai dati come segue:

$$\text{cicli per i miss sui dati} = I \times 36\% \times 4\% \times 40 = 0{,}56 \times I$$

Il numero totale dei cicli di stallo per la memoria è 0,80 I + 0,56 I = 1,36 I. Questo valore corrisponde a più di un ciclo di stallo per istruzione; di conseguenza il valore di CPI tenendo conto degli stalli in memoria sale a 2 + 1,36 = 3,36. Non essendoci alcuna variazione nel numero di istruzioni o nella frequenza di clock, il rapporto dei tempi di CPU è

$$\frac{\text{tempo di CPU con gli stalli}}{\text{tempo di CPU con cache ideale}} = \frac{I \times CPI_{stalli} \times \text{cicli di clock}}{I \times CPI_{ideali} \times \text{cicli di clock}} = \frac{CPI_{stalli}}{CPI_{ideali}} = \frac{3{,}36}{2}$$

Le prestazioni del sistema con la cache ideale sarebbero migliori di un fattore pari a $\frac{3{,}36}{2} = 1{,}68$.

Che cosa avverrebbe se il processore fosse più veloce, ma il sottosistema di memoria rimanesse invariato? La quantità di tempo spesa sugli stalli per la memoria consumerebbe una frazione sempre maggiore del tempo di esecuzione; ne è monito la legge di Amdahl, esaminata nel Capitolo 2; alcuni semplici esempi mostreranno quanto questo problema possa essere grave. Si supponga di velocizzare il sistema dell'esempio precedente riducendo il CPI da 2 ad 1 senza modificare la frequenza di clock, operazione possibile per mezzo di una pipeline migliore. Tenendo conto dei miss nella cache il sistema avrebbe un CPI pari ad 1 + 1,36 = 2,36 ed il sistema con la cache ideale sarebbe

$$\frac{2{,}36}{1} = 2{,}36 \text{ volte più veloce.}$$

La quantità di tempo di esecuzione speso per gli stalli sulla memoria passerebbe da

$$\frac{1,36}{3,36} = 41\%$$

a

$$\frac{1,36}{2,36} = 58\%$$

Analogamente aumentare la frequenza del clock senza modificare il sottosistema di memoria enfatizzerebbe la perdita di prestazioni dovuta ai miss nella cache, come testimoniato dall'esempio seguente.

Esempio

Prestazioni della cache al crescere della frequenza di clock

Si ipotizzi di aumentare le prestazioni del sistema dell'esempio precedente raddoppiandone la frequenza di clock. Poiché la velocità della memoria principale probabilmente sarà uguale, si assuma che non sia variata la quantità di tempo totale dovuta ai miss nella cache. Quanto sarà più veloce il sistema con il clock più veloce, assumendo la stessa frequenza di miss dell'esempio precedente?

Soluzione

Misurata con riferimento ai cicli di clock più veloci, la nuova penalità di miss avrà valore doppio, pari ad 80 cicli. Quindi

$$\text{cicli di miss totali per istruzione} = (2\% \times 80) + 36\% \times (4\% \times 80) = 2,75$$

Tenendo conto dei miss la macchina più veloce avrà CPI = 2+2,75 = 4,75, rispetto ad un valore di CPI di 3,36 comprensivo dei miss per la macchina più lenta.

Applicando la formula che fornisce il tempo di CPU dell'esempio precedente, si possono calcolare le prestazioni relative come segue:

$$\frac{\text{prestazioni con clock veloce}}{\text{prestazioni con clock lento}} = \frac{\text{tempo di esecuzione con clock lento}}{\text{tempo di esecuzione con clock veloce}} =$$

$$= \frac{I \times CPI \times \text{periodo di clock}}{I \times CPI \times \frac{\text{periodo di clock}}{2}} = \frac{3,36}{4,75 \times \frac{1}{2}} = 1,41$$

Il calcolatore con il clock più veloce è quindi circa 1,4 volte più veloce anziché 2 volte più veloce, per effetto della maggiore influenza dei miss nella cache.

Come illustrato dagli esempi le penalità relative dovute alla cache aumentano via via che i sistemi diventano più veloci. Inoltre se un sistema migliora sia la frequenza di clock che il valore di CPI, viene penalizzato due volte:

1. Più è basso il CPI, più evidente è l'impatto dei cicli dovuti agli stalli.

2. È improbabile che il sottosistema di memoria migliori con la stessa rapidità del processore. Nel calcolo del CPI, la penalità dei miss nella cache è misurata in numero di cicli di clock della CPU richiesti per la risoluzione di un miss: se le memorie principali di due elaboratori hanno lo stesso tempo di accesso assoluto, una maggiore frequenza di clock della CPU porta ad un maggior valore numerico della penalità di miss.

L'importanza delle prestazioni della cache è dunque maggiore per la CPU con un basso CPI ed un'elevata frequenza di clock, per cui il pericolo di sottovalutare il comportamento della cache nella valutazione delle prestazioni di tali macchine è maggiore. Come si vedrà nel Paragrafo 7.6 l'uso di processori veloci e dotati di pipeline nei Personal Computer e nelle workstation ha portato all'uso di sottosistemi di cache sofisticati anche in calcolatori del costo di qualche milione di lire.

Gli esempi e le equazioni precedenti ipotizzano che il tempo speso nelle situazioni di hit non sia un fattore rilevante nel determinare le prestazioni della cache. Come è ovvio, se tale tempo aumentasse, il tempo totale per accedere ad una parola nel sottosistema di memoria aumenterebbe di conseguenza, con il rischio di causare un aumento del periodo di clock del processore. Una cache di dimensioni maggiori potrebbe avere un tempo di accesso superiore, come accadrebbe per una scrivania molto grande (ad esempio di 3 metri quadrati), in cui il tempo per prendere un libro sarebbe maggiore. Oltre una certa soglia per una cache molto grande l'incremento del tempo dovuto agli hit potrebbe essere dominante rispetto al miglioramento ottenibile sulla frequenza di hit, portando ad una riduzione delle prestazioni del processore.

Il prossimo paragrafo esamina alcune organizzazioni alternative della cache che riducono la frequenza di miss ma possono aumentare il tempo di hit; altri esempi sono riportati tra gli errori ed i trabocchetti del Paragrafo 7.7.

Riduzione dei miss nella cache grazie ad una collocazione più flessibile dei blocchi

Negli esempi visti finora, al momento di collocare un blocco nella cache si era usato un metodo molto semplice: ciascun blocco poteva andare in una sola posizione della cache. Tale metodologia di collocazione prende il nome di *corrispondenza diretta* (*direct mapped*) poiché vi è un legame diretto tra l'indirizzo di ciascun blocco della memoria ed una specifica locazione del livello superiore della gerarchia. In realtà vi è un ampio spettro di possibilità per quanto riguarda la collocazione dei blocchi: un estremo di tale spettro è appunto costituito dalla corrispondenza diretta, dove ciascun blocco può essere posto in una ed una sola posizione.

All'estremo opposto vi è uno schema secondo il quale ciascun blocco può essere posto in *qualsiasi* locazione della cache: esso è detto *completamente associativo* (*fully associative*) poiché un blocco della memoria può essere associato a qualsiasi cella della cache. Per ritrovare un dato blocco all'interno di una cache completamente associativa occorre cercarlo in tutte le celle, poiché esso può risiedere in una qualunque di esse; per rendere realisticamente conveniente tale ricerca, questa viene compiuta in parallelo grazie ad un comparatore associato a ciascuna posizione della cache. Questi comparatori contribuiscono significativamente ad aumentare il costo dell'hardware, rendendo la collocazione completamente associativa conveniente solo per le cache con un numero ridotto di blocchi.

A metà strada tra i progetti a corrispondenza diretta e quelli completamente associativi si trovano le soluzioni cosiddette *set-associative* (il termine inglese signi-

fica associative all'interno di un insieme): in una cache di tipo set-associative vi è un numero fisso di posizioni (almeno due) in cui si può porre ciascun blocco; una cache di tipo set-associative con n possibili collocazioni per ciascun blocco è detta *ad n vie* (*n-way*). Una cache di tipo set-associative ad n vie è organizzata come un certo numero di insiemi, ciascuno dei quali consta di n blocchi; ciascun blocco della memoria è messo in corrispondenza con uno degli insiemi, determinato mediante il campo indice, ma può essere posto in un *qualsiasi* elemento dell'insieme. La collocazione di tipo set-associative è quindi una combinazione della corrispondenza diretta e dell'associatività completa: vi è corrispondenza diretta tra blocchi ed insiemi, mentre occorre ricercare in tutti i blocchi dell'insieme per trovare un dato corrispondente.

Si ricordi che in una cache a corrispondenza diretta la posizione dei blocchi di memoria era data da

(numero del blocco) modulo (numero di blocchi nella cache)

In una cache di tipo set-associative l'insieme che contiene un blocco di memoria è dato da

(numero del blocco) modulo (numero di insiemi nella cache)

Poiché il blocco può essere posto ovunque all'interno dell'insieme, occorre ricercare in *tutti gli elementi dell'insieme;* in una cache completamente associativa il blocco potrebbe essere ovunque e si dovrebbe cercare in *tutti i blocchi della cache.* Ad esempio la figura 7.15 mostra le possibili posizioni del blocco numero 12 in una cache con otto blocchi complessivi, secondo le strategie di collocazione delle cache a corrispondenza diretta, set-associative a due vie e completamente associativa.

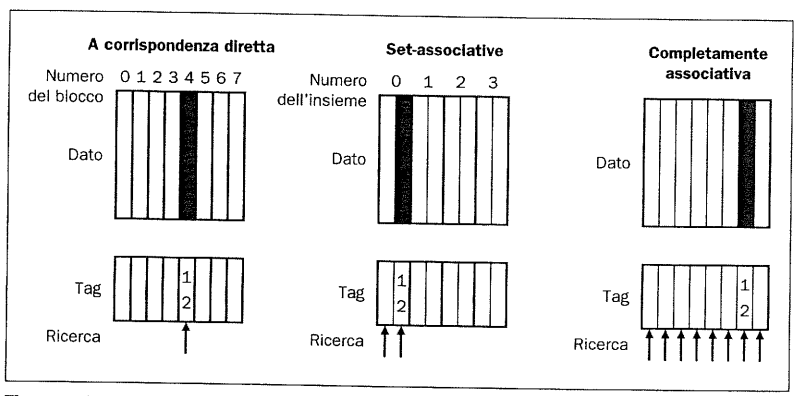


Figura 7.15 La posizione di un blocco di memoria il cui indirizzo è 12 in una cache con otto blocchi è diversa nelle metodologie di collocazione a corrispondenza diretta, set-associative e completamente associativa. Nella collocazione a corrispondenza diretta vi è un solo blocco in cui si può trovare il blocco di memoria 12, ed esso è dato da (12 modulo 8) = 4. In una cache set-associative a due vie vi sarebbero quattro insiemi ed il blocco numero 12 sarebbe nell'insieme (12 modulo 4) = 0; il blocco di memoria può essere in uno degli elementi dell'insieme. Con la collocazione completamente associativa il blocco di memoria di indirizzo 12 potrebbe comparire in uno qualsiasi degli otto blocchi della cache.

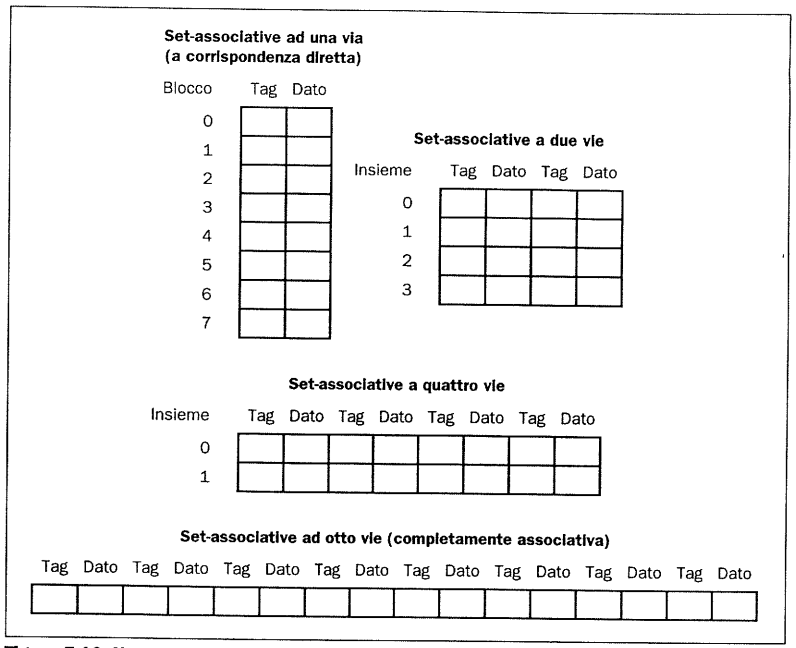


Figura 7.16 Una cache di otto blocchi configurata come cache a corrispondenza diretta, set-associative a due vie, a quattro vie e completamente associativa. La dimensione totale della cache, espressa in blocchi, è pari al numero di insiemi moltiplicato per l'entità dell'associatività. Per una data dimensione della cache un incremento dell'associatività riduce dunque il numero di insiemi, mentre allo stesso tempo aumenta il numero di elementi in ciascun insieme. Con otto blocchi, una cache set-associative ad otto vie coincide con una cache completamente associativa.

Si possono interpretare le diverse varianti delle strategie di collocazione come diverse versioni del metodo set-associative: una cache a corrispondenza diretta non è altro che una cache set-associative ad una via, poiché ciascun elemento della cache contiene un blocco e pertanto forma un insieme di un solo elemento; una cache completamente associativa di m elementi è invece una cache set-associative a m vie, poiché ha un solo insieme con m blocchi e ciascun elemento può risiedere in un qualsiasi blocco dell'insieme. La figura 7.16 illustra le possibili strutture associative per una cache di otto blocchi.

Il vantaggio di un aumento del grado di associatività risiede generalmente nella riduzione della frequenza di miss, come indicato dal prossimo esempio; lo svantaggio principale, come si vedrà in dettaglio tra breve, è un incremento del tempo di hit.

Esempio

Associatività nelle cache

Si considerino tre piccole cache, composte di quattro blocchi di una parola ciascuna. La prima cache è completamente associativa, la seconda è set-associative a due vie e la terza è a corrispondenza diretta. Si calcoli il numero di miss nel

caso delle tre organizzazioni della cache per la seguente sequenza di indirizzi di blocco: 0, 8, 0, 6, 8.

Soluzione

Il caso a corrispondenza diretta è il più semplice: si esamina innanzitutto a quale blocco della cache corrispondono gli indirizzi dei blocchi:

Indirizzo del blocco	Blocco della cache
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Si può ora vedere il contenuto della cache a seguito di ciascun accesso alla memoria, dove le celle vuote rappresentano i blocchi non validi e gli elementi colorati mostrano i nuovi blocchi aggiunti alla cache durante l'ultimo accesso:

Indirizzo del blocco di memoria a cui si è fatto accesso	Hit o miss	Contenuto dei blocchi della cache dopo l'accesso			
		0	1	2	3
0	miss	Memoria[0]			
8	miss	Memoria[8]			
0	miss	Memoria[0]			
6	miss	Memoria[0]		Memoria[6]	
8	miss	Memoria[8]		Memoria[6]	

La cache a corrispondenza diretta genera quindi cinque miss.

La cache di tipo set-associative ha due insiemi (di indice 0 ed 1) con due elementi per insieme; si determina innanzitutto a quale insieme corrisponde ciascun indirizzo di blocco:

Indirizzo del blocco	Insieme della cache
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

Poiché è possibile scegliere l'elemento dell'insieme da sostituire nel caso di un miss, occorre definire una regola di sostituzione. Le cache di tipo set-associative solitamente sostituiscono l'elemento usato meno di recente tra quelli dell'insieme, ossia viene sostituito il blocco che è rimasto inutilizzato da più tempo (tra breve si discuteranno le regole di sostituzione con maggior dettaglio). Con questa regola di sostituzione il contenuto della cache di tipo set-associative dopo i diversi accessi è il seguente:

Indirizzo del blocco di memoria a cui si è fatto accesso	Hit o miss	Contenuto dei blocchi della cache dopo l'accesso			
		Insieme 0	Insieme 1	Insieme 2	Insieme 3
0	miss	Memoria[0]			
8	miss	Memoria[0]	Memoria[8]		
0	hit	Memoria[0]	Memoria[8]		
6	miss	Memoria[0]	Memoria[6]		
8	miss	Memoria[8]	Memoria[6]		

Si noti che quando si fa riferimento al blocco 6, questo sostituisce il blocco 8, poiché quest'ultimo era stato utilizzato meno di recente rispetto al blocco 0. La cache di tipo set-associative a due vie genera quattro miss, uno in meno della cache a corrispondenza diretta.

La cache completamente associativa dispone di quattro blocchi (tutti nello stesso insieme), per cui ogni blocco di memoria può essere posto in un qualsiasi blocco della cache. La cache completamente associativa ha le prestazioni migliori, causando solamente tre miss:

Indirizzo del blocco di memoria a cui si è fatto accesso	Hit o miss	Contenuto dei blocchi della cache dopo l'accesso			
		Blocco 0	Blocco 1	Blocco 2	Blocco 3
0	miss	Memoria[0]			
8	miss	Memoria[0]	Memoria[8]		
0	hit	Memoria[0]	Memoria[8]		
6	miss	Memoria[0]	Memoria[8]	Memoria[6]	
8	hit	Memoria[0]	Memoria[8]	Memoria[6]	

In questa serie di accessi alla memoria, tre è il numero minimo di miss poiché vi sono tre distinti indirizzi di blocco a cui far riferimento. Si noti che se vi fossero stati otto blocchi nella cache, non vi sarebbero state sostituzioni nella cache set-associative a due vie (lo si verifichi per esercizio) e si sarebbe quindi raggiunto lo stesso numero di miss della cache completamente associativa. Analogamente se vi fossero stati 16 blocchi le tre cache avrebbero avuto lo stesso numero di miss: queste variazioni della frequenza di miss evidenziano come la dimensione della cache e l'associatività non siano fattori indipendenti ai fini delle prestazioni delle cache.

Qual è la riduzione della frequenza di miss ottenibile grazie all'associatività? La figura 7.17 riporta i miglioramenti per i programmi gcc e spice con un insieme di due cache di 64 KB (separate per istruzioni e dati) con blocchi di quattro parole, in cui l'associatività varia dalla corrispondenza diretta a quattro vie. Per gcc il passaggio da una via a due vie migliora la frequenza di miss combinata di circa il 20%, ma non si evidenzia alcun miglioramento nel passare ad un'associatività a quattro vie. Le basse frequenze di miss del programma spice non consentono significativi miglioramenti dipendenti dagli aumenti di associatività.

Programma	Associatività	Frequenza di miss delle istruzioni	Frequenza di miss dei dati	Frequenza di miss combinata
gcc	1	2,0%	1,7%	1,9%
gcc	2	1,6%	1,4%	1,5%
gcc	4	1,6%	1,4%	1,5%
spice	1	0,3%	0,6%	0,4%
spice	2	0,3%	0,6%	0,4%
spice	4	0,3%	0,6%	0,4%

Figura 7.17 Le frequenze di miss dei programmi gcc e spice con una cache simile a quella della DECStation 3100 ma con una dimensione del blocco pari a quattro parole e un'associatività variabile da una via a quattro vie.

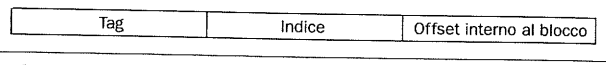


Figura 7.18 Le tre parti di un indirizzo per cache di tipo set-associative o a corrispondenza diretta. L'indice è utilizzato per selezionare l'insieme, mentre il tag è usato per scegliere il blocco mediante un confronto con i blocchi dell'insieme. L'offset nel blocco è l'indirizzo dei dati richiesti all'interno del blocco.

Ricerca di un blocco all'interno della cache

Si consideri a questo punto il problema di trovare un blocco in una cache di tipo set-associative. Esattamente come le cache a corrispondenza diretta, quelle set-associative comprendono un campo tag associato all'indirizzo che permette di risalire all'indirizzo del blocco. Occorre verificare il tag di ciascun blocco della cache dell'insieme considerato per determinare se corrisponde all'indirizzo proveniente dalla CPU. La figura 7.18 mostra come è decomposto ciascun indirizzo: il valore dell'indice viene usato per scegliere l'insieme contenente l'indirizzo considerato, all'interno del quale occorrerà controllare tutti i tag. Poiché la velocità è il fattore più importante, tutti i tag dell'insieme dovranno essere esaminati in parallelo: come nel caso delle cache completamente associative, una ricerca seriale renderebbe le cache set-associative troppo lente.

Se la dimensione totale viene mantenuta costante, un aumento dell'associatività aumenta il numero di blocchi per insieme, che corrisponde al numero di confronti contemporanei richiesti per poter compiere la ricerca in parallelo: ciascun incremento di un fattore due nell'associatività raddoppia il numero di blocchi per insieme e dimezza il numero di insiemi. Di conseguenza un incremento di un fattore due nell'associatività riduce la dimensione dell'indice di 1 bit ed aumenta la dimensione del campo tag di 1 bit. In una cache di tipo completamente associativo vi è in effetti un solo insieme, ed occorre controllare in parallelo tutti i blocchi; non vi è quindi alcun indice e l'intero indirizzo, con esclusione dell'offset nel blocco, viene confrontato con il tag dei blocchi. In altre parole si ricerca all'interno dell'intera cache senza far uso di alcun indice.

In una cache a corrispondenza diretta, come quella mostrata nella figura 7.7, è richiesto un solo comparatore poiché l'elemento può trovarsi solamente in un blocco e poiché si può accedere alla cache mediante il solo indice. In una cache di tipo set-associative a quattro vie, mostrata in figura 7.19, sono necessari quattro comparatori ed un multiplexer a quattro vie per poter scegliere tra i quattro potenziali elementi dell'insieme selezionato. L'accesso alla cache prevede di indicizzare l'insieme opportuno e poi di ricercare all'interno degli elementi dell'insieme. Il costo di una cache associativa risiede nei comparatori aggiuntivi e nel ritardo imposto dall'operazione di confronto e di selezione tra gli elementi dell'insieme.

Esempio

Dimensione dei campi tag rispetto all'associatività delle cache

Una maggiore associatività richiede più comparatori, ma anche più bit per i tag in ciascun blocco della cache. Disponendo di una cache di 4K blocchi ed indirizzi di 32 bit, si trovi il numero totale di insiemi ed il numero totale di bit per i tag nel caso di cache a corrispondenza diretta, set-associative a due vie e quatto vie e completamente associativa.

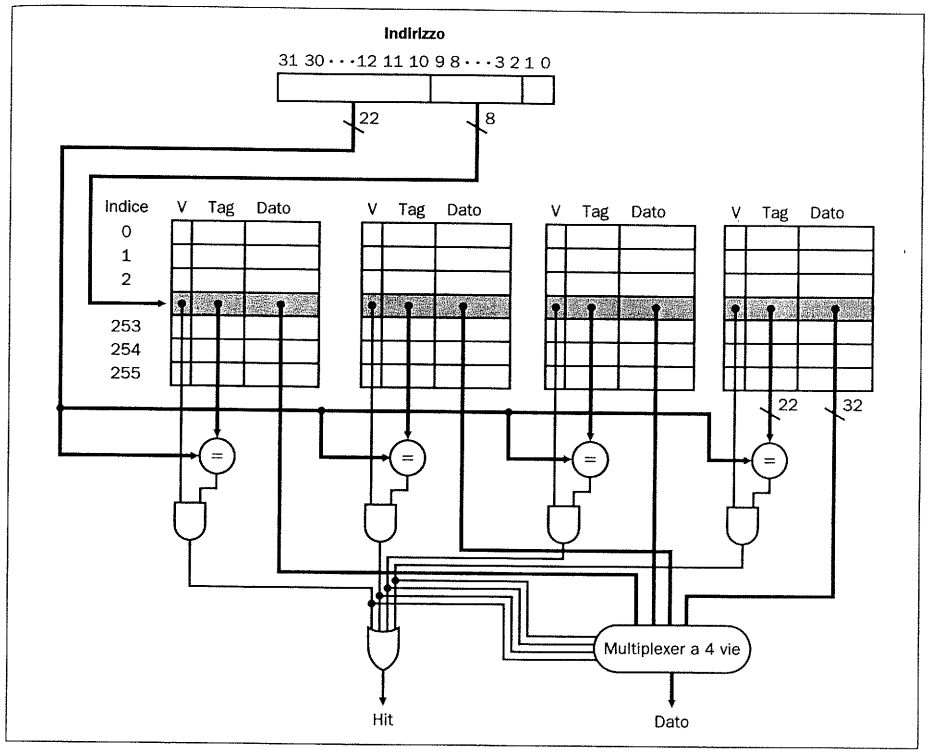


Figura 7.19 L'implementazione di una cache di tipo set-associative a quattro vie richiede quattro comparatori ed un multiplexer a quattro vie. I comparatori determinano quale elemento dell'insieme (se esiste) corrisponde al tag. L'uscita dei comparatori è utilizzata per selezionare il dato da uno dei quatto blocchi dell'insieme indicizzato, mediante un multiplexer con un segnale di selezione decodificato. In alcune implementazioni si utilizzano i segnali di abilitazione delle uscite (Output enable) delle parti di dato delle RAM della cache per determinare l'elemento dell'insieme che determinerà il valore in uscita; i segnali di abilitazione delle uscite provengono dai comparatori, facendo sì che l'elemento il cui tag corrisponde a quello dell'indirizzo richiesto possa pilotare le uscite. Questo tipo di organizzazione permette di ovviare alla necessità di un multiplexer.

Soluzione

La cache a corrispondenza diretta ha lo stesso numero di insiemi e di blocchi, da cui si deriva un indice di 12 bit, poiché $\log_2(4K) = 12$; il numero totale di bit per i tag è quindi $(32 - 12) \times 4K = 80$ Kbit.

Ciascun grado di associatività in più riduce il numero di insiemi di un fattore due e quindi riduce di uno il numero di bit usati come indice ed aumenta di uno il numero di bit usati come tag. In una cache set-associative a due vie vi sono dunque 2K insiemi ed il numero totale di bit per i tag è $(32 - 11) \times 2 \times 2K = 84$ Kbit. In una cache set-associative a quattro vie il numero totale di insiemi è 1K ed il numero totale di bit per i tag è $(32 - 10) \times 4 \times 1K = 88$ Kbit.

In una cache completamente associativa vi è un solo insieme di 4K blocchi, con una dimensione dei tag di 32 bit, per un totale di $32 \times 4K \times 1 = 128$ Kbit per i tag.

La scelta tra la corrispondenza diretta, quella set-associative o la completa associatività nei sottosistemi di memoria dipenderà dal confronto tra il costo di un miss rispetto al costo associato all'implementazione dell'associatività, misurato sia come tempo che come logica aggiuntiva.

Scelta del blocco da sostituire

Quando si verifica un fallimento nell'accesso ad una cache a corrispondenza diretta, il blocco richiesto può andare in una sola posizione, per cui occorre sostituire il blocco che occupava quella stessa posizione. Nelle cache di tipo associativo si ha invece un insieme di posizioni in cui porre il blocco, e quindi un insieme di blocchi da sostituire tra cui scegliere. In una cache completamente associativa tutti i blocchi sono candidati ad essere sostituiti, mentre in quelle set-associative si può scegliere tra i blocchi dell'insieme selezionato.

La tecnica utilizzata con maggior frequenza è quella dell'*elemento usato meno recentemente* (o *LRU*, dal termine inglese *least recently used*), già usata nell'esempio precedente. Il metodo LRU prevede che il blocco sostituito sia quello che è rimasto inutilizzato per la maggior quantità di tempo; esso viene implementato tenendo traccia di quando ciascun elemento dell'insieme è stato utilizzato relativamente agli altri elementi dell'insieme. In una cache di tipo set-associative a due vie, per determinare quale elemento dell'insieme è stato usato più di recente è sufficiente un singolo bit per insieme: ogni volta che un elemento dell'insieme viene selezionato, detto bit assume un valore tale da indicare questo elemento. Al crescere del grado di associatività l'implementazione dello schema LRU diviene più complessa: il Paragrafo 7.5 esaminerà degli schemi di sostituzione alternativi.

Riduzione della penalità di miss mediante cache a più livelli

Tutti gli elaboratori moderni fanno uso di cache; in molti casi, le cache sono addirittura implementate sullo stesso die del microprocessore usato come CPU. Per colmare ulteriormente il divario tra le alte frequenze di clock dei processori odierni ed il tempo relativamente lungo necessario ad accedere alle DRAM, i microprocessori ad alte prestazioni supportano un livello aggiuntivo di cache. Tale *cache di secondo livello*, che si trova spesso al di fuori del chip in un insieme separato di SRAM, è consultata ogniqualvolta avviene un miss nella cache principale. Se la cache di secondo livello contiene il dato richiesto la penalità di miss sarà solamente relativa al tempo di accesso alla cache di secondo livello, decisamente inferiore al tempo di accesso della memoria principale. Se il dato non è presente né nella cache principale né in quella secondaria, si renderà necessario un accesso alla memoria principale, determinando una penalità di miss maggiore.

Quanto può essere significativo il miglioramento delle prestazioni dovuto all'uso di una cache secondaria? L'esempio seguente cerca di dare una risposta a questa domanda.

Esempio

Prestazioni delle cache a più livelli

Si ipotizzi di disporre di un processore con un valore di CPI pari ad 1,0 (ipotizzando che tutti gli accessi siano risolti nella cache principale) ed una frequenza di clock di 500 MHz. Si ipotizzi un tempo di accesso alla memoria di 200 ns, compresa la gestione del miss; si ipotizzi infine che la frequenza di miss per istruzione in corrispondenza della cache principale sia del 5%. Quanto risulterebbe più veloce il calcolatore se si aggiungesse una cache secondaria con un tempo di accesso di 20 ns (sia nel caso di hit che di miss) ed abbastanza capiente da ridurre la frequenza di miss rispetto alla memoria principale al 2%?

Soluzione

La penalità per i miss che devono accedere alla memoria principale è di

$$\frac{200\ \text{ns}}{2\ \dfrac{\text{ns}}{\text{ciclo di clock}}} = 100 \text{ cicli di clock}$$

Il valore effettivo di CPI con un livello di cache è dato da

$$\text{CPI totale} = \text{CPI di partenza} + \text{cicli di stallo per istruzione dovuti alla memoria}$$

corrispondente a

$$\text{CPI totale} = 1{,}0 + \text{cicli di stallo per istruzione dovuti alla memoria} = \\ = 1{,}0 + 5\% \times 100 = 6{,}0$$

Con una cache a due livelli i miss nella cache principale (o di primo livello) possono essere soddisfatti dalla cache secondaria o dalla memoria principale; la penalità di miss per un accesso alla cache di secondo livello è di

$$\frac{20\ \text{ns}}{2\ \dfrac{\text{ns}}{\text{ciclo di clock}}} = 10 \text{ cicli di clock}$$

Se il miss viene risolto dalla cache di secondo livello, questo valore costituisce l'intera penalità di miss; se occorre invece far riferimento alla memoria principale, la penalità di miss totale sarà la somma dei tempi di accesso della cache secondaria e della memoria principale.
In una cache a due livelli, il valore totale di CPI è dato dalla somma dei cicli di stallo dovuti ai due livelli più il CPI di partenza

$$\text{CPI totale} = 1 + \text{cicli di stallo per istruzione dovuti al primo livello} + \\ + \text{cicli di stallo per istruzione dovuti al secondo livello} = \\ = 1 + 5\% \times 10 + 2\% \times 100 = 3{,}5.$$

Il calcolatore dotato di cache secondaria sarà quindi più veloce di un fattore pari a

$$\frac{6{,}0}{3{,}5} = 1{,}7$$

Un modo di procedere alternativo sarebbe stato quello di calcolare i cicli di stallo sommando quelli degli accessi che portano a degli hit nella cache secondaria ($(5\% - 2\%) \times 10 = 0{,}3$) e quelli degli accessi alla memoria principale, per i quali occorre conteggiare anche il costo dell'accesso alla cache secondaria oltre al tempo di accesso alla memoria vero e proprio ($2\% \times (10 + 100) = 2{,}2$).

Le considerazioni progettuali legate alle cache di primo e di secondo livello sono notevolmente diverse, poiché la presenza dell'ulteriore cache modifica le scelte ottimali rispetto a quelle per una cache ad un solo livello. In particolare una struttura delle cache a due livelli permette alla cache principale di concentrarsi sulla riduzione del tempo associato agli hit al fine di ridurre il periodo del clock, mentre la cache secondaria può giocare sulla frequenza di miss per ridurre la penalità dovuta alla lentezza della memoria principale.

Le interazioni tra le due cache permettono una tale divisione di ruoli: la penalità di miss della cache principale viene notevolmente ridotta grazie alla presenza della cache secondaria, permettendo così di avere dimensioni minori e frequenze di miss maggiori al primo livello. Nella cache secondaria il tempo di accesso non è così importante grazie alla presenza della cache principale, poiché il tempo di accesso della cache secondaria influenza la penalità di miss della cache principale ma non il suo tempo di hit, né tantomeno la frequenza di clock della CPU.

Gli effetti di tali modifiche introdotte dai due livelli si possono vedere confrontando le due cache con quella ottimale per un'architettura ad un solo livello. Rispetto ad una cache unica la cache principale di un sistema a più livelli è solitamente più piccola; inoltre tende ad utilizzare un blocco di dimensioni minori, associato ad una dimensione della cache minore e ad una ridotta penalità di miss. Nel confronto la cache di secondo livello sarà normalmente più grande di una cache unica, poiché il suo tempo di accesso è meno vincolante; date le maggiori dimensioni la cache di secondo livello farà solitamente uso di blocchi più grandi di quanto sarebbe conveniente in una sola cache.

Approfondimento: Con l'utilizzo di cache a più livelli nascono diverse complicazioni, una delle quali è data dal fatto che vi sono diversi tipi di miss e di frequenze di miss corrispondenti. Nell'esempio precedente si era vista la frequenza di miss della cache principale e la *frequenza di miss globale*, ossia la frazione degli accessi che sono assenti in tutti i livelli. Vi è anche una frequenza di miss per la cache secondaria, data dal rapporto di tutti i miss nella cache di secondo livello rispetto al numero di accessi: tale quantità è detta *frequenza di miss locale* della cache di secondo livello. Poiché la cache principale filtra gli accessi, soprattutto quelli dotati di una buona località spaziale e temporale, la frequenza di miss locale della cache secondaria è molto più alta di quella globale. Nell'esempio appena visto si può calcolare la frequenza di miss locale della cache secondaria come 2%/5% = 40%! Fortunatamente ciò che determina quanto frequentemente si debba accedere alla memoria principale è la frequenza di miss complessiva! Ulteriori complicazioni possono nascere per via delle dimensioni dei blocchi delle diverse cache, che saranno probabilmente diverse per potersi adattare alla maggiore o minore dimensione complessiva; per le stesse ragioni può anche variare il tipo di associatività delle cache. Le cache integrate sul chip sono solitamente costruite con un'associatività di due o quattro, mentre le cache esterne al chip raramente hanno associatività maggiore di due. Queste variazioni nella dimensione del blocco e nell'associatività introducono delle complicazioni nel modellare le cache, il che tipicamente significa che occorre simulare contemporaneamente i due livelli per comprenderne il comportamento.

Riepilogo

In questo paragrafo si è posta l'attenzione su tre argomenti: le prestazioni delle cache, l'uso dell'associatività al fine di ridurre la frequenza di miss e l'uso delle gerarchie di cache a più livelli per ridurre le penalità di miss.

Essendo il numero totale di cicli spesi da un programma pari alla somma dei cicli richiesti dal processore e di quelli di stallo per la memoria, il sottosistema di memoria può avere un impatto significativo sul tempo di esecuzione dei programmi. In effetti con l'aumentare delle prestazioni dei processori (mediante una riduzione del valore di CPI o un aumento della frequenza di clock) l'effetto relativo dei cicli di stallo per la memoria aumenta, facendo sì che un buon sottosistema di memoria sia un elemento critico per ottenere elevate prestazioni. Il numero di cicli di stallo per la memoria dipende sia dalla frequenza di miss che dalla penalità di miss. La sfida progettuale, come si vedrà nel Paragrafo 7.5, è di riuscire a ridurre uno di questi due fattori senza influenzare significativamente gli altri fattori critici della gerarchia delle memorie.

Per ridurre la frequenza di miss si è esaminato l'utilizzo di metodologie di collocazione di tipo set-associative: tali tecniche possono ridurre la frequenza di miss permettendo un posizionamento più flessibile dei blocchi nella cache. I metodi completamente associativi permettono ad un blocco di essere posto ovunque, ma allo stesso tempo richiedono di compiere una ricerca in tutti i blocchi della cache per soddisfare le richieste. Tale ricerca viene solitamente implementata mediante un comparatore per ciascun blocco della cache e controllando in parallelo tutti gli elementi della cache; il costo dei comparatori rende però improponibili cache completamente associative di grandi dimensioni. Una valida alternativa è costituita dalle cache di tipo set-associative, poiché è necessario ricercare unicamente tra gli elementi di un solo insieme che viene selezionato mediante il campo indice; la tecnica set-associative fornisce un miglioramento della frequenza di hit ma rallenta lievemente gli accessi, per via del costo dei confronti e della scelta tra gli elementi dell'insieme. Il fatto che le prestazioni migliori siano ottenibili da una cache a corrispondenza diretta o da una set-associative dipende sia dalla tecnologia che dai dettagli della sua implementazione.

Si sono infine esaminate le cache multi-livello come una possibile soluzione per ridurre la frequenza di miss, permettendo ad una seconda cache di farsi carico dei miss della cache principale. Le cache di secondo livello si sono notevolmente diffuse dal momento che i progettisti si sono resi conto che le limitate risorse di silicio e l'obiettivo di un'elevata frequenza di clock si oppongono ad un aumento della dimensione delle cache di primo livello. La cache secondaria, solitamente 10 o più volte più ampia di quella principale, cattura molti accessi che si erano risolti in un miss nella cache principale: in tali casi la penalità di miss è pari al tempo di accesso alla cache di secondo livello (tipicamente meno di 10 cicli) anziché il tempo di accesso alla memoria (tipicamente oltre 40 cicli). Come per l'associatività, i compromessi progettuali tra la dimensione della cache di secondo livello ed il suo tempo d'accesso dipendono da molti aspetti legati all'implementazione.

7.4 La memoria virtuale

...è stato scoperto un sistema per far percepire al programmatore la combinazione dei nuclei magnetici e del tamburo come un'unica memoria ad un solo livello, compiendo automaticamente i trasferimenti necessari.

Kilburn et al., "*One-level storage systems*", 1962

Nel paragrafo precedente si è visto come le cache possano servire a fornire un accesso veloce alle parti del codice e dei dati di un programma usate frequentemente; in modo del tutto analogo la memoria principale può agire come "cache" rispetto

ai dispositivi di memorizzazione secondari, solitamente realizzati con dischi magnetici. Una tale tecnica è detta *memoria virtuale*. Vi sono due motivazioni che spingono all'uso della memoria virtuale: permettere una condivisione efficiente e sicura della memoria tra molti programmi ed eliminare le difficoltà di programmazione legate ad una quantità piccola e limitata di memoria principale.

Si consideri un gruppo di programmi in esecuzione contemporanea su un calcolatore: la memoria complessiva richiesta da tutti i programmi potrebbe essere molto superiore alla quantità di memoria principale disponibile sul calcolatore, ma in ogni istante di tempo solo una frazione di questa memoria viene usata attivamente. La memoria principale deve solo contenere le parti attive dei diversi programmi, così come una cache contiene solo la parte attiva di un programma: in questo modo si potrà condividere in modo efficiente il processore, oltre che la memoria principale. Naturalmente prima di permettere a più programmi di condividere la stessa memoria occorrerà proteggerli l'uno dall'altro, garantendo che un programma possa solo leggere e scrivere le parti di memoria principale che gli sono state assegnate.

Non si può sapere quali programmi condivideranno la memoria con quali altri nel momento in cui si compilano: infatti i programmi che condividono la memoria posso cambiare dinamicamente mentre altri sono in esecuzione. Per via di questa interazione dinamica è preferibile compilare ciascun programma in uno *spazio di indirizzamento* separato, ossia un intervallo separato di locazioni di memoria accessibili solo al programma. La memoria virtuale implementa la traduzione dallo spazio di indirizzamento di un programma agli indirizzi fisici; il processo di traduzione garantisce la protezione dello spazio di indirizzamento di un programma dagli altri programmi.

Una seconda motivazione per la memoria virtuale è di permettere ad un singolo programma utente di superare le dimensioni fisiche della memoria principale. In precedenza se un programma diveniva troppo grande per essere contenuto in memoria, farlo funzionare diventava responsabilità del programmatore; i programmatori dividevano i programmi in parti ed identificavano le parti mutuamente esclusive. Tali parti erano dette *overlay* (poiché si sovrascrivevano a vicenda) e venivano caricate e scaricate sotto il controllo del programma utente durante la sua esecuzione; il programmatore doveva garantire che il programma non cercasse mai di accedere ad un overlay che non era caricato e che gli overlay caricati non eccedessero mai la dimensione totale della memoria. Gli overlay erano tradizionalmente organizzati come moduli, ciascuno dei quali conteneva sia codice che dati; le chiamate a procedure in moduli diversi portavano alla sovrascrittura di un modulo con un altro.

Come è facile immaginare, questa responsabilità si traduceva in un notevole carico sui programmatori. La memoria virtuale, inventata per sollevare i programmatori da queste difficoltà, gestisce in modo automatico i due livelli della gerarchia delle memorie costituiti dalla memoria principale (talvolta detta *memoria fisica* per distinguerla dalla memoria virtuale) e dalla memoria di massa.

Sebbene i concetti implicati nella memoria virtuale e nelle cache siano fondamentalmente gli stessi, le loro diverse radici storiche hanno portato all'uso di terminologie diverse. Un blocco di memoria virtuale è detto *pagina* e la condizione corrispondente ad un miss nella memoria virtuale prende il nome di *mancanza di pagina*, spesso indicata con il termine inglese *page fault*. In presenza della memoria virtuale, la CPU genera un *indirizzo virtuale*, che viene tradotto da un insieme di componenti hardware e software in un *indirizzo fisico*, a sua volta utilizzato per accedere alla memoria principale. La figura 7.20 riporta la memoria indirizzata in modo vir-

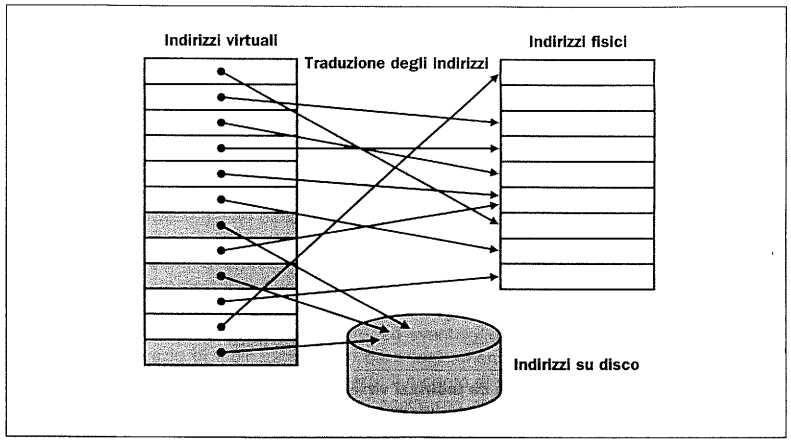


Figura 7.20 Nella memoria virtuale i blocchi di memoria (detti pagine) sono posti in corrispondenza tra un insieme di indirizzi (detti indirizzi virtuali) ed un altro (detti indirizzi fisici). Il processore genera indirizzi virtuali, mentre gli accessi alla memoria avvengono attraverso gli indirizzi fisici. Sia la memoria virtuale che quella fisica sono suddivise in pagine, per cui una pagina virtuale corrisponde in realtà ad una pagina fisica. È ovviamente possibile per una pagina virtuale non essere presente nella memoria principale e quindi non avere un indirizzo fisico corrispondente, risiedendo essa su disco. Le pagine fisiche sono condivisibili facendo in modo che a due indirizzi virtuali corrisponda lo stesso indirizzo fisico: questa caratteristica permette a programmi diversi di condividere dati o codice.

tuale, le cui pagine corrispondono a parti della memoria principale; questo processo prende il nome di *rimappatura della memoria* (*memory mapping*) o *traduzione degli indirizzi*. Attualmente i due livelli gerarchici gestiti dalla memoria virtuale sono le DRAM ed i dischi magnetici (si veda il Capitolo 1, Paragrafo 1.3). Ritornando all'analogia della biblioteca, si può pensare all'indirizzo virtuale come al titolo di un libro e all'indirizzo fisico come alla collocazione di tale libro nella biblioteca.

La memoria virtuale semplifica anche il caricamento del programma all'atto della sua esecuzione permettendo in modo automatico la cosiddetta *rilocazione* (o ricollocazione). La rilocazione consiste nel porre in corrispondenza gli indirizzi virtuali usati da un programma con i diversi indirizzi fisici prima che questi vengano usati per indirizzare la memoria; la rilocazione permette quindi di caricare il programma in una qualsiasi locazione all'interno della memoria principale. Inoltre tutti i sistemi dotati di memoria virtuale oggi funzionanti rilocano il programma trattandolo come un insieme di blocchi di dimensioni fisse (pagine), eliminando quindi la necessità di trovare un blocco di memoria contiguo da allocare ad un programma: il sistema operativo deve invece trovare semplicemente un numero sufficiente di pagine nella memoria principale. In precedenza i problemi di rilocazione richiedevano uno speciale supporto hardware e speciali funzioni del sistema operativo, mentre oggigiorno la memoria virtuale è in grado di fornire questo servizio aggiuntivo.

Nella memoria virtuale gli indirizzi sono articolati in due parti: un *numero di pagina virtuale* ed un *offset interno alla pagina*; la figura 7.21 mostra il processo di traduzione del numero di pagina virtuale in un *numero di pagina fisica*. Il numero di pagina fisica rappresenta la parte più significativa dell'indirizzo fisico, mentre l'off-

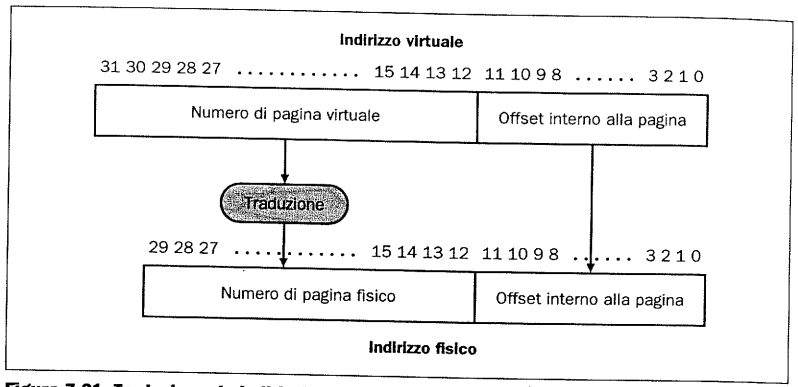


Figura 7.21 Traduzione da indirizzi virtuali ad indirizzi fisici. La dimensione delle pagine è di 2^{12} = 4 KB; il numero di pagine fisiche permesse è 2^{18}, poiché il numero di pagina fisica ha 18 bit. Questo significa che la dimensione della memoria fisica può arrivare ad 1 GB, mentre lo spazio di indirizzamento virtuale è di 4 GB.

set interno alla pagina, che non viene modificato, ne costituisce la parte meno significativa; il numero di bit nel campo di offset nella pagina determina quindi la dimensione delle pagine. Il numero di pagine indirizzabili mediante gli indirizzi virtuali non deve necessariamente corrispondere al numero di pagine indirizzabili dagli indirizzi fisici: il disporre di un numero di pagine virtuali maggiore del numero di pagine fisiche è alla base dell'illusione di una capacità pressoché illimitata della memoria virtuale.

Molte scelte di progetto nei sistemi con memoria virtuale sono motivate dall'elevato costo dei miss, che nel caso della memoria virtuale sono tradizionalmente denominati *page fault* (mancanza di una pagina). Un singolo page fault richiede milioni di cicli per essere trattato (la prima tabella del Paragrafo 7.1 mostra la velocità relativa della memoria principale e dei dischi). Questa enorme penalità di miss, che per le dimensioni normali delle pagine è dovuta in gran parte al tempo necessario a prelevare la prima parola, porta a parecchie decisioni chiave nel progetto di sistemi a memoria virtuale:

- Le pagine devono essere abbastanza grandi da poter ammortizzare l'elevato costo di accesso. Le dimensioni tipiche odierne vanno da 4 KB a 16 KB, ma sono in corso di sviluppo sistemi che supportano pagine di 32 KB e 64 KB, portando ad una scomparsa delle pagine di 4 KB.
- Le soluzioni che riducono la frequenza dei page fault hanno un'importanza cruciale: da questo punto di vista, la tecnica più utilizzata prevede la collocazione completamente associativa delle pagine.
- I page fault possono essere gestiti dal software poiché l'overhead è comunque trascurabile rispetto al tempo di accesso al disco. Inoltre le implementazioni software si possono permettere di usare algoritmi più intelligenti per scegliere come posizionare le pagine, perché anche dei piccoli miglioramenti della frequenza di miss compensano il costo di tali algoritmi.
- La tecnica di scrittura write-through non può funzionare nella memoria virtuale, poiché le scritture sono troppo lunghe: i sistemi di memoria virtuale usano quindi la tecnica write-back.

I paragrafi seguenti tratteranno l'impatto di questi fattori nel progetto dei sistemi dotati di memoria virtuale.

Approfondimento: In questo libro la trattazione della memoria virtuale si concentra sulla *paginazione*, che usa blocchi di dimensione fissa; esiste anche uno schema con blocchi di dimensioni variabili detto *segmentazione*. Nel caso della segmentazione gli indirizzi consistono di due parti: un numero di segmento ed un offset interno al segmento. Il numero di segmento è convertito in un indirizzo fisico, a cui si aggiunge l'offset per trovare il reale indirizzo fisico. Poiché i segmenti hanno dimensione variabile è necessario un controllo dei limiti per verificare che l'offset sia interno al segmento. L'uso principale che si fa della segmentazione sono i metodi più potenti di protezione e condivisione dello spazio di indirizzamento. La maggior parte dei libri di testo sui sistemi operativi contiene discussioni approfondite sul confronto tra segmentazione e paginazione e sull'uso della segmentazione al fine di permettere condivisione logica dello spazio di indirizzamento. Lo svantaggio principale della segmentazione è di separare lo spazio di indirizzamento in parti logicamente separate che devono essere manipolate come indirizzi composti di due parti: il numero di segmento e l'offset. Al contrario la paginazione rende il confine tra numero di pagina ed offset invisibile a programmatori e compilatori.

I segmenti sono anche stati utilizzati per estendere lo spazio di indirizzamento senza modificare la dimensione della parola dei processori; tali approcci non hanno avuto successo per via della complessità e delle penalizzazioni nelle prestazioni insite in un indirizzo composto di due parti di cui compilatori e programmatori devono essere consci.

Molti progettisti di architetture dividono lo spazio di indirizzamento in grandi blocchi di dimensioni fisse per semplificare i meccanismi di protezione tra sistema operativo e programmi utente e per aumentare l'efficienza implementativa della paginazione. Sebbene tali divisioni siano spesso chiamate "segmenti", questo meccanismo è molto più semplice della segmentazione con blocchi di dimensione variabile ed è invisibile ai programmi utente; lo si esaminerà tra breve.

Collocazione e ritrovamento delle pagine

Per via della penalità incredibilmente elevata dei page fault, i progettisti tendono a ridurne il numero ottimizzando la collocazione delle pagine. Se si permette ad una pagina virtuale di essere posta in una qualsiasi pagina fisica, il sistema operativo potrà scegliere di sostituire qualsiasi pagina reputi opportuno quando avviene un page fault. Ad esempio il sistema operativo potrà usare algoritmi sofisticati e strutture dati complesse per tenere traccia dell'utilizzo delle pagine al fine di cercare di scegliere una pagina che non verrà richiesta per molto tempo. La possibilità di usare delle tecniche di sostituzione intelligenti e flessibili è in effetti la motivazione principale per l'uso della collocazione completamente associativa delle pagine; naturalmente il posizionamento completamente associativo riduce anche la frequenza dei page fault.

Come già menzionato la difficoltà di utilizzo della collocazione completamente associativa risiede nel ritrovamento delle pagine, perché queste possono essere ovunque nel livello superiore della gerarchia: una ricerca esaustiva si rivela inopportuna. Nel caso della memoria virtuale le pagine vengono ritrovate mediante una tabella completa che indicizza la memoria: tale struttura è detta *tabella delle pagine* (*page table*). La tabella delle pagine, che risiede in memoria, è indicizzata con il numero di pagina preso dall'indirizzo virtuale e contiene l'indirizzo della pagina fisica corrispondente. Ciascun programma ha la propria tabella delle pagine, che pone in corrispondenza lo spazio di indirizzamento virtuale di quel programma con la memoria principale. Nell'analogia della biblioteca la tabella delle pagine cor-

risponde ad una corrispondenza tra i titoli dei libri e le loro collocazioni. Così come le schede del catalogo possono contenere delle informazioni su libri presenti in una diversa biblioteca, si vedrà che la tabella delle pagine può contenere degli elementi per le pagine non presenti in memoria. Per indicare la posizione della tabella delle pagine in memoria l'hardware comprende un registro che punta alla sua locazione iniziale; tale registro verrà detto *registro della tabella delle pagine* (*page table register*). Si ipotizzi per ora che la tabella delle pagine sia in un'area fissa e contigua della memoria.

La figura 7.22 fa uso del registro della tabella delle pagine, dell'indirizzo virtuale e della tabella delle pagine identificata dal registro per illustrare il processo mediante il quale l'hardware genera l'indirizzo fisico. Ciascun elemento della tabella delle pagine fa uso di un bit di validità, esattamente come succedeva per le

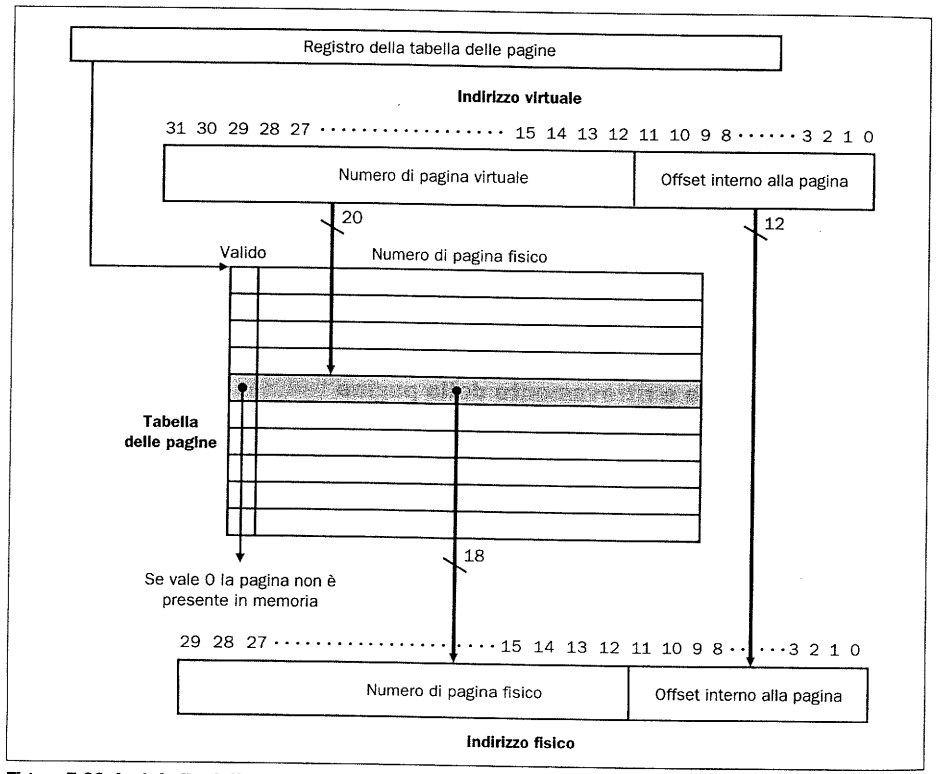


Figura 7.22 La tabella delle pagine è indicizzata con il numero della pagina virtuale e fornisce la porzione corrispondente dell'indirizzo fisico. L'indirizzo di partenza della tabella delle pagine è fornito dal registro della tabella delle pagine. In questa figura la dimensione delle pagine è di 2^{12} byte, ossia 4 KB; lo spazio di indirizzamento virtuale è di 2^{32} byte, ossia 4 gigabyte, e lo spazio di indirizzamento fisico è di 2^{30} byte, permettendo alla memoria principale di arrivare fino ad un gigabyte. Il numero di elementi della tabella è 2^{20}, circa un milione. Il bit di validità di ciascun elemento indica se la corrispondenza è legale: se vale 0, la pagina non è presente in memoria. Sebbene gli elementi della tabella delle pagine qui disegnati richiedano un'ampiezza di soli 19 byte, tipicamente verrebbero arrotondati a 32 byte per maggior facilità di indirizzamento. I bit restanti saranno usati per memorizzare informazioni aggiuntive relative a ciascuna pagina, come quelle sulla protezione.

cache. Se il bit è pari a 0 la pagina non è presente nella memoria principale e si verifica un page fault; se è pari ad 1 la pagina è valida e l'elemento contiene il numero della pagina fisica.

Poiché la tabella delle pagine contiene un elemento per ciascuna possibile pagina virtuale non vi è necessità di un campo tag. Nella terminologia delle memorie cache l'indice, che è utilizzato per accedere alla tabella delle pagine, corrisponde all'intero indirizzo di blocco, ossia del numero della pagina virtuale.

Interfaccia Hardware Software

La tabella delle pagine, insieme al contatore di programma ed ai registri, specifica lo stato di un programma. Se si vuole permettere ad un altro programma di utilizzare la CPU si dovranno salvare le informazioni di stato; il programma potrà poi riprendere l'esecuzione dopo aver ripristinato il proprio stato. Queste informazioni di stato sono in grado di definire completamente lo *stato* di un *processo*: un processo è considerato *attivo* quando è in possesso della CPU, *inattivo* in caso contrario. Il sistema operativo può rendere attivo un processo caricando lo stato del processo, compreso il contatore di programma, che farà riprendere l'esecuzione esattamente dal valore del program counter salvato.

Lo spazio di indirizzamento del processo, e quindi tutti i dati a cui questo può accedere nella memoria, è definito dalla sua tabella delle pagine, che risiede in memoria. Anziché salvare l'intera tabella delle pagine, il sistema operativo si limita a modificare il registro della tabella delle pagine in modo da farlo puntare alla tabella delle pagine del processo che vuole attivare. Ciascun processo deve avere la propria tabella delle pagine poiché tutti i processi usano gli stessi indirizzi virtuali; il sistema operativo è responsabile dell'allocazione della memoria fisica e dell'aggiornamento delle tabelle delle pagine, in modo che gli spazi di indirizzamento virtuali di diversi processi non collidano. Come si vedrà tra breve, l'uso di tabelle delle pagine separate fornisce anche un meccanismo di protezione reciproca dei processi.

I page fault

Se il bit di validità di una pagina vale 0, si verifica un page fault. A questo punto occorre trasferire il controllo al sistema operativo: tale trasferimento avviene mediante il meccanismo delle eccezioni, i cui dettagli verranno discussi più avanti in questo paragrafo. Il sistema operativo, non appena prende il controllo, deve trovare la pagina richiesta nel livello successivo della gerarchia (solitamente il disco magnetico) e decidere dove porla nella memoria principale.

L'indirizzo virtuale da solo non è sufficiente a determinare dove la pagina si trovi nel disco. Ritornando all'analogia della biblioteca non si può conoscere la posizione di un libro sugli scaffali solamente a partire dal suo titolo: occorre invece cercare il libro nel catalogo ed ottenere l'indirizzo della sua collocazione sugli scaffali. Allo stesso modo, nei sistemi con memoria virtuale occorre tenere traccia della collocazione su disco di tutte le pagine dello spazio di indirizzamento virtuale.

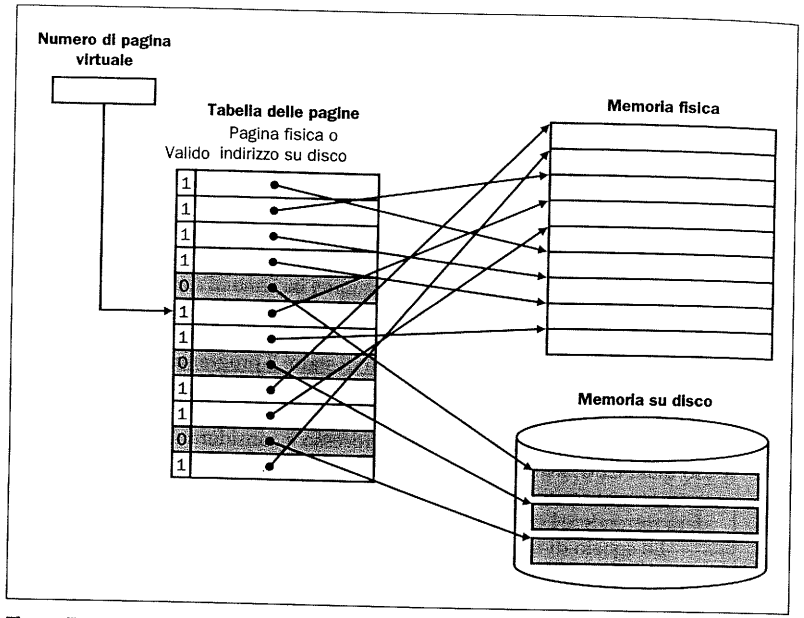


Figura 7.23 La tabella delle pagine pone in corrispondenza ciascuna pagina della memoria virtuale con una pagina nella memoria principale o con una pagina memorizzata su disco, che rappresenta il livello successivo della gerarchia. Il numero di pagina virtuale viene utilizzato per indicizzare la tabella delle pagine: se il bit di validità vale 1, la tabella delle pagine fornisce un numero di pagina fisica (ossia l'indirizzo di partenza della pagina in memoria); se il bit di validità è pari a 0, la pagina attualmente risiede su disco all'indirizzo specificato. In molti sistemi la tabella degli indirizzi delle pagine fisiche e la tabella degli indirizzi delle pagine su disco, sebbene siano logicamente una sola tabella, vengono memorizzate in due strutture dati separate. La separazione delle tabelle è in parte giustificata dal fatto che occorre ricordare l'indirizzo su disco di tutte le pagine, anche di quelle attualmente nella memoria principale. Si ricordi che le pagine nella memoria principale e quelle su disco hanno uguali dimensioni.

Poiché non si conosce in anticipo quando una data pagina in memoria verrà sostituita, il sistema operativo generalmente crea lo spazio su disco per tutte le pagine di un processo all'atto della creazione del processo stesso. Nello stesso tempo crea anche una struttura dati per ricordare dove le diverse pagine virtuali risiedano sul disco; tale struttura dati può far parte della tabella delle pagine oppure può essere una struttura dati ausiliaria indicizzata nello stesso modo in cui si indicizza la tabella delle pagine. La figura 7.23 mostra un'organizzazione dei dati in cui una sola tabella contiene sia i numeri di pagina fisica, sia gli indirizzi su disco.

Il sistema operativo crea altresì una struttura dati che tiene traccia di quali processi e di quali indirizzi virtuali usino una data pagina fisica. Quando avviene un page fault se tutte le pagine della memoria principale sono utilizzate il sistema operativo deve scegliere quale pagina sostituire. Poiché si vuole minimizzare il numero di page fault, la maggior parte dei sistemi operativi cerca di scegliere una pagina che si ipotizza non sarà richiesta nel futuro imminente; utilizzando il passato per predire il futuro, i sistemi operativi seguono il già citato metodo di sostituzione LRU (least recently used, ossia usato meno di recente). Il sistema operativo

determina quindi la pagina usata meno recentemente, assumendo che se la pagina non è stata usata da molto tempo sarà richiesta con minor probabilità di una pagina a cui è stato fatto riferimento di recente.

Ad esempio si supponga che i riferimenti alle pagine più recenti siano stati 10, 12, 9, 7, 11, 10 (in quest'ordine) e che si faccia poi riferimento alla pagina 8, non presente in memoria. La pagina che gode della proprietà LRU è la 12: nella sostituzione di tipo LRU si sostituirebbe la pagina 12 nella memoria principale con la pagina 8. Se il riferimento successivo causasse un altro page fault, si sostituirebbe la pagina 9, poiché sarebbe la pagina LRU tra quelle presenti in memoria.

Interfaccia Hardware Software

L'implementazione di uno schema LRU esatto è troppo costosa, poiché richiederebbe di aggiornare una struttura dati ad *ogni* acceso alla memoria; la maggior parte dei sistemi operativi, invece, approssima la tecnica LRU tenendo traccia di quali pagine siano state usate recentemente e quali no. Per assistere il sistema operativo nella stima delle pagine LRU, alcuni sistemi dispongono di un *bit di uso* o *bit di riferimento* che è posto ad 1 ogni volta che si accede alla pagina. Il sistema operativo periodicamente azzera i bit di riferimento e li memorizza, in modo da determinare quali pagine siano state toccate in un certo periodo di tempo. Disponendo di queste informazioni il sistema operativo potrà scegliere una pagina che sia tra quelle a cui si è fatto riferimento meno di recente (identificata dal fatto che ha il bit di riferimento pari a 0). Se un tale bit non viene messo a disposizione dalle risorse hardware, il sistema operativo dovrà trovare dei metodi alternativi per stimare gli accessi alle pagine.

Approfondimento: Con indirizzi virtuali di 32 bit, pagine di 4 KB e 4 byte per ciascun elemento della tabella delle pagine, si può calcolare la dimensione totale della tabella delle pagine come segue:

$$\text{numero di elementi della tabella delle pagine} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{dimensione della tabella delle pagine} = 2^{20} \text{ elementi della tabella delle pagine} \times 2^2 \frac{\text{byte}}{\text{elemento della tabella delle pagine}} = 4 \text{ MB}$$

In altre parole occorrerebbero 4 MB di memoria per ciascun programma in esecuzione in ogni istante di tempo. In un calcolatore con decine o centinaia di programmi attivi ed una dimensione fissa della tabella delle pagine, tutta o gran parte della memoria sarebbe riempita dalle tabelle delle pagine!

Esistono diverse tecniche che si possono utilizzare per ridurre la quantità di spazio richiesta dalla tabella delle pagine; le cinque tecniche sotto riportate mirano a ridurre la quantità totale di spazio richiesto e a minimizzare la parte della memoria principale dedicata alle tabelle delle pagine:

1. La tecnica più semplice consiste nel disporre di un registro di limite che delimiti la dimensione della tabella delle pagine per un dato processo: se il numero di una pagina virtuale diviene più grande del contenuto del registro di limite occorre aggiungere degli elementi

alla tabella delle pagine. Questa tecnica permette alla tabella delle pagine di crescere via via che il processo richiede più spazio: la tabella delle pagine quindi sarà grande solo se il processo sta usando molte pagine dello spazio di indirizzamento virtuale. Questa tecnica richiede che lo spazio di indirizzamento si espanda in una sola direzione.

2. Permettere la crescita in una sola dimensione non è sufficiente, poiché molti linguaggi richiedono due aree di dimensioni espandibili: una contiene lo stack e l'altra (detta solitamente heap) contiene i dati allocati dinamicamente. A causa di questa dualità risulta conveniente dividere la tabella delle pagine e lasciarla crescere dall'indirizzo più alto verso il basso e dall'indirizzo più basso verso l'alto; in questo caso vi saranno due tabelle delle pagine separate e due limiti separati. L'uso di due tabelle delle pagine divide lo spazio di indirizzamento in due segmenti: solitamente il bit più significativo dell'indirizzo determina quale segmento, e quindi quale tabella delle pagine debba essere usato. Essendo il segmento specificato dal bit di ordine superiore dell'indirizzo, ciascun segmento può crescere fino a metà dello spazio di indirizzamento; un registro di limite per ciascun segmento ne specifica la dimensione corrente, che può crescere a gradini di una pagina. Questo tipo di segmentazione è usato in molte architetture, tra cui quella del MIPS; a differenza della segmentazione discussa nell'approfondimento precedente questa forma di segmentazione è invisibile al programma applicativo, anche se non al sistema operativo. Lo svantaggio maggiore di questa soluzione è che non funziona molto bene quando lo spazio di indirizzamento viene usato in modo sparso anziché come un insieme contiguo di indirizzi virtuali.
3. Un altro approccio alla riduzione della dimensione della tabella delle pagine prevede di applicare una funzione di hash all'indirizzo virtuale in modo che la struttura dati della tabella delle pagine debba avere una dimensione determinata solo dal numero di *pagine fisiche* nella memoria principale. Tale struttura è denominata *tabella delle pagine invertita*; ovviamente il processo di ricerca in una tabella della pagine invertita è più complesso perché non è sufficiente indicizzare la tabella delle pagine.
4. Si possono anche utilizzare più livelli di tabelle delle pagine in modo da ridurre la quantità totale di spazio: il primo livello fornisce la corrispondenza per grossi blocchi di dimensione fissa dello spazio di indirizzamento virtuale, da 64 a 256 pagine ciascuno. Questi blocchi sono talvolta detti segmenti, e questo primo livello di tabella di corrispondenza viene detto tabella dei segmenti, anche se questi sono invisibili all'utente. Ciascun elemento della tabella dei segmenti indica se una o più pagine del segmento siano allocate, e in caso affermativo punta ad una tabella delle pagine per quel segmento. La traduzione degli indirizzi avviene innanzitutto ricercando il segmento nella tabella dei segmenti mediante i bit più significativi dell'indirizzo; se l'indirizzo del segmento è valido i successivi bit più significativi servono ad indicizzare la tabella delle pagine indicata dall'elemento della tabella dei segmenti. Questo metodo permette di usare lo spazio di indirizzamento in modo sparso (possono essere attivi molti segmenti non contigui tra loro) senza dover allocare l'intera tabella delle pagine; ciò risulta particolarmente utile con spazi di indirizzamento molto ampi ed in sistemi software che richiedano l'allocazione non contigua. Lo svantaggio principale del modello di corrispondenza a due livelli è il processo più complesso di traduzione degli indirizzi.
5. Per ridurre la quantità di memoria principale dedicata alle tabelle delle pagine la maggior parte degli elaboratori moderni permette alle tabelle delle pagine di venire a loro volta paginate. Anche se ciò potrebbe sembrare strano, è sufficiente applicare lo stesso principio fondamentale della memoria virtuale e permettendo semplicemente alle tabelle delle pagine di risiedere nello spazio di indirizzamento virtuale. In aggiunta vi sono alcuni problemi piccoli ma critici, come la possibilità di una serie infinita di page fault, che occorre evitare. Capire come vengano superati questi problemi richiede un gran livello di dettaglio ed è altamente specifico per ciascun elaboratore; in poche parole i problemi vengono evitati ponendo tutte le tabelle delle pagine nello spazio di indirizzamento del sistema operativo e ponendo almeno alcune tabelle delle pagine di sistema in una parte della memoria principale che sia indirizzata fisicamente e che sia sempre presente e mai su disco.

E le scritture?

La differenza tra il tempo di accesso alla cache e quello alla memoria principale è dell'ordine di qualche decina di cicli, per cui si possono usare delle soluzioni di tipo write-through, anche se occorre un buffer delle scritture per nascondere la latenza delle operazioni di scrittura al processore. In un sistema con memoria virtuale la scrittura al livello successivo della gerarchia (il disco) richiede milioni di cicli di clock del processore, per cui costruire un buffer delle scritture per permettere al sistema di scrivere immediatamente su disco sarebbe assolutamente illogico. I sistemi con memoria virtuale devono quindi usare la tecnica write-back, permettendo alle singole istruzioni di scrittura di scrivere nella pagina in memoria e ricopiando la pagina su disco solo quando questa viene sostituita. La ricopiatura al livello inferiore della gerarchia è alla base di un altro nome per questa tecnica di gestione delle scritture, ossia *copy back* (ricopiatura).

Interfaccia Hardware Software

Uno schema write-back ha un secondo grande vantaggio nei sistemi con memoria virtuale: poiché il tempo di trasferimento su disco è ridotto rispetto al suo tempo di accesso, la ricopiatura di un'intera pagina è molto più efficiente dello scrivere su disco le singole parole. L'operazione di ricopiatura, anche se più efficiente del trasferimento delle singole parole, rimane comunque costosa: per questo motivo sarebbe utile sapere se una pagina *necessita* di essere ricopiata su disco quando deve essere sostituita. Per tenere traccia del fatto che ci siano state delle scritture ad una pagina da quando questa è stata caricata in memoria, la tabella delle pagine è arricchita di un *bit di modifica* (detto solitamente *dirty bit*, letteralmente bit che determina se una pagina è "sporca"). Il bit di modifica viene posto ad 1 la prima volta che si scrive nella pagina: se il sistema operativo decide di sostituire quella pagina, il bit di modifica indica se la pagina debba essere riscritta su disco prima che la sua posizione di memoria sia assegnata ad un'altra pagina.

Una traduzione veloce degli indirizzi: il TLB

Poiché le tabelle delle pagine risiedono nella memoria principale, ciascun accesso in memoria da parte di un programma potrebbe richiedere il doppio del tempo: un accesso in memoria per ottenere l'indirizzo fisico ed un secondo accesso per caricare il dato. Il fattore chiave per migliorare le prestazioni di accesso è fare affidamento sulla località dei riferimenti alla tabella delle pagine: quando viene calcolata la traduzione di un numero di pagina virtuale il risultato sarà probabilmente richiesto ancora nel futuro immediato, poiché i riferimenti alle parole della stessa pagina hanno località sia spaziale che temporale.

Di conseguenza i calcolatori moderni contengono una cache speciale per tenere traccia delle traduzioni effettuate di recente: questa cache delle traduzioni è tradizionalmente denominata *translation-lookaside buffer* (*TLB*, buffer di supporto alle traduzioni). Il TLB corrisponde al fogliettino di carta che si usa tipicamente per trascrivere le collocazioni dei libri della biblioteca che si sono ricercati nel catalogo a schede: anziché cercare in continuazione nell'intero catalogo, si memorizza la collocazione di alcuni libri e si usa il pezzo di carta come una cache.

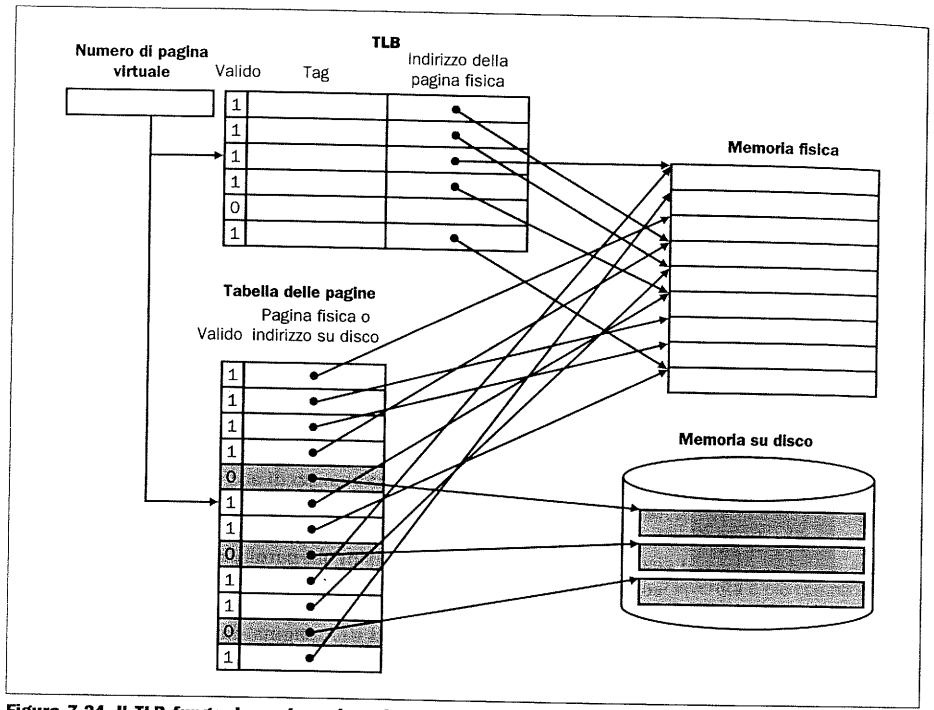


Figura 7.24 Il TLB funge da cache nei confronti della tabella delle pagine per i soli elementi che corrispondono alla memoria fisica. Il TLB contiene un sottoinsieme delle corrispondenze pagina virtuale-pagina fisica presenti nella tabella delle pagine. Le corrispondenze date dal TLB sono evidenziate in tratto colorato. Poiché il TLB è una cache, deve disporre di un campo tag; se nel TLB non vi è l'elemento che corrisponde ad una data pagina, occorre esaminare la tabella delle pagine. La tabella delle pagine fornisce un numero di pagina fisica per la pagina richiesta (che può essere usato per riempire un elemento del TLB) o indica che la pagina risiede su disco, nel qual caso avviene un page fault. Poiché la tabella delle pagine ha un elemento per tutte le pagine virtuali (in altre parole *non* è una cache), non è necessario il campo tag.

Il TLB è una cache che contiene solo le corrispondenze della tabella delle pagine, per cui il campo tag nel TLB contiene una porzione del numero di pagina virtuale, mentre il campo dati contiene un numero di pagina fisica. Poiché non sarà più necessario accedere alla tabella delle pagine ad ogni riferimento in memoria, facendo invece accesso al TLB, quest'ultimo dovrà includere dei bit aggiuntivi, come il bit di riferimento ed il bit di modifica. La figura 7.24 mostra come il TLB agisca come una cache per gli accessi alla tabella delle pagine.

Per ciascun riferimento si consulta il numero di pagina virtuale nel TLB: se il valore è presente il numero di pagina fisica viene usato per formare l'indirizzo ed il bit di riferimento corrispondente viene posto ad uno. Se il processore sta effettuando una scrittura si pone ad uno anche il bit di modifica. Se si verifica un miss nel TLB, occorre determinare se si è di fronte ad un page fault o semplicemente ad un miss nel TLB: se la pagina è presente in memoria, allora la situazione di miss nel TLB indica semplicemente che è assente la traduzione, per cui la CPU può risolvere il riferimento caricando nel TLB la traduzione presa dalla tabella delle pagine e

ripetendo l'accesso. Se la pagina non è presente in memoria, una condizione di miss nel TLB indica un reale page fault; in questo caso la CPU invoca il sistema operativo per mezzo di un'eccezione. Poiché il TLB ha un numero di elementi molto minore del numero di pagine nella memoria principale, i miss nel TLB saranno molto più frequenti dei reali page fault.

I fallimenti di accesso nel TLB possono essere gestiti via hardware o via software; in pratica vi è poca differenza di prestazioni tra i due approcci poiché le operazioni principali da compiere sono le stesse in entrambi i casi.

Non appena si verifica un miss nel TLB e non appena si è letta la traduzione corrispondente dalla tabella delle pagine, occorre scegliere un elemento del TLB da sostituire. Poiché i bit di riferimento e di modifica sono contenuti nell'elemento del TLB occorre ricopiare questi bit nell'elemento corrispondente della tabella delle pagine nel momento in cui si sostituisce un elemento. Questi bit sono l'unica parte del TLB che possano venire modificati; l'uso di una strategia di tipo write-back (ossia ricopiando tali bit solo quando si verifica un miss anziché ad ogni scrittura) si rivela molto efficiente, poiché la frequenza di miss che ci si aspetta dal TLB è bassa. Alcuni calcolatori usano altre tecniche per approssimare i bit di riferimento ed i bit di modifica, eliminando quindi la necessità di scrivere nel TLB eccezion fatta per il caricamento di un nuovo elemento della tabella in corrispondenza di un miss.

Alcuni valori tipici relativi al TLB possono essere i seguenti:

- Dimensione del TLB: 32– 4096 elementi
- Dimensione del blocco: 1–2 elementi della tabella delle pagine (tipicamente di 4-8 byte ciascuno)
- Tempo di hit: 0,5–1 ciclo di clock
- Penalità di miss: 10–30 cicli di clock
- Frequenza di miss: 0,01% –1%.

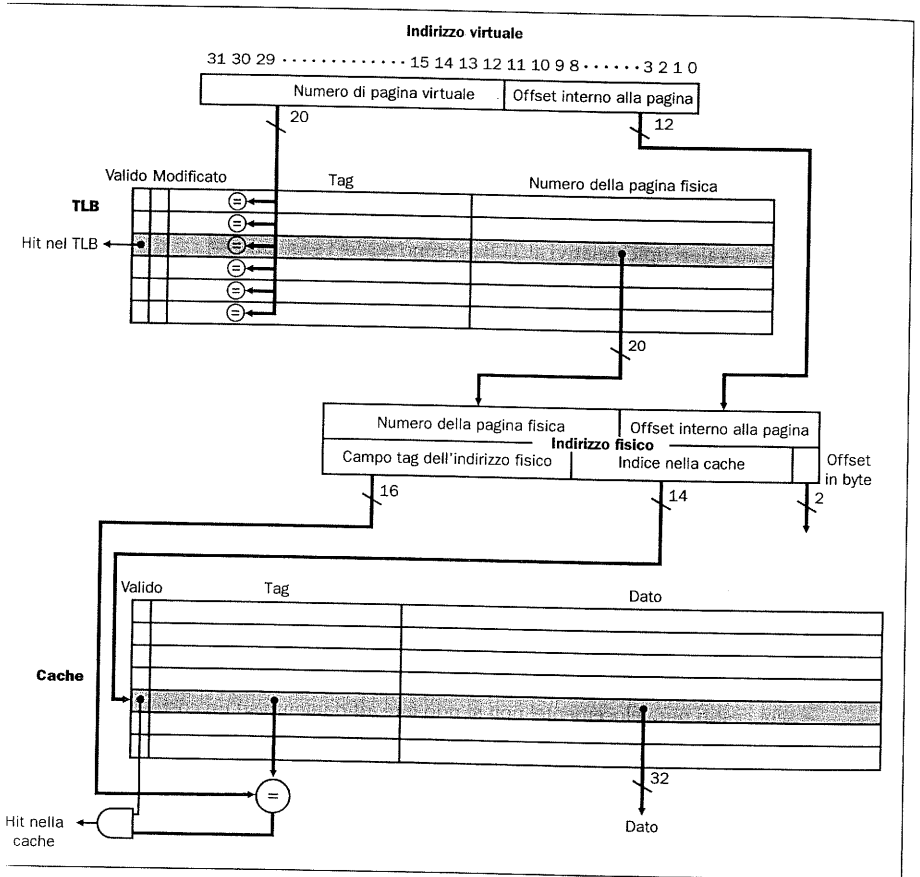
I progettisti hanno sperimentato un ampio spettro di possibili valori di associatività nei TLB. Alcuni sistemi usano dei piccoli TLB completamente associativi poiché la corrispondenza completamente associativa permette una frequenza di miss minore; inoltre, poiché la dimensione del TLB è ridotta, il costo di una corrispondenza completamente associativa è contenuto. Altri elaboratori fanno uso di TLB più capienti, spesso non associativi o con bassa associatività. Con una corrispondenza completamente associativa diviene più complesso sostituire un elemento perché l'implementazione hardware di uno schema LRU è troppo costosa. Inoltre, poiché i miss nel TLB sono molto più frequenti dei page fault e devono quindi essere gestiti più velocemente, non si può fare affidamento su un costoso algoritmo software, come invece si fa per i page fault. In definitiva molti sistemi forniscono la possibilità di scegliere casualmente l'elemento da sostituire. I diversi metodi di sostituzione saranno esaminati in maggior dettaglio nel Paragrafo 7.5.

Il TLB del MIPS R2000

Per analizzare meglio l'applicazione pratica delle tecniche citate, si analizzerà ora il TLB del MIPS R2000, usato nella DECStation 3100; questo TLB, sebbene estremamente semplice, contiene molte delle caratteristiche dei TLB più recenti. Il sottosistema di memoria usa pagine di 4 KB ed uno spazio di indirizzamento di 32 bit; il numero di pagina virtuale è quindi di 20 bit, come mostrato dalla figura 7.22; l'indirizzo fisico ha le stesse dimensioni di quello virtuale. Il TLB contiene 64 elementi,

è completamente associativo ed è condiviso dagli accessi ai dati e da quelli alle istruzioni. Ciascun elemento è lungo 64 bit e contiene un campo tag di 20 bit (pari al numero di pagina virtuale per quell'elemento del TLB), il numero di pagina fisica corrispondente (anch'esso di 20 bit), un bit di validità, un bit di modifica ed altri bit dedicati a vari compiti di gestione.

La figura 7.25 mostra il TLB ed una delle cache, mentre la figura 7.26 riporta i passi necessari ad elaborare le richieste di lettura o di scrittura. Quando si verifica un miss nel TLB, il processore MIPS salva il numero di pagina del riferimento alla memoria in un registro dedicato e genera un'eccezione; l'eccezione invoca il siste-



igura 7.25 Il TLB e la cache implementano il processo che porta da un indirizzo virtuale ad un dato ella DECStation 3100. La figura riporta l'organizzazione del TLB e di una delle cache della DECStation 3100. Il ~~diagramma~~ tratta l'operazione di lettura, mentre le scritture sono descritte nella figura 7.26. Mentre la cache è a ~~rrispondenza~~ diretta, il TLB è completamente associativo, il che significa che occorre confrontare tutti i campi ~~ag~~ del TLB con il valore dell'indice, poiché l'elemento richiesto può essere ovunque nel TLB. Se il bit di validità ~~l~~ elemento corrispondente vale 1, l'accesso al TLB è un hit, ed il numero di pagina insieme all'offset nella pagi-na formano l'indice che viene utilizzato per accedere alla cache.

diagramma
corrispondenza
tag dell'

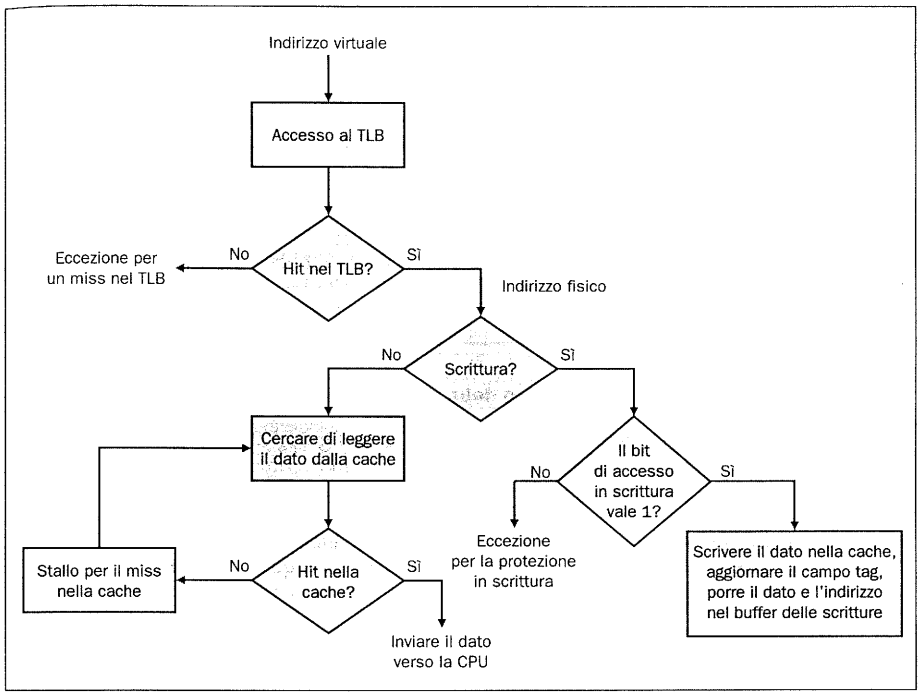


Figura 7.26 Gestione di un'operazione di lettura o di scrittura con il TLB e la cache della DECStation 3100. Se il TLB genera un hit si può accedere alla cache con l'indirizzo fisico corrispondente. Se l'operazione è una scrittura l'elemento della cache viene sovrascritto ed il dato è inviato verso il buffer delle scritture; si ricordi tuttavia che nella cache della DECStation 3100 non può mai verificarsi un miss in scrittura, visto che la dimensione dei blocchi è di una parola e la cache è di tipo write-through. Per le letture, la cache può generare un miss o un hit e quindi fornire il dato oppure causare uno stallo mentre i dati sono prelevati dalla memoria. In realtà il TLB non contiene un vero bit di modifica: usa invece il bit di accesso in scrittura per rilevare la prima scrittura e per aggiornare il bit di modifica nella tabella delle pagine. Si noti che un hit nel TLB ed un hit nella cache sono eventi indipendenti e che l'accesso alla cache non è neppure tentato quando avviene un miss nel TLB. Inoltre un hit nella cache si può solo verificare dopo un hit nel TLB, il che significa che il dato deve essere presente in memoria. La relazione tra i miss nel TLB ed i miss nella cache è esaminata in maggior dettaglio negli esercizi alla fine del capitolo.

ma operativo, il quale gestisce il miss via software. Per trovare l'indirizzo fisico corrispondente alla pagina mancante, la procedura per i miss nel TLB indicizza la tabella delle pagine usando il numero di pagina dell'indirizzo virtuale ed il registro della tabella delle pagine, che indica l'indirizzo di partenza della tabella delle pagine del processo attivo. Facendo uso di uno speciale gruppo di istruzioni di sistema in grado di aggiornare il TLB, il sistema operativo carica l'indirizzo fisico dalla tabella delle pagine nel TLB; se la tabella delle pagine non contiene un indirizzo fisico valido si verifica un vero page fault. Un miss nel TLB può essere risolto anche in soli 10 cicli, ma in media richiede circa 16 cicli. Il processore gestisce via hardware un indice che suggerisce l'elemento che deve essere sostituito; tale elemento viene scelto casualmente.

Vi è una complicazione aggiuntiva relativa alle richieste di scrittura: occorre infatti controllare il bit di accesso in scrittura presente nel TLB: tale bit impedisce al

programma di scrivere nelle pagine alle quali ha solo un accesso in lettura. Se il programma cerca di scrivere ed il bit di accesso in scrittura vale 0, viene generata un'eccezione; il bit di accesso in scrittura fa parte dei meccanismi di protezione, che verranno discussi in seguito.

Integrare memoria virtuale, TLB e cache

Nelle migliori condizioni, un indirizzo virtuale viene tradotto dal TLB ed inviato alla cache, dove il dato relativo viene trovato e restituito alla CPU. Nel caso peggiore un accesso alla memoria potrebbe causare dei miss nelle tre componenti della gerarchia delle memorie: il TLB, la tabella delle pagine e la cache. L'esempio seguente illustra in maggior dettaglio queste interazioni.

Funzionamento complessivo della gerarchia delle memorie

Esempio Nella gerarchia delle memorie illustrata nella figura 7.25, che comprende un TLB ed una cache organizzate come descritto, un riferimento alla memoria può andare incontro a tre diversi tipi di miss: un miss nella cache, un miss nel TLB ed un page fault. Si considerino tutte le combinazioni dei tre eventi in cui si verificano uno o più miss (vi sono sette possibilità). Per ciascuna possibilità si dica se l'evento può effettivamente verificarsi e sotto quali circostanze.

Soluzione La figura 7.27 riporta le combinazioni possibili ed indica se si possano verificare nella pratica.

I sistemi di memoria virtuale ed i sistemi di cache collaborano a realizzare la gerarchia, per cui i dati non possono essere nella cache se non sono presenti nella memoria principale. Il sistema operativo gioca un ruolo importante nel supportare questa gerarchia eliminando il contenuto di una pagina dalla cache quando decide di trasferire la pagina su disco. Allo stesso tempo il sistema operativo modifica le

Cache	TLB	Memoria virtuale	Possibile? Se sì, in quali circostanze?
miss	hit	hit	Possibile, anche se la tabella delle pagine non viene mai realmente consultata quando avviene un hit nel TLB.
hit	miss	hit	Si ha un miss nel TLB, ma l'elemento è presente nella tabella delle pagine; si ritenta l'operazione ed il dato è presente nella cache.
miss	miss	hit	Si ha un miss nel TLB, ma l'elemento è presente nella tabella delle pagine; si ritenta l'operazione e si verifica un miss nella cache.
miss	miss	miss	Si ha un miss nel TLB seguito da un page fault; si ritenta l'operazione e si verifica un miss nella cache.
miss	hit	miss	Impossibile: non si può avere una traduzione nel TLB se la pagina non è in memoria.
hit	hit	miss	Impossibile: non si può avere una traduzione nel TLB se la pagina non è in memoria.
hit	miss	miss	Impossibile: non si possono avere dei dati nella cache se la pagina non è in memoria

Figura 7.27 Le combinazioni possibili di eventi nel TLB, nel sistema di memoria virtuale e nella cache. Tre di queste combinazioni sono impossibili ed una è possibile (hit nel TLB, hit nella memoria virtuale, miss nella cache ma non viene mai rilevata.

tabelle delle pagine ed il TLB, in modo che tutti i tentativi di accedere ai dati della pagina generino un page fault.

Approfondimento: La figura 7.27 assume che tutti gli indirizzi di memoria siano tradotti in indirizzi fisici prima di accedere alla cache; la figura 7.25 mostra un'organizzazione del sottosistema di memoria di questo tipo. Con questa organizzazione la cache ha *indici fisici* e *tag fisici* (sia l'indice della cache che il campo tag sono indirizzi fisici e non virtuali). In un sistema di questo tipo, il tempo richiesto per accedere alla memoria, assumendo un hit nella cache, deve prevedere un accesso al TLB ed un accesso alla cache; naturalmente tali accessi possono essere realizzati in pipeline.

In alternativa la CPU può indicizzare la cache con un indirizzo che sia completamente o parzialmente virtuale: in questo caso si parla di *cache indirizzata virtualmente* ed usa indirizzi virtuali nei campi tag, per cui tale tipo di cache ha *indici virtuali* e *tag virtuali*. In questi tipi di cache i circuiti di traduzione degli indirizzi (ad esempio il TLB) sono inutilizzati durante il normale accesso alla cache, poiché questa è pilotata mediante un indirizzo virtuale non ancora tradotto in indirizzo fisico. Quando avviene un miss nella cache, tuttavia, il processore deve tradurre l'indirizzo in un indirizzo fisico in modo da poter prelevare il blocco della cache dalla memoria principale.

Quando si accede alla cache con indirizzi virtuali e le pagine sono condivise tra diversi programmi (che vi potrebbero accedere con indirizzi virtuali diversi) sorge la possibilità che si presenti il fenomeno detto *aliasing*: esso avviene quando lo stesso oggetto ha due nomi; in questo caso si hanno due indirizzi virtuali per la stessa pagina. Tale ambiguità crea un problema poiché una parola di quella pagina potrebbe essere presente nella cache in due diverse posizioni, ciascuna corrispondente ad un diverso indirizzo virtuale: un programma potrebbe quindi scrivere un dato senza che il secondo programma si renda conto che esso è cambiato. Le cache completamente indicizzate virtualmente introducono quindi dei vincoli progettuali sulla cache e sul TLB per ridurre il fenomeno dell'aliasing o impongono al sistema operativo, ed eventualmente all'utente, di prendere dei provvedimenti per garantire che non possano esistere degli alias.

Un compromesso frequente tra queste due soluzioni progettuali è costituito dalle cache indicizzate virtualmente (talvolta usando solo la parte dell'indirizzo corrispondente all'offset nella pagina, che è in realtà un indirizzo fisico poiché non viene tradotto), ma che usano dei campi tag fisici. Si fa quindi accesso in parallelo alla cache ed al TLB, confrontando i campi tag dell'indirizzo fisico provenienti dalla cache con l'indirizzo fisico ottenuto dal TLB. I progetti di questo tipo, indicizzati virtualmente ma con tag fisici, mirano ad ottenere i vantaggi in termini di prestazioni delle cache indicizzate in modo del tutto virtuale mantenendo la semplicità architetturale di una cache indirizzata fisicamente.

Implementazione dei meccanismi di protezione mediante la memoria virtuale

Una delle funzioni più importanti della memoria virtuale è di permettere la condivisione di un'unica memoria principale da parte di più processi, allo stesso tempo fornendo la protezione della memoria tra questi processi e tra i processi ed il sistema operativo. Il meccanismo di protezione deve garantire che, sebbene più processi stiano condividendo la stessa memoria principale, un singolo processo che non rispetta le regole non possa scrivere nello spazio di indirizzamento di un altro processo utente o nel sistema operativo, sia intenzionalmente che accidentalmente. Ad esempio se il programma di gestione dei voti degli studenti viene eseguito su un calcolatore contemporaneamente ai programmi degli studenti del primo corso di programmazione, non si vuole che un programma errato di un principiante modi-

fichi i voti di qualcuno. Il bit di accesso in scrittura del TLB può proteggere una pagina affinché non possa essere scritta.

È necessario anche evitare che un processo possa leggere i dati di un altro processo. Ad esempio, non si vuole che i programmi degli studenti possano leggere i voti mentre questi si trovano nella memoria del processore. Non appena si permette di condividere la memoria principale, occorre fornire ad un processo la possibilità di proteggere i suoi dati dalle letture e dalle scritture da parte di altri processi; in caso contrario la condivisione della memoria si potrebbe rivelare una maledizione più che una benedizione!

Si ricordi che ciascun processo ha il proprio spazio di indirizzamento virtuale, per cui se il sistema operativo organizza le tabelle delle pagine in modo che pagine virtuali indipendenti corrispondano a pagine fisiche diverse, un processo non potrà accedere ai dati di un altro. Ciò richiede ovviamente che i processi utente non possano modificare la corrispondenza data dalla tabella delle pagine: il sistema operativo può garantire la protezione se può vietare ad un processo utente di modificare le proprie tabelle delle pagine. Allo stesso tempo il sistema operativo deve poter modificare le tabelle delle pagine: i due requisiti si possono soddisfare allo stesso tempo semplicemente ponendo le tabelle delle pagine nello spazio di indirizzamento del sistema operativo.

Interfaccia Hardware Software

Per permettere al sistema operativo di implementare la protezione nel sistema di memoria virtuale, le risorse hardware devono fornire almeno le tre caratteristiche minime elencate nel seguito.

1. Supportare almeno due modi di funzionamento che indichino se il processo in esecuzione è un processo utente o un processo del sistema operativo, altrimenti detto processo *kernel* (nucleo), processo *supervisore* o processo *esecutivo*.
2. Far sì che una parte dello stato della CPU possa essere letta ma non scritta da un processo utente: tale parte deve includere il bit che determina il modo utente o supervisore, il puntatore alla tabella delle pagine ed il TLB. Per scrivere tali elementi il processore usa delle istruzioni speciali che sono disponibili solo in modo supervisore.
3. Fornire dei meccanismi grazie ai quali la CPU possa passare dal modo utente al modo supervisore, e viceversa. La prima direzione viene solitamente realizzata mediante una eccezione di tipo *chiamata di sistema* (*system call*), implementata da un'istruzione speciale (nel set di istruzioni MIPS è *syscall*) che trasferisce il controllo ad una locazione predeterminata nello spazio di indirizzamento del supervisore. Come per tutte le altre eccezioni viene salvato il contatore di programma dell'istruzione di chiamata di sistema e la CPU viene posta in modo supervisore. Il ritorno dalla gestione dell'eccezione al modo utente avviene con l'istruzione di ritorno da eccezione (*return from exception*, RFE), che ripristina lo stato del processo che aveva generato l'eccezione.

Mediante i meccanismi illustrati e grazie alla memorizzazione delle tabelle delle pagine nello spazio di indirizzamento del sistema operativo, quest'ultimo può modificare le tabelle delle pagine e vietare ai processi utenti di modificarle, garantendo quindi che un processo utente possa solo accedere allo spazio di memoria ad esso assegnato dal sistema operativo.

Quando i processi devono condividere delle informazioni secondo modalità controllate, il sistema operativo li deve assistere, poiché accedere alle informazioni di un altro processo richiede di modificare la tabella delle pagine del processo che vi deve accedere. Si può usare il bit di accesso in scrittura per limitare la condivisione ad una sola condivisione in lettura; tale bit, come il resto della tabella delle pagine, può essere modificato solo dal sistema operativo. Per permettere ad un processo, ad esempio P1, di leggere una pagina del processo P2, P2 deve chiedere al sistema operativo di creare un elemento della tabella delle pagine per una pagina virtuale dello spazio di indirizzamento di P1 che punti alla stessa pagina fisica che P2 vuole condividere. Il sistema operativo può usare il bit di protezione in scrittura per evitare che P1 possa scrivere dei dati, se P2 lo richiede. Tutti i bit che determinano i diritti di accesso per una pagina devono essere inclusi sia nella tabella delle pagine che nel TLB, poiché la tabella delle pagine viene consultata solo in corrispondenza dei *miss* nel TLB.

Approfondimento: Quando il sistema operativo decide di passare ad eseguire il processo P2 ed era in esecuzione il processo P1 (azione detta *cambio di contesto* o *cambio di processo*), deve garantire che P2 non possa accedere alle tabelle delle pagine di P1 perché ciò potrebbe compromettere la protezione. In assenza di un meccanismo basato su TLB, è sufficiente modificare il registro della tabella delle pagine in modo che punti alla tabella delle pagine di P2 anziché a quella di P1; in presenza del TLB, occorre invalidare gli elementi del TLB che corrispondono a P1, sia per proteggere i dati di P1 che per forzare il TLB a caricare gli elementi relativi a P2. Se la frequenza di cambio di processo fosse elevata ciò potrebbe rivelarsi decisamente inefficiente. Per esempio P2 potrebbe caricare solo alcuni elementi del TLB prima che il sistema operativo torni a P1; sfortunatamente P1 vedrebbe volatilizzati tutti i suoi elementi del TLB e dovrebbe ricaricarli attraverso una serie di miss nel TLB. Il problema nasce poiché gli indirizzi virtuali usati da P1 e da P2 sono gli stessi ed occorre quindi ripulire il TLB per evitare confusione tra tali indirizzi. Un'alternativa diffusa è quella di estendere lo spazio di indirizzamento virtuale per mezzo di un *identificatore di processo* o *identificatore di task*: esso è un piccolo campo che identifica il processo attualmente in esecuzione e viene memorizzato in un registro aggiornato dal sistema operativo quando cambia processo. L'identificatore di processo è concatenato alla parte tag del TLB, in modo che un hit nel TLB avvenga *solo se* sia il numero di pagina che l'identificatore di processo coincidono. Questa soluzione permette di eliminare la necessità di ripulire il TLB, eccetto in rare occasioni.

Possono nascere dei problemi di natura simile anche per la cache, poiché in occasione di un cambio di processo la cache contiene dei dati del processo in esecuzione. I problemi sono diversi per le cache indirizzate fisicamente e per quelle indirizzate virtualmente, ed esistono diverse soluzioni, come quella degli identificatori di processo, che si possono usare per garantire che ciascun processo possa accedere ai propri dati.

Gestione dei page fault e dei miss nel TLB

Nonostante la traduzione di un indirizzo virtuale in uno fisico mediante il TLB sia semplice nel caso di un hit nel TLB, la gestione dei miss nel TLB e dei page fault è più complessa. Un miss nel TLB si verifica quando non vi è alcun elemento nel TLB stesso che corrisponde ad un indirizzo virtuale; i miss nel TLB possono indicare due possibilità diverse:

1. La pagina è presente in memoria, per cui è sufficiente aggiungere l'elemento mancante al TLB.
2. La pagina non è presente in memoria: occorre trasferire il controllo al sistema operativo per trattare il caso di un page fault.

Come si può determinare quale di queste due circostanze si è verificata? Nella gestione del miss nel TLB si deve ricercare un elemento della tabella delle pagine da trasferire nel TLB: se tale elemento ha un bit di validità pari a 0, allora la pagina corrispondente non è in memoria e si verifica un page fault, anziché solo un miss nel TLB. Se il bit di validità vale 1 si può semplicemente prelevare il numero della pagina fisica dall'elemento nella tabella delle pagine ed usarlo per creare il nuovo elemento nel TLB. Un miss nel TLB può essere gestito via software o via hardware poiché richiede solo una breve sequenza di operazioni per copiare un elemento valido della tabella delle pagine dalla memoria al TLB.

La gestione di un page fault richiede invece di usare il meccanismo delle eccezioni per interrompere il processo attivo, trasferire il controllo al sistema operativo ed in seguito riprendere l'esecuzione del processo interrotto; il page fault verrà riconosciuto ad un certo punto del ciclo di clock in cui avviene l'accesso alla memoria. Per rieseguire l'istruzione dopo la risoluzione del page fault occorre salvare il contatore di programma dell'istruzione che lo ha causato; come si era già visto nei Capitoli 5 e 6 si usa il registro EPC (exception program counter) per memorizzare tale valore.

In aggiunta a ciò, la segnalazione dell'eccezione dovuta al page fault deve avvenire entro la fine dello stesso ciclo di clock usato per l'accesso alla memoria, in modo che al ciclo di clock successivo si inizi la gestione dell'eccezione anziché continuare la normale esecuzione delle istruzioni. Se il page fault non venisse riconosciuto in questo ciclo, un'istruzione di load potrebbe sovrascrivere un registro, con risultati disastrosi quando si dovesse rieseguire l'istruzione. Si consideri per esempio l'istruzione `lw $1,0($1)`: il processore deve poter evitare che l'operazione di scrittura del risultato abbia luogo, altrimenti non potrebbe rieseguire correttamente l'istruzione, visto che il contenuto di `$1` sarebbe stato distrutto. Anche per le operazioni di scrittura nasce una complicazione analoga: occorre evitare che una scrittura in memoria venga completata in presenza di un page fault; ciò viene solitamente ottenuto mascherando i segnali di controllo della scrittura destinati alla memoria.

Non appena il processo che ha generato il page fault è stato interrotto ed il sistema operativo ha preso il controllo, quest'ultimo usa il registro Causa per diagnosticare la causa dell'eccezione. Poiché l'eccezione è un page fault, il sistema operativo sa che occorrerà un tempo di elaborazione non trascurabile, per cui salva l'intero stato del processo attivo. Tale stato comprende tutti i registri general-purpose e quelli in virgola mobile, il registro di indirizzo della tabella delle pagine, il registro EPC ed il registro Causa. L'indirizzo virtuale che ha causato il page fault dipende dal fatto che la pagina mancante fosse quella che conteneva l'istruzione o il dato. L'indirizzo dell'istruzione che ha causato il fault è memorizzato in EPC: se il page fault era dovuto all'istruzione, EPC contiene l'indirizzo virtuale della pagina mancante; in caso contrario l'indirizzo virtuale mancante può essere calcolato esaminando l'istruzione (il cui indirizzo è in EPC) e valutando il registro base ed il campo offset.

Quando il sistema operativo ha determinato l'indirizzo virtuale che ha causato il page fault deve compiere tre passi:

1. Consultare l'elemento della tabella delle pagine per l'indirizzo virtuale in questione e determinare la collocazione su disco della relativa pagina.
2. Scegliere una pagina fisica da sostituire; se la pagina prescelta è modificata, occorre scriverla su disco prima di poter trasferire una nuova pagina virtuale nel suo spazio fisico.
3. Iniziare un'operazione di lettura per portare la pagina richiesta dal disco alla pagina fisica prescelta.

Ovviamente l'ultimo passo richiederà milioni di cicli di clock del processore (lo stesso vale per il secondo passo se la pagina è modificata); di conseguenza il sistema operativo sceglierà di assegnare la CPU ad un altro processo in attesa del completamento dell'accesso su disco. Poiché il sistema operativo ha salvato lo stato del processo, può liberamente dare il controllo del processore ad un altro processo.

Quando la lettura da disco della pagina è completata, il sistema operativo può ripristinare lo stato del processo che aveva originariamente causato il page fault e può eseguire l'istruzione di ritorno da eccezione. Tale istruzione riporterà il processore dal modo kernel al modo utente e ripristinerà il valore del contatore di programma: in tal modo il processo utente ripeterà l'esecuzione dell'istruzione che aveva causato il fault, questa volta accedendo alla pagina, e continuerà l'esecuzione.

Far sì che le istruzioni siano *rieseguibili*, in modo da poter gestire l'eccezione e riprendere l'istruzione in seguito, è relativamente facile in un'architettura come quella del MIPS: poiché ciascuna istruzione scrive un solo dato e la scrittura avviene alla fine del ciclo di esecuzione dell'istruzione si può semplicemente evitare che l'istruzione venga completata (evitando di compiere la scrittura) e rieseguire l'istruzione dall'inizio.

Nel caso di processori con set di istruzioni molto più complessi che possono modificare molte locazioni di memoria e scrivere molti dati, è molto più difficile rendere le istruzioni rieseguibili. L'esecuzione di un'istruzione può infatti generare molti page fault a metà dell'istruzione stessa; ad esempio alcuni processori hanno istruzioni per trasferire dei blocchi di memoria che modificano migliaia di parole di dato. In tali sistemi spesso le istruzioni non possono essere rieseguite dall'inizio, come invece avviene per le istruzioni MIPS: l'istruzione deve essere dunque interrotta ed in seguito ripresa a metà della sua esecuzione. Riprendere un'istruzione nel mezzo della sua esecuzione impone di salvare alcune particolari informazioni di stato, gestire l'eccezione ed infine ripristinare tali informazioni di stato. Per far sì che tutto ciò funzioni correttamente è richiesta una cooperazione attenta e dettagliata tra il codice di gestione delle eccezioni nel sistema operativo ed il comportamento dell'hardware.

Interfaccia Hardware Software

Tra il momento in cui si inizia ad eseguire la procedura di gestione delle interruzioni all'interno del sistema operativo ed il momento in cui il sistema operativo ha salvato l'intero stato del processo, il sistema operativo è particolarmente vulnerabile; se si verificasse ad esempio una seconda eccezione mentre il sistema operativo sta ancora elaborando la prima, l'unità di controllo sovrascriverebbe il registro EPC, rendendo impossibile ritornare all'istruzione che aveva causato il page fault! Si possono evitare disastri di questo tipo mediante la possibilità di disabilitare ed abilitare le eccezioni: quando si verifica la prima eccezione, viene posto ad 1 un bit che disabilita tutte le altre eccezioni; ciò può essere fatto mentre si pone ad 1 il bit per il modo supervisore. Il sistema operativo a questo punto salverà quanto basta dello stato per permettergli di poter continuare correttamente se dovesse verificarsi una seconda eccezione (in particolare, il contatore di programma per le eccezioni ed il registro Causa), e potrà poi riabilitare le eccezioni. Questo procedimento garantisce che le eccezioni non causino la perdita dello stato del processore rendendo impossibile la ripresa dell'istruzione interrotta.

Interfaccia Hardware Software

Poiché il TLB è il sottoinsieme della tabella delle pagine a cui si accede ad ogni ciclo di clock, le violazioni di protezione sono interpretabili anche come eccezioni provenienti dal TLB; il sistema operativo può gestire queste situazioni mediante la stessa logica già utilizzata per gestire i miss nel TLB ed i page fault. Il registro Causa assumerà dei valori particolari per indicare le violazioni della protezione (ad esempio un tentativo di effettuare una scrittura quando il bit di accesso in scrittura vale 0) e distinguerle dai miss nel TLB. Il sistema operativo potrà poi accedere al TLB o all'elemento della tabella delle pagine corrispondente a quella pagina virtuale, in modo da esaminare i diritti di accesso del processo e fornire l'opportuna indicazione di errore.

Approfondimento: La gestione delle situazioni di miss nel TLB via software è analoga alla gestione dei page fault: entrambe le situazioni sono identificate dallo stesso evento nel TLB del MIPS R2000. Per velocizzare la gestione dei semplici miss nel TLB, essendo questi molto più frequenti dei veri page fault, si usano due valori diversi nel registro Causa del MIPS. Il primo valore indica che non vi era un elemento corrispondente all'interno del TLB, mentre il secondo indica che l'elemento è presente nel TLB ma la pagina non è presente il memoria (il bit di validità del TLB replica infatti il bit di validità della pagina in memoria). In un processore MIPS R2000/R3000 questi due eventi sono trattati separatamente: poiché l'eccezione per l'assenza dell'elemento nel TLB è molto più frequente, il sistema operativo in corrispondenza di tale eccezione carica il TLB dalla tabella delle pagine senza esaminare l'elemento caricato e fa ripartire l'istruzione. Se l'elemento non è valido si verificherà una seconda eccezione, che il sistema operativo riconosce come page fault: questa tecnica rende più veloce il caso frequente del miss nel TLB, introducendo una lieve penalità nelle prestazioni per il caso meno frequente di un page fault.

Riepilogo

La memoria virtuale è il nome dato al livello della gerarchia delle memorie che gestisce la cache tra la memoria principale ed il disco. La memoria virtuale permette a ciascun programma di espandere il proprio spazio di indirizzamento al di là dei limiti della memoria principale. Inoltre nei sistemi di elaborazione più recenti la memoria virtuale supporta la condivisione della memoria principale tra diversi processi attivi simultaneamente, la cui combinazione richiederebbe molta più memoria di quanta sia presente; per supportare la condivisione la memoria virtuale fornisce anche dei meccanismi di protezione della memoria stessa.

La gestione della gerarchia delle memorie tra la memoria principale ed il disco è resa più complessa dall'elevato costo dei page fault; vi sono diverse tecniche adatte a ridurre la frequenza di miss:

1. I blocchi, detti pagine, sono abbastanza ampi da poter trarre vantaggio dalla località spaziale e quindi ridurre la frequenza di miss.
2. La corrispondenza tra indirizzi virtuali ed indirizzi fisici, implementata mediante la tabella delle pagine, è resa completamente associativa in modo che ogni pagina virtuale possa risiedere ovunque nella memoria fisica.
3. Il sistema operativo usa tecniche particolari, quali la sostituzione LRU ed un bit di riferimento, per determinare quale pagina sostituire.

Le operazioni di scrittura su disco sono costose, pertanto la memoria virtuale adotta uno schema di tipo write-back e tiene traccia del fatto che una pagina sia stata modificata (mediante un bit di modifica) per evitare di scrivere su disco le pagine che non siano variate.

Il meccanismo della memoria virtuale fornisce la traduzione dagli indirizzi virtuali usati dal programma allo spazio degli indirizzi fisici usato nell'accesso alla memoria; tale traduzione permette la condivisione protetta della memoria principale e fornisce diversi benefici aggiuntivi, tra cui la semplificazione dell'allocazione della memoria. Per garantire che i processi siano reciprocamente protetti è necessario che solo il sistema operativo possa modificare le traduzioni degli indirizzi: ai programmi utente non è permesso modificare le tabelle delle pagine. Si può realizzare una condivisione controllata delle pagine tra diversi processi mediante l'intervento del sistema operativo e grazie ai bit di accesso nella tabella delle pagine, che indicano se il programma utente abbia diritto di lettura o di scrittura per la pagina.

Se la CPU dovesse accedere ad una tabella delle pagine residente in memoria per tradurre ogni accesso, l'overhead introdotto dalla memoria virtuale sarebbe troppo elevato: per questo motivo si usa un TLB come cache per le traduzioni fornite dalla tabella delle pagine. Ciascun indirizzo viene quindi tradotto da indirizzo virtuale ad indirizzo fisico mediante le traduzioni presenti nel TLB.

Le cache, la memoria virtuale, il TLB condividono lo stesso insieme di principi e politiche di gestione; il paragrafo seguente presenta tale visione unificata.

7.5 Un quadro di riferimento comune per le gerarchie di memorie

Giunti a questo punto dell'esposizione, si può riconoscere che i diversi tipi di gerarchie di memorie condividono molti aspetti; sebbene molti dettagli differiscano quantitativamente, molte delle politiche e delle caratteristiche che determinano il funzionamento di una gerarchia sono qualitativamente simili. La figura 7.28 riporta le variazioni di alcune delle caratteristiche quantitative delle gerarchie di memorie. Nel resto di questo paragrafo si discuteranno gli aspetti operativi comuni a tutte le gerarchie di memorie e come questi ne determinino il comportamento. Si esamineranno le politiche di gestione mediante quattro domande che riguardano una qualsiasi coppia di livelli adiacenti nella gerarchia delle memorie, anche se per semplicità di userà la terminologia relativa alle cache.

Caratteristica	Valore tipico per le cache	Valore tipico per la memoria paginata	Valore tipico per il TLB
Dimensione totale in blocchi	1000-100000	2000-250000	32-4000
Dimensione totale in kilobyte	8-8000	8000-8000000	0,25-32
Dimensione del blocco in byte	16-256	4000-64000	4-32
Penalità di miss in cicli di clock	10-100	1000000-10000000	10-100
Frequenza di miss	0,1%-10%	0,00001%-0,0001%	0,01%-2%

Figura 7.28 I principali parametri progettuali quantitativi che caratterizzano le tre principali gerarchie di memorie in un elaboratore. I valori riportati erano valori tipici per i rispettivi livelli nel 1997. Il fatto che molti intervalli di variazione siano ampi è anche dovuto alla variazione nel tempo di altri parametri collegati; ad esempio al crescere delle dimensioni della cache per contrastare le maggiori penalità di miss, anche la dimensione dei blocchi deve aumentare.

Domanda 1: Dove si può posizionare un blocco?

Si è visto che si possono posizionare i blocchi nel livello superiore della gerarchia secondo una varietà di schemi, che vanno dalla corrispondenza diretta alla completa associatività. Come già anticipato, l'intera gamma di possibilità può essere interpretata come variazioni della tecnica set-associative, in cui variano il numero di insiemi ed il numero di blocchi per insieme:

Nome della tecnica	Numero di insiemi	Blocchi per insieme
Corrispondenza diretta	numero di blocchi nella cache	1
Set-associative	numero di blocchi nella cache / associatività	associatività (solitamente 2-8)
Completamente associativa	1	numero di blocchi nella cache

Il vantaggio ottenibile da un aumento del grado di associatività è solitamente una riduzione della frequenza di miss; tale fenomeno è dovuto alla riduzione dei miss che competono sulla stessa locazione di memoria. Questo fenomeno verrà esaminato tra breve, dopo aver visto quale può essere l'entità del miglioramento: la figura 7.29 riporta dei dati misurati su un carico di lavoro composto dai benchmark SPEC92 con cache variabili tra 1 KB e 128 KB, con strutture comprese tra la corrispondenza diretta e la tecnica set-associative ad otto vie. I maggiori miglioramenti si ottengono al passare dalla corrispondenza diretta allo schema set-associative a due vie, portando ad una riduzione intorno al 20%-30% della frequenza di miss. Al crescere della dimensione della cache il miglioramento relativo dovuto all'associatività è costante o aumenta lievemente; poiché il valore reale della frequenza di miss decresce al crescere della dimensione della cache la variazione assoluta della frequenza di miss portata dall'associatività si riduce notevolmente per le cache più grandi. Gli svantaggi principali dell'associatività, come già detto, sono un costo maggiore ed un tempo di accesso superiore.

Domanda 2: Come si ritrova un blocco?

Le tecniche per determinare dove si trova un blocco dipendono dalla tecnica di collocazione, che determina il numero di posizioni possibili; si può riassumere la situazione come segue:

Associatività	Metodo di ricerca	Confronti richiesti
Corrispondenza diretta	indicizzazione	1
Set-associative	indicizzazione dell'insieme, ricerca tra gli elementi	grado di associatività
Completamente associativa	ricerca di tutti gli elementi della cache	dimensione della cache
	tabella di ricerca separata	0

La scelta tra la corrispondenza diretta, la tecnica set-associative e l'associatività completa nelle gerarchie di memorie dipende dal costo di un miss rispetto al costo dell'implementazione delle tecniche associative, misurato sia come ritardo che in termini componenti aggiuntivi. In generale non vale la pena implementare elevati gradi di associatività nelle cache poiché il costo dei comparatori cresce mentre la frequenza di miss migliora solo lievemente. Le cache completamente associative non sono proponibili a meno che abbiano dimensioni molto ridotte, nel qual caso il costo dei comparatori non è predominante e dove i miglioramenti sulla frequenza di miss assoluta sono più evidenti.

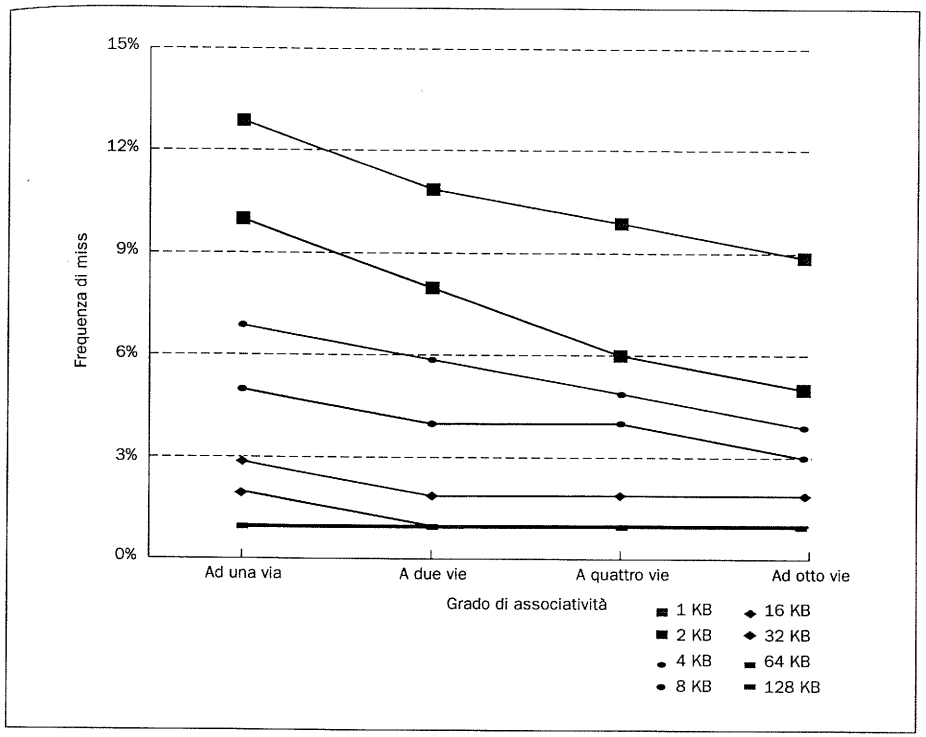


Figura 7.29 Le frequenze di miss relative alle otto dimensioni delle cache si riducono all'aumentare dell'associatività. Mentre si ottiene un beneficio considerevole nel passare da tecniche set-associative ad una via (corrispondenza diretta) a due vie, il beneficio portato da gradi superiori di associatività è più limitato (si ha una riduzione dell'8%-16% passando da due vie a quattro vie mentre il miglioramento da una via a due vie è intorno al 20%-30%). La variazione è ancora minore passando da quattro vie ad otto vie, dove le frequenze di miss sono molto vicine a quelle di una cache completamente associativa. Le cache di dimensioni minori ricevono dei benefici assoluti maggiori dall'aumento dell'associatività, poiché la frequenza di miss di partenza per le cache piccole è maggiore. I dati sono stati calcolati a partire dai benchmark SPEC92 interi e in virgola mobile sempre con un blocco di dimensioni pari a 32 byte. I dati sono stati raccolti da Mark Hill e dai suoi studenti e sono disponibili in rete mediante il link presente all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*.

Nel sistemi di memoria virtuale vi è una tabella di corrispondenza (detta tabella delle pagine) per indicizzare la memoria; tale tabella richiede dello spazio per essere memorizzata, ma soprattutto il suo uso come indice richiede un accesso in memoria aggiuntivo. La scelta dell'associatività completa e la tabella aggiuntiva sono motivate da quattro considerazioni:

1. La completa associatività è vantaggiosa poiché i miss sono *molto* costosi.
2. La completa associatività permette al software di usare tecniche di sostituzione sofisticate, progettate specificatamente per ridurre la frequenza di miss. Si esamineranno tali tecniche tra breve.

3. Una tabella di corrispondenza completa può essere facilmente indicizzata senza necessità di componenti aggiuntivi e senza operazioni di ricerca.
4. La grande dimensione delle pagine implica che l'overhead introdotto dalla tabella delle pagine sia ridotto (l'uso di una tabella di ricerca aggiuntiva, analoga alla tabella delle pagine per la memoria virtuale, non sarebbe opportuna nel caso di una cache, poiché tale tabella sarebbe molto più grande della tabella delle pagine e non potrebbe quindi avere tempi di accesso ridotti).

In definitiva i sistemi di memoria virtuale sono sempre completamente associativi.

Per le cache ed i TLB si usa spesso la tecnica di collocazione set-associative, dove ciascun accesso combina una fase di indicizzazione ed una ricerca in un insieme di dimensioni ridotte. Negli ultimi tempi molti elaboratori usano cache a corrispondenza diretta per via del loro migliore tempo di accesso e della semplicità; il tempo di accesso è minore poiché la ricerca del blocco richiesto non dipende da operazioni di confronto. Tali scelte progettuali dipendono da molti dettagli dell'implementazione, come il fatto che la cache sia sullo stesso chip del processore o su un chip separato, dalla tecnologia usata per implementare la cache e dalla rilevanza del tempo di accesso della cache nella determinazione del tempo di ciclo del processore.

Domanda 3: Quale blocco deve essere sostituito al verificarsi di un miss nella cache?

Quando in una cache associativa si verifica un miss occorre decidere quale blocco sostituire; nelle cache completamente associative tutti i blocchi sono candidati alla sostituzione, mentre in quelle set-associative si può scegliere solo tra i blocchi dell'insieme. La sostituzione non pone ovviamente problemi nelle cache a corrispondenza diretta poiché vi è un solo candidato.

Sono già state citate le due tecniche di sostituzione principali per le cache completamente associative o set-associative:

- *Casuale:* si sceglie casualmente tra i blocchi candidati, talvolta con l'aiuto di un circuito dedicato.
- *Usato meno di recente (LRU, Least Recently Used):* il blocco sostituito è quello che non è stato usato da un tempo maggiore.

In pratica la tecnica LRU non viene implementata per via del costo nel calcolare le informazioni necessarie, eccezion fatta per le gerarchie con un basso grado di associatività (tipicamente da due a quattro). Anche per l'associatività a quattro vie lo schema LRU viene spesso approssimato: ad esempio si tiene traccia di quale coppia di blocchi sia stata usata meno di recente (il che richiede un bit) e si tiene traccia di quale blocco di ciascuna coppia sia stato usato meno di recente (che richiede un bit per coppia). Con valori maggiori dell'associatività la tecnica LRU viene approssimata oppure si usa la sostituzione casuale. Nelle cache l'algoritmo di sostituzione è realizzato in hardware, per cui deve essere facilmente implementabile: la sostituzione casuale ha i necessari requisiti di semplicità e si è visto che in una cache set-associative a due vie la sostituzione casuale genera una frequenza di miss circa 1,1 volte superiore alla sostituzione LRU esatta. Al crescere delle dimensioni della cache per entrambe le tecniche la frequenza di miss si riduce e le differenze assolute divengono trascurabili. In effetti, la sostituzione casuale talvolta fornisce risultati migliori delle approssimazioni, spesso troppo semplificate, degli schemi LRU che si possono implementare in hardware.

Nel caso della memoria virtuale si usa sempre qualche approssimazione della tecnica LRU poiché anche una piccola riduzione della frequenza di miss può essere rilevante, visto l'enorme costo dei miss. Sono spesso disponibili dei bit di riferimento (o altre soluzioni funzionalmente equivalenti) per aiutare il sistema operativo a tenere traccia dell'insieme delle pagine usate meno di recente. Poiché i miss sono molto costosi e relativamente poco frequenti, ha senso calcolare queste informazioni approssimate quasi esclusivamente via software.

Domanda 4: Cosa succede nelle operazioni di scrittura?

Una caratteristica fondamentale delle gerarchie di memorie è il loro comportamento in corrispondenza delle operazioni di scrittura; si sono già viste le due opzioni principali:

- *Write-through*: le informazioni sono scritte sia nel blocco residente nella cache che nel blocco del livello inferiore della gerarchia delle memorie (la memoria principale nel caso di una cache). Le cache descritte nel Paragrafo 7.2 usavano questo schema.
- *Write-back* (anche detto *copy-back*): le informazioni sono scritte solo nel blocco della cache. Il blocco modificato viene scritto al livello inferiore della gerarchia solo quando deve essere sostituito. I sistemi di memoria virtuale usano sempre la tecnica write-back per i motivi esaminati nel Paragrafo 7.4.

Ciascuno dei due schemi ha dei vantaggi; per la tecnica write-back essi sono:

- Le singole parole possono essere scritte dal processore alla frequenza alla quale la cache le può accettare, anziché alla frequenza dettata dalla memoria.
- Le scritture di più parole nello stesso blocco richiedono una sola operazione di scrittura al livello inferiore della gerarchia.
- Quando il blocco viene ricopiato il sistema può sfruttare adeguatamente l'elevata larghezza di banda per i trasferimenti, poiché si scrive l'intero blocco.

I vantaggi della tecnica write-through sono invece:

- I miss sono più semplici e meno costosi poiché non richiedono mai di scrivere un blocco al livello inferiore.
- È più semplice implementare la tecnica write-through rispetto alla write-back, anche se nei sistemi ad alte prestazioni una cache di tipo write-through richiede la presenza di un buffer delle scritture.

Nel caso della memoria virtuale è percorribile solo la strada della politica write-back per via dell'elevata latenza delle scritture al livello inferiore della gerarchia (il disco). Visto che le prestazioni delle CPU continuano a crescere ad un tasso maggiore del miglioramento delle memorie principali basate su DRAM, la frequenza a cui il processore genera le scritture tenderà a superare la frequenza a cui il sottosistema di memoria le può elaborare, anche tenendo conto di memorie fisicamente e logicamente più ampie; di conseguenza nel futuro sempre più cache useranno una strategia di tipo write-back.

Approfondimento: Le operazioni di scrittura introducono per le cache diverse complicazioni non presenti nel caso delle letture; se ne discuteranno qui due: la politica di gestione delle situazioni di miss in scrittura e l'implementazione efficiente delle operazioni di scrittura nelle cache di tipo write-back.

Quadro d'insieme

Mentre cache, TLB e memoria virtuale possono a prima vista apparire molto diversi, sono basati sugli stessi principi di località e possono essere compresi esaminando le rispettive risposte alle quattro domande.

Domanda 1: Dove si può posizionare un blocco?

Risposta: In una posizione (corrispondenza diretta), in alcune posizioni (set-associative) o in qualsiasi posizione (completamente associativa).

Domanda 2: Come si ritrova un blocco?

Risposta: Vi sono quattro metodi: indicizzazione (usata per le cache a corrispondenza diretta), ricerca limitata (come nelle cache set-associative), ricerca completa (come per le cache completamente associative) e mediante tabella di ricerca separata (come nel caso della tabella delle pagine).

Domanda 3: Quale blocco deve essere sostituito al verificarsi di un miss nella cache?

Risposta: Solitamente il blocco usato meno di recente o un blocco scelto casualmente.

Domanda 4: Cosa succede nelle operazioni di scrittura?

Risposta: Ciascun livello della gerarchia può usare lo schema write-through o write-back.

Si consideri un miss in una cache di tipo write-through. La strategia seguita dalla maggior parte dei progetti, detta *fetch-on-miss* (carica all'atto del miss), *fetch-on-write* (carica all'atto della scrittura) o talvolta *allocate-on-miss* (alloca all'atto del miss), alloca un blocco della cache all'indirizzo che ha causato il miss e carica il resto del blocco nella cache prima di scrivere il dato e continuare l'esecuzione. In alternativa si potrebbe allocare il blocco nella cache ma non caricare i dati (detto *no-fetch-on-write*) o non allocare neppure il blocco (*no-allocate-on-write*); un altro nome per queste strategie che non mettono nella cache i dati scritti è *write-around* (scrivi aggirando) poiché i dati "girano intorno" alla cache per arrivare alla memoria. La motivazione dietro a questi schemi risiede nell'osservazione che talvolta i programmi scrivono degli interi blocchi di dati prima di leggerli: in tali casi viene eliminato il caricamento associato al miss in scrittura iniziale. Vi sono diversi aspetti subdoli nell'implementazione di queste tecniche con blocchi di più parole, che verranno discussi nel prossimo capoverso. Si noti che la cache della DEC 3100 è un caso particolare poiché la dimensione del blocco di una parola permette alla cache di realizzare la tecnica allocate-on-write senza dover effettuare alcun caricamento.

Implementare efficientemente le operazioni di scrittura è più complesso per le cache che usano una strategia write-back rispetto alle write-through: nelle cache di tipo write-back si deve scrivere il blocco in memoria se i dati della cache sono stati modificati e se si verifica un miss. Se si scrivesse semplicemente il dato nel blocco senza preventivamente controllare

se l'operazione di scrittura ha causato un hit nella cache (come è invece lecito fare per le cache write-through), si distruggerebbe il contenuto del blocco, per il quale non esiste una copia aggiornata in memoria. Una cache write-through può scrivere il dato nella cache e contemporaneamente leggere il campo tag: se il tag è diverso, si è verificato un miss; poiché la cache è di tipo write-through la sovrascrittura del blocco nella cache non comporta alcuna conseguenza.

Nelle cache di tipo write-back, poiché non è lecito sovrascrivere il blocco, le istruzioni di store richiedono due cicli (un ciclo per controllare se si è verificato un hit, seguito da un altro ciclo per effettuare la scrittura) oppure un buffer aggiuntivo, detto *buffer per le store*, che memorizzi il dato; in pratica si fa sì che la scrittura impieghi un solo ciclo realizzandola in pipeline. Se si usa un buffer per le store, il processore effettua la ricerca nella cache e pone il dato nel buffer durante il normale ciclo di accesso alla cache. Se si verifica un hit il dato viene scritto dal buffer per le store alla cache durante il primo ciclo in cui la cache non viene utilizzata.

Nelle cache write-through, invece, le scritture si possono sempre eseguire in un solo ciclo. Vi sono però delle complicazioni aggiuntive per i blocchi di più parole, poiché non si può semplicemente aggiornare il campo tag quando si scrive un dato: bisogna invece leggere il tag e scrivere la parte di dato del blocco scelto. Se il tag corrisponde all'indirizzo del blocco che viene scritto, il processore può continuare normalmente poiché ha aggiornato il blocco corretto. Se il tag non corrisponde, il processore genera un miss in scrittura per reperire il resto del blocco corrispondente a quell'indirizzo. Poiché è sempre possibile sovrascrivere i dati, gli hit in scrittura richiedono sempre un solo ciclo.

Le tre C: un modello intuitivo per comprendere il comportamento delle gerarchie di memorie

In questo paragrafo si esaminerà un modello che fornisce una buona spiegazione delle cause dei miss nelle gerarchie di memorie e dell'influenza delle modifiche alle gerarchie sul numero di miss. Si esporranno i concetti facendo riferimento alle cache, ma questi sono applicabili direttamente a tutti gli altri livelli della gerarchia. In questo modello ciascun miss è classificato in una delle seguenti categorie (le tre C):

- *Miss Certi*: sono i miss nella cache dovuti al primo accesso ad un blocco che non era mai stato nella cache; sono anche detti *miss di partenza a freddo* o *miss obbligatori*.
- *Miss per Capacità*: sono i miss nella cache che sorgono quando la cache non può contenere tutti i blocchi necessari all'esecuzione del programma; i miss per capacità avvengono perché alcuni blocchi vengono sostituiti e devono essere prelevati più tardi quando vi si farà ancora accesso.
- *Miss per Conflitti*: sono i miss nella cache che avvengono nelle cache set-associative o a corrispondenza diretta quando più blocchi competono per lo stesso insieme; i miss per conflitti sono quelli che avvengono in una cache a corrispondenza diretta o set-associative e che scomparirebbero usando una cache completamente associativa delle stesse dimensioni; sono anche detti *miss per collisioni*.

La figura 7.30 mostra la ripartizione della frequenza di miss tra queste tre cause possibili; ciascuna delle cause può essere affrontata modificando alcuni degli aspetti del progetto della cache. Poiché i miss per conflitti nascono direttamente dalla concorrenza su uno stesso blocco della cache, un aumento di associatività li può ridurre; l'associatività può però aumentare il tempo di accesso, portando a prestazioni complessive inferiori.

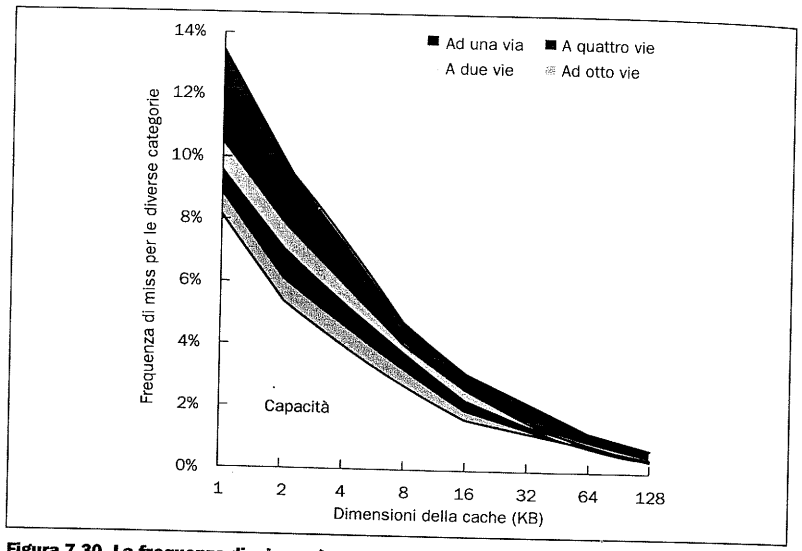


Figura 7.30 La frequenza di miss può essere suddivisa nelle tre cause dei miss. Il grafico riporta la frequenza di miss complessiva e le sue componenti per diverse dimensioni di cache. I dati sono relativi ai benchmark SPEC92 interi e a virgola mobile e provengono dalla stessa fonte della figura 7.29. La componente dei miss certi, per via del lungo tempo di esecuzione dei benchmark SPEC92 e per via della dimensione dei blocchi di 32 byte, è pari allo 0,2% e risulta invisibile in questo grafico. La componente successiva è la frequenza di miss per capacità, che dipende dalla dimensione della cache. La parte di miss per conflitti, che dipende sia dall'associatività che dalla dimensione della cache, è riportata per diversi valori del grado di associatività, da una via ad otto vie. In ciascuno dei casi la parte colorata corrisponde all'incremento della frequenza di miss rilevato quando l'associatività scende dal grado direttamente superiore a quello esaminato. Per esempio la parte relativa all'associatività a due vie indica i miss aggiuntivi rispetto a quelli ottenuti con un grado di associatività di quattro anziché due. La differenza tra la frequenza di miss di una cache a corrispondenza diretta e quella di una cache completamente associativa delle stesse dimensioni è quindi data dalla somma delle aree etichettate come ad otto vie, a quattro vie, a due vie e ad una via.

I miss per capacità si possono ridurre agevolmente aumentando le dimensioni della cache; in effetti, le cache sono costantemente cresciute di dimensioni nel corso degli anni. Quando si rende la cache più ampia occorre tenere ovviamente presente l'incremento nel tempo di accesso, che potrebbe portare ad una riduzione delle prestazioni complessive.

Poiché i miss certi sono generati dal primo riferimento ad un blocco, il metodo principale per ridurne il numero è quello di aumentare la dimensione dei blocchi: in questo modo si riduce il numero di riferimenti necessari a toccare tutti i blocchi del programma almeno una volta, poiché il programma consisterà di un numero minore di blocchi di cache. Un aumento eccessivo della dimensione del blocco potrebbe avere un effetto negativo sulle prestazioni per via dell'aumento della penalità di miss.

La suddivisione dei fallimenti di accesso nelle tre C è un utile modello quantitativo. Nelle cache reali molte scelte progettuali interagiscono e la modifica di una caratteristica della cache influenza spesso diverse componenti della frequenza di miss; nonostante questa lacuna l'applicazione del modello è un buon modo per comprendere meglio le prestazioni dei progetti basati su cache.

Quadro d'insieme

La sfida progettuale nel progetto delle gerarchie di memorie è resa difficile dal fatto che ogni modifica che potenzialmente migliora la frequenza di miss può avere degli effetti negativi sulle prestazioni, come riassunto nella figura 7.31. La combinazione di effetti positivi e negativi rende il progetto della gerarchia delle memorie un compito arduo.

Modifica progettuale	Effetto sulla frequenza di miss	Possibile effetto negativo sulle prestazioni
Aumento delle dimensioni	riduzione dei miss per capacità	può aumentare il tempo di accesso
Aumento dell'associatività	riduzione della frequenza di miss dovuta ai miss per conflitto	può aumentare il tempo di accesso
Aumento della dimensione dei blocchi	riduzione della frequenza di miss in un ampio intervallo di dimensioni dei blocchi	può aumentare la penalità di miss

Figura 7.31 Le sfide nel progetto delle gerarchie di memorie.

7.6 Un caso reale: le gerarchie di memorie del Pentium Pro e del PowerPC 604

In questo paragrafo si esamineranno le gerarchie di memorie di due microprocessori moderni: l'Intel Pentium Pro (PPro) ed il PowerPC (PPC) 604. Nel 1997, il PPro era usato in una gamma di PC da tavolo di fascia alta e di server di diversi produttori, tra cui Dell, DEC, Gateway, HP, Intergraph e Micron; le frequenze di clock di tali prodotti sono comprese tra 150 e 250 MHz. Nel 1997 il PPC 604 era usato in diversi modelli dell'Apple Macintosh (7600, 8500 e 9500) ed in diverse workstation IBM RS/6000 (i prodotti della serie 41 e 42) con frequenze di clock tra i 100 ed i 250 Mhz.

Sia il PPro che il PPC 604 forniscono il supporto per cache di secondo livello esterne al chip principale della CPU. In un certo senso il PPro è unico poiché usa una cache secondaria di 256 KB o 512 KB su un die separato integrato nello stesso package, come mostrato in figura 7.32; tale organizzazione permette di ridurre il tempo di accesso alla cache secondaria e limita il numero di piedini sul package, perché i piedini relativi alla cache secondaria rimangono interni al package. Il PPC usa delle SRAM più convenzionali in package separati; i sistemi Apple Power

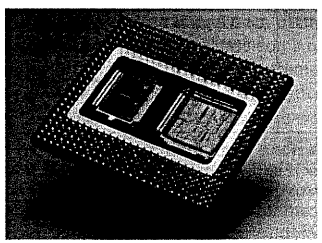


Figura 7.32 Il processore Intel Pentium Pro con il chip per la cache secondaria (a sinistra) nello stesso package del processore (a destra). Fotografia gentilmente concessa da Intel.

Macintosh 7600 e 8500 usano cache secondarie di 256 KB, mentre la serie 9500 adotta cache secondarie di 512 KB. Le organizzazioni delle cache di secondo livello (L2) sono in una certa misura flessibili, ma le dimensioni citate sono quelle tipiche.

Tecniche aggiuntive per ridurre le penalità di miss

Sia il Pentium Pro che il PowerPC 604 dispongono di ottimizzazioni aggiuntive che permettono loro di ridurre le penalità di miss; la prima di tali ottimizzazioni è costituita dalla restituzione anticipata della parola richiesta in corrispondenza di un miss, come illustrato dall'approfondimento di pagina 491. Inoltre il PPC 604 implementa delle ottimizzazioni simili a quelle descritte negli approfondimenti del Paragrafo 7.2, denominate stallo all'utilizzo e ripresa immediata, che permettono al processore di continuare ad eseguire le istruzioni anche in presenza di miss nella cache e di riprendere l'esecuzione non appena la parola critica è trasferita nella cache.

Il PPro va lievemente oltre queste ottimizzazioni e permette al processore di continuare ad eseguire le istruzioni che accedono alla cache dei dati durante la gestione di un miss, al contrario di altre tecniche (come la ripresa immediata e lo stallo all'utilizzo) che permettono di eseguire solo le istruzioni che non usano la cache dei dati. Questa tecnica, denominata *cache non bloccante*, sta divenendo ampiamente diffusa tra i progettisti che vogliono nascondere la latenza dei miss nella cache. Il PPro implementa le due possibili varianti di cache non bloccante: la variante *hit nel miss* permette di gestire degli hit nella cache durante l'elaborazione di un miss, mentre la *miss nel miss* permette di avere più di un miss pendente nella cache. La prima variante mira a nascondere una parte della latenza di miss con del lavoro utile, mentre la seconda mira a sovrapporre temporalmente i periodi di latenza di due diversi miss.

Le gerarchie di memorie del Pentium Pro e del PowerPC 604

Il PPro ed il PPC 604 differiscono nel meccanismo di traduzione degli indirizzi e tali differenze si propagano ai rispettivi TLB, come indicato in figura 7.33. L'architettura del PowerPC ha un indirizzo virtuale di 52 bit, più ampio dello spazio di indirizzamento di 32 bit del PPro.

Caratteristica	Intel Pentium Pro	PowerPC 604
Indirizzi virtuali	32 bit	52 bit
Indirizzi fisici	32 bit	32 bit
Dimensione delle pagine	4 KB, 4 MB	4 KB, selezionabile, 256 MB
Organizzazione del TLB	Un TLB per le istruzioni ed uno per i dati Entrambi set-associative a due vie Sostituzione pseudo-LRU TLB delle istruzioni: 32 elementi TLB dei dati: 64 elementi Miss nel TLB gestiti via hardware	Un TLB per le istruzioni ed uno per i dati Entrambi set-associative a due vie Sostituzione LRU TLB delle istruzioni: 128 elementi TLB dei dati: 128 elementi Miss nel TLB gestiti via hardware

Figura 7.33 Traduzione degli indirizzi e circuiti del TLB per il Pentium Pro ed il PowerPC 604. I due processori supportano pagine di grandi dimensioni usate per scopi particolari, come per il sistema operativo o per accedere ad un buffer di quadro. La tecnica delle pagine grandi permette di non sprecare un elevato numero di elementi per descrivere un singolo oggetto sempre presente. Il PPC 604 fornisce anche una dimensione variabile delle pagine per poter usare pagine ancora più grandi (sempre potenze di 2).

Caratteristica	Intel Pentium Pro	PowerPC 604
Organizzazione della cache	Cache istruzioni e dati separate	Cache istruzioni e dati separate
Dimensione della cache	8 KB per ciascuna cache	16 KB per ciascuna cache
Associatività della cache	Set-associative a quattro vie	Set-associative a quattro vie
Sostituzione	Sostituzione LRU approssimata	Sostituzione LRU
Dimensione dei blocchi	32 byte	32 byte
Gestione delle operazioni di scrittura	Write-back	Write-back o write-through

Figura 7.34 Cache di primo livello nel Pentium Pro e nel PowerPC 604. Le cache principali dei due processori hanno tag e indici fisici, come tutte le cache esaminate in questo capitolo; le soluzioni alternative sono discusse nell'approfondimento legato alla figura 7.27. Le cache di secondo livello del PPro e del PowerPC 604 contengono sia codice che dati e sono ampie 256 KB o 512 KB.

A livello della cache principale, il PPC 604 ed il PPro differiscono come dimensioni e come ottimizzazioni adottate per ridurre la penalità di miss, come si è appena visto; a parte tali differenze le cache principali dei due processori sono molto simili, come evidenziato in figura 7.34.

Le gerarchie di memorie sofisticate del PowerPC 604 e del Pentium Pro riflettono il notevole sforzo progettuale speso nel cercare di tenere sotto controllo il divario tra il periodo di clock del processore ed il tempo di ciclo della memoria. I futuri sviluppi delle pipeline dei processori e la crescente adozione di sistemi a più processori (che presentano tutta una serie di problemi legati alle gerarchie di memorie) forniranno ai progettisti molte sfide interessanti.

Approfondimento: Molte sono le sfide che i progettisti dei sottosistemi di memoria devono affrontare per i processori ad alte prestazioni; questo approfondimento ne discute tre, presenti nel PPro e/o nel PPC 604: il supporto di più accessi alla memoria per ciclo di clock, lo sfruttamento efficiente delle cache non bloccanti e l'implementazione efficiente delle cache di tipo write-back.

Una delle sfide più importanti riguardanti il progetto delle cache è il supporto di processori che vogliano eseguire più di un'istruzione con accesso a memoria per ciclo di clock; ad esempio, la struttura a pipeline del Pentium Pro permette di eseguire un'operazione di load ed una di store ad ogni ciclo di clock. La cache di primo livello può supportare le richieste multiple ricorrendo a due diverse tecniche: innanzitutto la cache può avere più porte (come il register file), permettendo più accessi contemporanei allo stesso blocco della cache. Costruire una cache a più porte tuttavia è spesso troppo costoso, in quanto le celle di RAM in una memoria a più porte devono essere molto più grandi delle celle per RAM ad una porta. In pratica questo approccio è usato solo per cache di ridotte dimensioni (qualche decina di elementi). La soluzione alternativa, usata nel Pentium Pro, prevede di dividere la cache in banchi e di permettere accessi multipli ed indipendenti (una load ed una store) a patto che siano su banchi diversi. Se si verifica un conflitto, la load ha precedenza rispetto alla store, poiché è spesso più critica; in caso di conflitto la presenza di un buffer per le store evita uno stallo del processore, in modo analogo a quanto avviene con il buffer per le scritture.

Una seconda sfida notevole è l'integrazione e lo sfruttamento delle possibilità offerte dalle cache non bloccanti; per ottenere prestazioni significative dalle cache non bloccanti, come nel caso del Pentium Pro, occorre ricorrere all'esecuzione "non ordinata" (out-of-order) delle istruzioni, descritta nell'ultimo capitolo. Senza tale possibilità il processore non riuscirebbe a nascondere una buona parte della latenza di miss, poiché entrerebbe in stallo in corrispondenza dell'istruzione che deve usare il dato poco dopo aver scoperto il miss. Con l'esecuzione fuori ordine, invece, il processore può eseguire altre istruzioni durante il tempo di miss; la capacità di continuare l'esecuzione può sia nascondere la latenza di miss (gestendo gli hit nei miss) che accumulare degli ulteriori miss (gestendo i miss nei miss). Per poter sovrapporre una buona parte del tempo di miss di due miss pendenti occorre disporre di un

sottosistema di memoria a larga banda capace di gestire più miss in parallelo. Nei calcolatori da tavolo la memoria potrà trarre solo un piccolo vantaggio da tale capacità, ma i server più grandi ed i sistemi a più processori spesso dispongono di sottosistemi di memoria capaci di gestire più miss pendenti in parallelo.

Poiché le cache di tipo write-back stanno divenendo la norma, la loro implementazione efficiente diviene sempre più importante. Nella discussione delle cache di tipo write-through si era rilevata la necessità dei buffer delle scritture per renderle utilizzabili praticamente; anche molte cache di tipo write-back comprendono dei buffer delle scritture per ridurre la penalità di miss quando in seguito ad un miss occorre sostituire un blocco modificato. In tal caso anziché scrivere il blocco modificato nella memoria e leggere il blocco richiesto (forzando il processore ad uno stallo di due cicli per l'accesso alla memoria), il blocco modificato è copiato in un buffer di ricopiatura associato alla cache, il blocco richiesto viene letto dalla memoria e si riprende l'esecuzione (con uno stallo di un solo ciclo di accesso alla memoria). Il contenuto del buffer di ricopiatura viene poi ricopiato nella memoria; se non si verifica immediatamente un secondo miss questa tecnica può dimezzare la penalità di miss quando occorre sostituire dei blocchi modificati. L'Intel PPro dispone di un tal buffer di ricopiatura sia per la cache principale che per quella secondaria.

7.7 Errori e trabocchetti

La gerarchia delle memorie è uno degli aspetti più quantitativi delle architetture dei sistemi di elaborazione, e pertanto potrebbe sembrare meno vulnerabile ad errori e trabocchetti. Al contrario, non solo sono proliferati molti errori e si è incappati in molti trabocchetti, ma alcuni di questi hanno portato a risultati fortemente negativi. Si esaminerà per primo un trabocchetto che cattura spesso gli studenti durante la soluzione di esercizi o di esami.

Trabocchetto: dimenticare di tenere conto dell'indirizzamento dei byte o della dimensione dei blocchi della cache nel simulare le cache.

Nella simulazione (manuale o automatica) delle cache occorre accertarsi di tenere conto dell'effetto dell'indirizzamento dei byte o dei blocchi di più parole nel determinare quale sia il blocco della cache corrispondente ad un dato indirizzo. Ad esempio, avendo una cache a corrispondenza diretta di 32 byte con blocchi di 4 byte, l'indirizzo del byte 36 corrisponde al blocco di indirizzo 9 e (9 modulo 8) = 1. D'altra parte se l'indirizzo 36 fosse un indirizzo di parola corrisponderebbe al blocco (36 mod 8) = 4. Ci si accerti che il problema specifichi chiaramente la base rispetto alla quale è espresso l'indirizzo.

In modo simile occorre tenere conto della dimensione dei blocchi: si ipotizzi di avere una cache di 256 byte ed un blocco di 32 byte; in quale blocco cade l'indirizzo 300 espresso in byte? L'indirizzo in byte 300 è l'indirizzo di blocco

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

Il numero di blocchi nella cache è

$$\left\lfloor \frac{256}{32} \right\rfloor = 8$$

Il blocco numero 9 cade dunque nel blocco della cache numero (9 modulo 8) = 1.

Questo errore confonde molte persone, compresi gli autori di questo testo (nelle versioni preliminari) e gli insegnanti che dimenticano di specificare se gli indirizzi

sono da intendersi in parole, byte o numero di blocchi. Si ricordi questo trabocchetto nello svolgimento degli esercizi.

Trabocchetto: usare la frequenza di miss come unica metrica per valutare la gerarchia delle memorie.

Come appena discusso, la frequenza di miss può essere una metrica fuorviante se si ignorano gli altri parametri di una cache. Si consideri un esempio specifico: si supponga di eseguire il carico di lavoro della figura 7.29; un incremento della dimensione della cache a corrispondenza diretta da 16 KB a 32 KB riduce la frequenza di miss da 3,0% a circa 2,0% nel caso di una cache set-associative a due vie. Si supponga che il calcolatore con la cache più grande abbia un periodo di clock di 2 ns, mentre quello con la cache più piccola abbia un clock di periodo 1,6 ns; si supponga inoltre che il valore di CPI in assenza di stalli per la memoria sia invariato. Se la penalità di miss per la cache secondaria è di 20 ns e vi sono 1,5 riferimenti per istruzione (un riferimento all'istruzione e 0,5 ai dati), il sistema con la cache più grande è in realtà più lento, nonostante la sua migliore frequenza di hit. Per comprendere questo fenomeno si usi la seguente equazione:

$$\text{tempo di CPU} = (\text{cicli di esecuzione della CPU} + \text{cicli di clock di stallo per la memoria}) \times \text{periodo del clock}$$

dove i cicli di stallo per la memoria sono calcolati in base all'equazione già vista:

$$\text{cicli di clock di stallo per la memoria} = \frac{\text{istruzioni}}{\text{programma}} \times \frac{\text{numero di miss}}{\text{istruzione}} \times \text{penalità di miss}$$

Il termine *miss per istruzione* combina le frequenze di miss delle istruzioni e dei dati in un singolo termine

$$\frac{\text{numero di miss}}{\text{istruzione}} = \text{frequenza di miss per le istruzioni} + \left(\text{frequenza di miss per i dati} \times \frac{\text{riferimenti ai dati}}{\text{istruzione}}\right)$$

Per la cache di minori dimensioni (dove I indica le *istruzioni per programma*),

$$\text{cicli di clock di stallo per la memoria} = I \times (3\% \times 1{,}5) \times \left\lceil \frac{\text{penalità di miss assoluta}}{\text{periodo di clock}} \right\rceil$$

$$\text{cicli di clock di stallo per la memoria} = I \times 0{,}045 \times \left\lceil \frac{20}{1{,}6} \right\rceil = 0{,}585 \times I$$

Per il sistema con la cache di maggiori dimensioni,

$$\text{cicli di clock di stallo per la memoria} = I \times (2\% \times 1{,}5) \times \left\lceil \frac{\text{penalità di miss assoluta}}{\text{periodo di clock}} \right\rceil$$

$$\text{cicli di clock di stallo per la memoria} = I \times 0{,}030 \times \left\lceil \frac{20}{2} \right\rceil = 0{,}30 \times I$$

Questi risultati si possono ora comporre nell'equazione per il tempo di CPU. Si indichi con C il CPI in assenza di stalli per la memoria; il numero di cicli di clock di esecuzione della CPU sarà C × I. Questa considerazione porta al seguente tempo di esecuzione della CPU per il sistema con la cache più piccola:

$$\text{tempo di CPU} = (\text{cicli di esecuzione della CPU} + \text{cicli di clock di stallo per la memoria}) \times \text{periodo del clock}$$

$$\text{tempo di CPU} = ((C \times I) + (0{,}585 \times I)) \times 1{,}6 \text{ ns} = (1{,}6C + 0{,}936) \times I$$

Nel caso della cache più grande si ottiene

$$\text{tempo di CPU} = ((C \times I) + (0{,}30 \times I)) \times 2 \text{ ns} = (2C + 0{,}6) \times I$$

Il calcolatore con la cache più grande sarà più veloce se (1,6C+0,936) > (2C+0,6), cioè se (C < 0,84). In calcolatori aventi un CPI in presenza di pipeline superiore a 0,84, il sistema più veloce è quello con la cache più piccola.

Anche se sembrerebbe ovvio che sia un errore concentrarsi sulla frequenza di miss ed ignorare l'impatto del progetto della cache sulla frequenza di clock, molti progettisti in passato hanno lavorato principalmente in funzione della frequenza di miss, ignorandone le implicazioni sul periodo di clock.

Trabocchetto: ignorare il comportamento del sottosistema di memoria nello scrivere i programmi o nel generare il codice in un compilatore.

Questo trabocchetto potrebbe anche essere espresso come un errore: "I programmatori possono tranquillamente ignorare le gerarchie di memorie nello scrivere il codice". Il trabocchetto verrà qui illustrato con un esempio di moltiplicazione di matrici, ma si potrebbero usare molti altri esempi.

Quello seguente è il ciclo interno della moltiplicazione di matrice vista nel Capitolo 4:

```
for (i=0; i!=500; i=i+1)
   for (j=0; j!=500; j=j+1)
      for (k=0; k!=500; k=k+1)
         x[i][j] = x[i][j] + y[i][k]*z[k][j];
```

Se il programma viene eseguito con matrici 500 × 500 in doppia precisione, il tempo di CPU su una Silicon Graphics Challenge L che contiene un processore MIPS R4000 con una cache di secondo livello di 1 MB è di 77,2 secondi. Se l'ordine dei cicli viene modificato in `k`, `j`, `i` (in modo che `i` sia il più interno) il tempo di esecuzione crolla a 44,2 secondi! L'unica differenza è la modalità con cui il programma accede alla memoria ed il conseguente effetto sulla gerarchia delle memorie. Altre ottimizzazioni con una tecnica di riduzione in blocchi (detta *blocking*) possono portare ad un tempo di esecuzione inferiore ai 10 secondi per lo stesso codice! Tale ottimizzazione era alla base dei risultati sul benchmark matrix300 discussi nel Capitolo 2.

Trabocchetto: estendere lo spazio di indirizzamento aggiungendo i segmenti ad uno spazio di indirizzamento non segmentato.

Negli anni '70 molti programmi iniziarono a crescere fino al punto da non poter contenere il codice ed i dati disponendo di indirizzi di soli 16 bit. Si riprogettarono quindi i processori in modo da offrire indirizzi di 32 bit, o mediante uno spazio di

indirizzamento non segmentato di 32 bit (detto anche *spazio di indirizzamento piatto*), o aggiungendo 16 bit per specificare un segmento agli esistenti indirizzi di 16 bit. Da un punto di vista di mercato l'aggiunta di segmenti visibili al programmatore che forzavano programmatori e compilatori a decomporre i programmi in segmenti poteva risolvere il problema dell'indirizzamento; sfortunatamente in questo modo nascono dei problemi ogni volta che un linguaggio di programmazione vuole un indirizzo più grande di un segmento, come per gli indici di matrici di grandi dimensioni, puntatori senza restrizioni o parametri passati per riferimento. Inoltre l'aggiunta dei segmenti può spezzare tutti gli indirizzi in due parole, una per il segmento ed una per l'offset all'interno del segmento, causando dei problemi nell'uso degli indirizzi nei registri. Alla data di completamento di questo libro si stanno raggiungendo i limiti imposti dagli indirizzi espressi su 32 bit. Alcune architetture, come la MIPS R4000, la DEC Alpha e la Sun SPARC hanno scelto di supportare spazi di indirizzamento piatti a 64 bit. Altre, come la HP PA-RISC, forniscono uno spazio di indirizzamento esteso mediante segmentazione, come soluzione temporanea; altre architetture ancora, come la 80x86, saranno probabilmente rimpiazzate da una nuova architettura a 64 bit.

7.8 Note conclusive

La difficoltà di costruire un sottosistema di memoria che tenga il passo con le CPU sempre più veloci è sottolineata dal fatto che la materia prima per costruire la memoria principale, composta dai dispositivi DRAM, è essenzialmente la stessa sui calcolatori più veloci e su quelli più lenti ed economici. È solamente il principio di località che può fornire un appiglio per superare la lunga latenza degli accessi in memoria, e la fondatezza di una tale strategia è stata dimostrata a tutti i livelli della gerarchia delle memorie. Sebbene i diversi livelli della gerarchia sembrino qualitativamente molto diversi, in realtà seguono strategie simili nel loro operato e sfruttano le stesse proprietà di località.

Poiché le velocità delle CPU continuano a migliorare più rapidamente dei tempi di accesso alle DRAM o dei tempi di accesso al disco, la memoria diverrà sempre più il fattore limitante per le prestazioni. I processori continuano a migliorare le prestazioni con un tasso impressionante e le DRAM sembrano continuare sulla strada dell'incremento di densità di quattro volte ogni tre anni; tuttavia il *tempo di accesso* delle DRAM viene migliorato ad un tasso molto minore, intorno al 9% all'anno. La figura 7.35 riporta le stime (ottimistiche e pessimistiche) del periodo di clock del processore rispetto all'incremento stabile del 9% annuo sulla velocità delle DRAM. I dati riportati tengono conto solo dell'effetto della riduzione del periodo di clock del processore, ma vi è anche una tendenza alla riduzione del fattore prestazionale CPI, portando ad un numero maggiore di accessi per ciclo e ad una maggiore penalità relativa per la lentezza dell'accesso alla memoria.

I recenti miglioramenti nella tecnologia delle DRAM (le DRAM sincrone e le tecniche ad esse collegate) hanno portato ad un incremento nella larghezza di banda potenziale: tale maggiore banda potenziale ha permesso ai progettisti di aumentare la dimensione dei blocchi delle cache incidendo solo marginalmente sulla penalità di miss. In futuro questi miglioramenti e questi compromessi progettuali saranno un elemento critico nella limitazione della perdita di prestazioni dovuta al sottosistema di memoria.

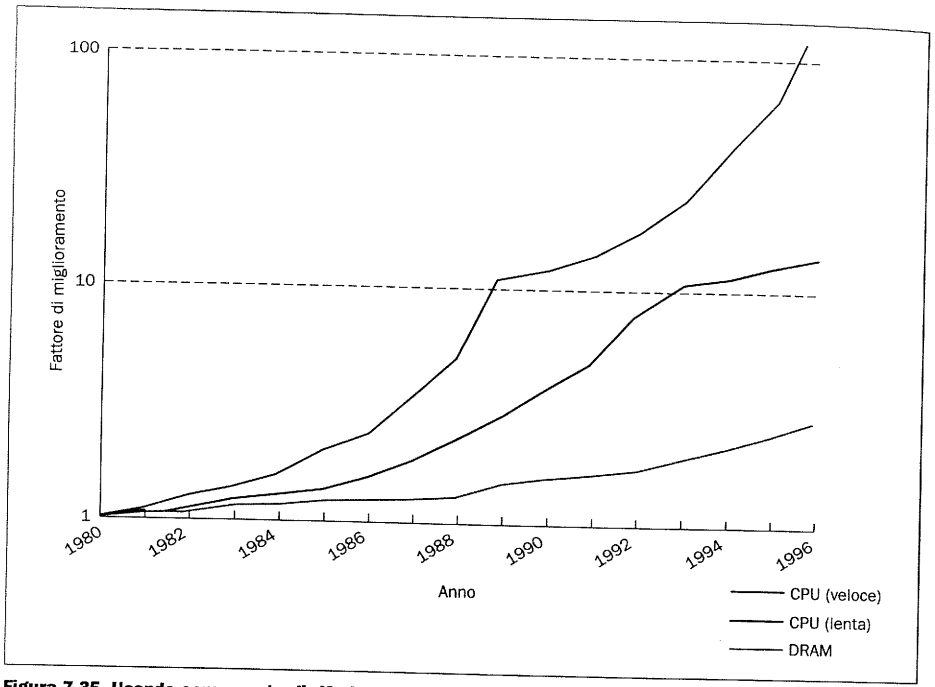


Figura 7.35 Usando come punto di riferimento le prestazioni nel 1980, viene riportato il tempo di accesso delle DRAM rispetto al periodo di clock dei processori in funzione del tempo. La DRAM di riferimento è quella di 64 KB del 1980 ed il miglioramento medio annuale del tempo di accesso è pari al 9%. Le curve che riportano le prestazioni dei processori veloci e lenti sono basate sui tipici processori prodotti nel periodo; il grafico dei processori lenti riporta un miglioramento del 15% annuo nel periodo di clock fino al 1985 e del 25% in seguito, mentre i processori veloci mostrano un miglioramento prestazionale del 25% annuo tra il 1980 ed il 1985 e del 40% successivamente. Naturalmente le prestazioni dei processori sono migliorate anche per via di fattori diversi dalla frequenza di clock: le riduzioni del CPI portano ad un fattore aggiuntivo pari a 15-20 per le CPU veloci nell'arco di tempo considerato. La riduzione del CPI enfatizza ulteriormente le perdite dovute alle prestazioni del sottosistema di memoria, anche se il crescente divario del CPI potrebbe non portare a tempi di accesso maggiori in caso di hit nella cache. Si noti che l'asse verticale deve essere su scala logaritmica per poter rappresentare il divario tra le prestazioni dei processori e delle DRAM.

Tendenze recenti

La sfida progettuale legata alla costruzione di gerarchie di memorie capaci di ricucire questo divario crescente, come si era notato nel Quadro d'insieme al termine del Paragrafo 7.5, risiede nel fatto che tutte le scelte di progetto hardware possibili nelle gerarchie di memorie hanno effetti sia positivi che negativi sulle prestazioni. Questo significa che per ogni livello della gerarchia vi è un punto ottimale per le prestazioni, che deve prevedere una certa quantità di miss. Se ciò è vero, come si può superare il divario crescente tra la velocità delle CPU ed i livelli inferiori della gerarchia? Questa domanda è attualmente oggetto di molti filoni di ricerca.

Le cache di primo livello integrate sul chip hanno in qualche modo contribuito a ridurre il divario crescente tra il periodo di clock della CPU ed il periodo di ciclo delle SRAM su chip separato. Per ridurre la differenza tra le piccole cache sul chip

e la DRAM si sono diffuse le cache di secondo livello: oggigiorno tutti i Personal Computer di medie prestazioni e di fascia alta usano cache di secondo livello; in effetti i server più potenti usano cache di secondo livello di 1-16 MB! Le cache di secondo livello permettono di sfruttare più agevolmente altre ottimizzazioni per due ragioni: innanzitutto i parametri di progetto di una cache di secondo livello sono diversi da quelli della cache di primo livello; ad esempio, essendo la cache di secondo livello più capiente, può usare dei blocchi più grandi. In secondo luogo la cache di secondo livello non viene usata in continuazione dalla CPU, a differenza della cache di primo livello: questo fatto permette di prevedere che la cache di secondo livello possa fare qualcosa mentre non viene utilizzata in modo da poter prevenire dei miss futuri; si vedrà un esempio tra pochissimo.

Un ulteriore tentativo per ridurre il divario di prestazioni tra processori e DRAM è la ridefinizione dell'interfaccia esistente sui chip di DRAM: sono in corso diversi sforzi per progettare un'interfaccia che offra una larghezza di banda molto maggiore delle attuali DRAM, in parte fornendo un clock ai chip di DRAM per sincronizzare i trasferimenti ed in parte aumentando il numero di piedini sulle DRAM. Queste tecniche facilitano il passaggio a blocchi di dimensioni maggiori, in quanto la componente di tempo di trasferimento della penalità di miss viene ridotta. Sebbene queste DRAM attualmente abbiano un prezzo superiore (tipicamente del 10%-30%), sembra che, almeno in alcuni contesti, i progettisti siano disposti a tollerare i maggiori costi. Si è avuto uno sviluppo collegato sull'uso delle SRAM sincrone, che sono usate per le cache di secondo livello, per poter ridurre il tempo necessario a trasferire un blocco dalla cache secondaria a quella principale: le SRAM sincrone stanno divenendo la scelta standard per le cache secondarie.

Un'altra direzione possibile consiste nel chiedere aiuto al software: la gestione efficiente delle gerarchie di memorie mediante una serie di trasformazioni dei programmi e con l'uso di particolari risorse hardware è un importante filone di ricerca nel campo dei compilatori. Si esplorano due diverse idee: da una parte si cerca di riorganizzare il programma in modo da aumentarne la località spaziale e temporale. L'approccio si concentra sui programmi basati fortemente su cicli che usino grandi matrici come struttura dati principale; un esempio tipico è costituito dai problemi di algebra lineare di grandi dimensioni. Ristrutturando i cicli in cui si accede alle matrici si può ottenere una località molto maggiore, e conseguenti migliori prestazioni della cache; l'esempio riportato nel Paragrafo 7.7 mostra quanto possa essere efficace una semplice modifica della struttura dei cicli.

La seconda idea tende ad usare il cosiddetto *pre-caricamento* guidato dal compilatore: il pre-caricamento (in inglese, prefetching) consiste nel caricare un blocco di dati nella cache prima che vi si faccia riferimento. Il compilatore cerca di identificare i blocchi di dati che serviranno in futuro e, usando speciali istruzioni, richiede alla gerarchia delle memorie di portare tali blocchi nella cache. Quando si farà effettivamente riferimento al blocco, questo verrà trovato nella cache e non causerà un miss. L'uso delle cache secondarie ha aumentato l'importanza del pre-caricamento perché la cache di secondo livello può essere impegnata con i pre-caricamenti mentre la cache principale continua a rispondere alle richieste del processore.

Come si vedrà nel Capitolo 9 i sottosistemi di memoria sono anche un elemento progettuale chiave per i processori paralleli. L'importanza crescente delle gerarchie di memorie nella determinazione delle prestazioni dei sistemi sia ad un processore che a più processori significa che quest'area continuerà ad impegnare sia progettisti che ricercatori per i prossimi anni.

7.9 Inquadramento storico e approfondimenti

...l'unica innovazione che da sola fece decollare il mercato degli elaboratori fu l'invenzione di una forma di memoria affidabile, la memoria a nuclei magnetici ... Aveva un costo ragionevole, era affidabile e, essendo affidabile, si poterono costruire in breve tempo memorie di grandi dimensioni.

Maurice Wilkes, *Memories of a Computer Pioneer*, 1985

Lo sviluppo di molti dei concetti trattati in questo capitolo è stato determinato dai miglioramenti rivoluzionari nella tecnologia utilizzata per le memorie. Prima di discutere lo sviluppo storico delle gerarchie di memorie, conviene fare una breve carrellata sulle tecnologie di memorizzazione. Questo paragrafo approfondisce la tecnologia usata per la memoria principale e per le cache, mentre il Capitolo 8 conterrà dei cenni storici sullo sviluppo della tecnologia dei dischi.

Il sistema ENIAC disponeva di un limitato numero di registri (circa 20) per memorizzare i propri dati, implementati con la stessa tecnologia a tubi a vuoto usata per la costruzione dei circuiti logici; tuttavia la tecnologia a tubi a vuoto si rivelò troppo costosa per poter passare a capacità di memoria maggiori. Eckert ebbe l'idea di sviluppare una nuova tecnologia basata su linee di ritardo di mercurio: in tale tecnologia i segnali elettrici erano convertiti in vibrazioni inviate ad un tubo di mercurio, che quando raggiungevano l'estremità opposta venivano lette e rimesse in circolo nel sistema. Ciascuna linea di ritardo di mercurio poteva memorizzare circa 0,5 Kbit; sebbene i bit fossero accessibili solo in modo seriale, le linee di ritardo di mercurio erano circa 100 volte più economiche delle memorie a tubi a vuoto. Le prime linee di ritardo di mercurio funzionanti di cui si ha notizia furono sviluppate a Cambridge per il sistema EDSAC; la figura 7.36 mostra le linee di ritardo di mercurio dell'EDSAC, che disponeva di 32 serbatoi per un totale di 512 parole di 36 bit.

Nonostante l'enorme progresso offerto dalle linee di ritardo di mercurio, esse si rivelarono molto poco affidabili ed ancora troppo costose; la rivoluzione arrivò con l'invenzione della memoria a nuclei da parte di J. Forrester al MIT nell'ambito del progetto Whirlwind, nei primi anni '50 (si veda la figura 7.37). Le memorie a nuclei usano un nucleo di ferrite, che può venire magnetizzato e, una volta magnetizzato, può fungere da memoria (esattamente come un nastro per registrazioni magnetiche memorizza dell'informazione); la lettura dei valori memorizzati nei diversi nuclei era resa possibile da un gruppo di fili che ne attraversano il centro, il quale aveva una dimensione di 0,1-1,0 mm. La versione finale del Whirlwind comprendeva una memoria a nuclei di 2048 parole di 16 bit, per un totale di 32 Kbit. La memoria a nuclei fu un progresso incredibile: era più economica, più veloce, molto più affidabile e permetteva maggiori densità; aveva qualità tanto superiori alle tecnologie alternative da divenire la tecnologia dominante per le memorie solo pochi anni dopo la sua invenzione e da rimanerlo per quasi 20 anni.

La tecnologia capace di sostituire la memoria a nuclei magnetici fu la stessa che viene usata ancora oggi sia per la circuiteria logica che per le memorie: il circuito integrato. Sebbene i registri fossero costruiti con memoria basata su transistor già negli anni '60 e sebbene i sistemi IBM usassero negli anni '70 memoria a transistor per memorizzare il microcodice e per la cache, la costruzione della memoria princi-

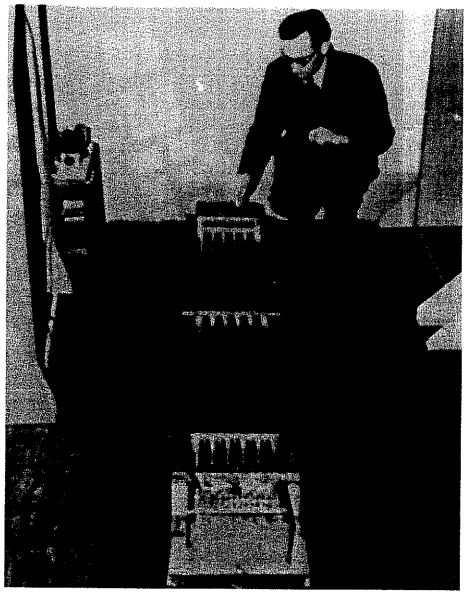


Figura 7.36 Le linee di ritardo di mercurio nel calcolatore EDSAC. Questa tecnologia rese possibile la costruzione del primo calcolatore di tipo stored program; il giovane ingegnere della fotografia non è altri che Maurice Wilkes, l'architetto capo dell'EDSAC. Fotografia gentilmente concessa da Computer Museum, Boston.

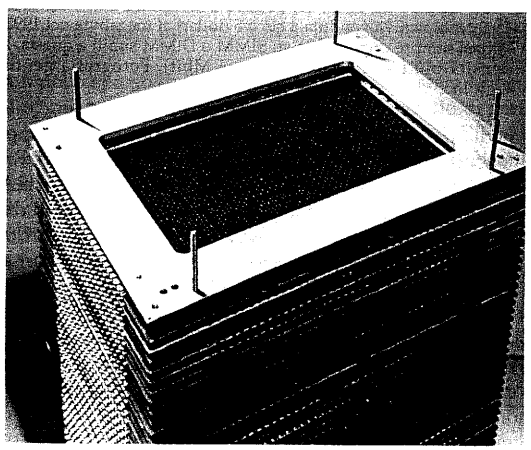


Figura 7.37 Un piano di memoria a nuclei del Whirlwind contenente 256 nuclei sistemati su una griglia 16x16. La memoria a nuclei fu inventata nell'ambito del sistema Whirlwind, usato per problemi di difesa aerea, ed è ora esposta allo Smithsonian Museum (incidentalmente, Ken Olsen, il fondatore e presidente della Digital per 20 anni, costruì la macchina che collaudò queste memorie; fu il suo primo calcolatore). Fotografia gentilmente concessa da Computer Museum, Boston.

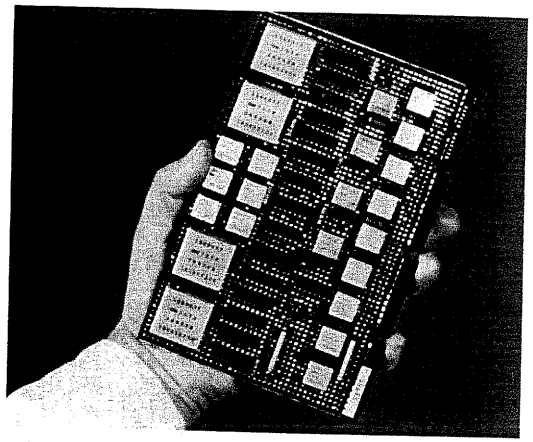


Figura 7.38 Una delle prime schede di DRAM. La scheda usa dei chip di 18 Kbit. Fotografia gentilmente concessa da IBM.

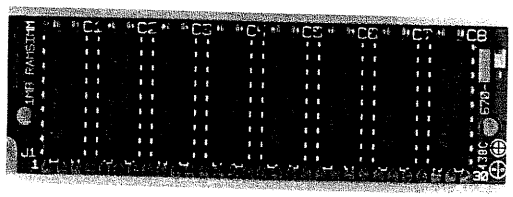


Figura 7.39 Una SIMM di 1 MB, costruita del 1986 con chip di 1 Mbit. Questa SIMM, usata in un Macintosh, veniva venduta a circa 5 dollari al megabyte nel 1997. Nel 1997 la memoria principale è quasi sempre organizzata su SIMM o DIMM simili a questa, anche se con chip di memoria di densità più elevata (16 Mbit o 64 Mbit). Fotografia gentilmente concessa da MIPS Technology, Inc.

pale mediante transistor rimase proibitiva fino allo sviluppo del circuito integrato. Con i circuiti integrati divenne possibile costruire una DRAM (memoria dinamica ad accesso casuale, si veda l'Appendice B): le prime DRAM furono costruite da Intel nel 1970 e poco dopo nacquero i primi sistemi dotati di memorie DRAM (come alternativa veloce ai nuclei); si usava 1 Kbit di DRAM. Alcune indiscrezioni riferiscono infatti che Intel avesse sviluppato i microprocessori almeno in parte per riuscire a vendere più DRAM. La figura 7.38 mostra una delle prime schede di DRAM. Nei tardi anni '70 la memoria a nuclei era già una curiosità storica; analogamente alla notevole espansione in termini di capacità di memoria portata dalla tecnologia delle memorie a nuclei, la tecnologia delle DRAM permise un'espansione di analoga portata. Negli anni '90 molti personal computer disponevano di una quantità di memoria pari a quella dei più grandi sistemi dotati di memoria a nuclei.

Oggigiorno le DRAM sono normalmente organizzate come una serie di chip su una piccola scheda detta SIMM (single inline memory module) o DIMM (dual inline memory module); la SIMM mostrata in figura 7.39 contiene 1 MB in totale e veniva venduta per circa 5 dollari nel 1997; le SIMM e DIMM più grandi disponibili nel 1997 erano di 64 MB. Anche se le DRAM rimarranno la tecnologia dominante per la memoria per qualche anno ancora, sono in corso di sviluppo delle notevoli

innovazioni sui package delle DRAM per fornire maggiori larghezza di banda e densità.

Lo sviluppo delle gerarchie di memorie

Già i pionieri dell'informatica previdero la necessità di una gerarchia di memorie e ne coniarono il termine, ma la gestione automatica di due livelli fu proposta per la prima volta da Kilburn e dai suoi colleghi e fu dimostrata alla University of Manchester sul calcolatore Atlas, che implementava la memoria virtuale; questo succedeva un anno *prima* che fosse annunciato l'IBM 360. La IBM pianificò di includere la memoria virtuale nella generazione successiva (System/370), ma il sistema operativo OS/360 nel 1970 non era in grado di supportarla; la memoria virtuale per la famiglia 370 fu annunciata nel 1972, e fu per questo sistema che fu coniato il termine *translation-lookaside buffer* (TLB). Gli unici elaboratori odierni privi di memoria virtuale sono i supercomputer, e vi sono buone probabilità che la includano nel prossimo futuro.

I problemi legati allo spazio di indirizzamento inadeguato afflissero ripetutamente i progettisti: gli architetti del PDP-11 identificarono nello spazio di indirizzamento ridotto l'unico errore architetturale difficile da correggere. Quando fu progettato il PDP-11 la densità delle memorie a nuclei stava crescendo molto lentamente e la concorrenza di altre 100 aziende produttrici di minicomputer significava che la DEC correva il rischio di non avere un prodotto competitivo sul piano dei costi se ogni indirizzo avesse dovuto attraversare due volte l'unità di elaborazione a 16 bit; da qui nacque la decisione di aggiungere solo 4 bit di indirizzo rispetto al predecessore del PDP-11. Gli architetti dell'IBM 360 erano consci dell'importanza dello spazio di indirizzamento e pianificarono un'architettura estendibile fino a 32 bit di indirizzi; nell'IBM 360 si usavano solo 24 bit, poiché i modelli di fascia bassa della serie 360 sarebbero stati troppo rallentati da indirizzi più ampi. Sfortunatamente l'estensione a 32 bit fu resa più complessa dal fatto che i programmatori presero a memorizzare delle informazioni aggiuntive negli 8 bit più significativi "inutilizzati".

L'esaurimento dello spazio di indirizzamento è spesso stata la causa della morte delle architetture, mentre altre architetture sono state in grado di gestire una transizione verso spazi di indirizzamento più ampi. Per esempio il PDP-11, una macchina a 16 bit, fu sostituita dal VAX a 32 bit. L'80386 estese l'architettura dell'80286 da 16 a 32 bit. Diversi set di istruzioni RISC hanno recentemente completato la transizione dall'indirizzamento a 32 bit a quello a 64 bit per mezzo di un'estensione compatibile del set di istruzioni stesso. È stato inoltre annunciato un piano di Intel per sviluppare un nuovo set di istruzioni in collaborazione con HP; il nuovo set di istruzioni avrà uno spazio di indirizzamento di 64 bit.

Molte delle prime idee sulle gerarchie di memorie furono sviluppate in Inghilterra; pochi anni dopo l'articolo sul sistema Atlas, Wilkes [1965] pubblicò il primo articolo che descriveva il concetto di cache, denominandola "slave" (schiavo):

> *Viene discusso l'uso di una memoria a nuclei veloce, diciamo di 32.000 parole, come slave di una memoria a nuclei più lenta, diciamo di un milione di parole, in modo tale che in pratica il tempo di accesso effettivo sia più vicino a quello della memoria veloce che a quello della memoria lenta.*

Questo articolo di due pagine descrive una cache a corrispondenza diretta. Anche se questa è la prima pubblicazione sulle cache, la prima implementazione fu

probabilmente una cache a corrispondenza diretta costruita alla University of Cambridge da Scarron e descritta al congresso IFIP nel 1965: era basata su una memoria a diodi ad effetto tunnel, la forma di memoria più veloce disponibile in quel periodo.

In seguito a quella pubblicazione la IBM attivò un progetto che portò al primo sistema commerciale dotato di cache, l'IBM 360/85. Alla IBM Gibson riconobbe che il comportamento degli accessi alla memoria poteva avere un impatto notevole sulle prestazioni: descrisse come misurare il comportamento del programma ed il comportamento della cache e mostrò che la frequenza di miss variava in programmi diversi. Per mezzo di un campione di 20 programmi (con 3 milioni di riferimenti ciascuno, un numero incredibile per quei tempi), Gibson analizzò l'efficacia delle cache usando come metrica il tempo medio di accesso alla memoria. Conti, Gibson e Pitkowski descrissero le conseguenti prestazioni del 360/85 nel primo articolo in cui fu usato il termine *cache* nel 1968. A partire da questo primo lavoro è divenuto chiaro che le cache sono una delle idee più importanti non solo per le architetture dei sistemi di elaborazione, ma anche nei sistemi software; l'idea di cache ha trovato applicazione nei sistemi operativi, nelle reti di calcolatori, nelle basi di dati e nei compilatori, per citarne alcuni. Vi sono migliaia di articoli sull'argomento delle cache, ed esse continuano a costituire un'area di ricerca importante.

Meccanismi di protezione

Il supporto delle architetture ai meccanismi di protezione è variato notevolmente nel corso degli scorsi 20 anni. Nei primi calcolatori, prima dell'avvento della memoria virtuale, la protezione era nel migliore dei casi molto semplificata; negli anni '70 si inventarono dei meccanismi più elaborati che supportavano diversi livelli di protezione (detti *anelli*, in inglese *ring*). Nei tardi anni '70 e negli anni '80 si pensarono prima ed implementarono poi meccanismi di protezione molto elaborati: essi supportavano una grande varietà di potenti modelli di protezione che permettevano versioni controllate della condivisione, in cui un processo poteva condividere dei dati ed allo stesso tempo controllare cosa altri processi potessero fare su tali dati. Il metodo più potente, che faceva uso delle cosiddette *capability*, prevedeva di creare un oggetto che descriveva i diritti di accesso ad una parte della memoria; le diverse capability potevano essere poi passate ad altri processi, fornendo quindi l'accesso all'oggetto descritto dalla capability. Il supporto di questo meccanismo di protezione sofisticato si rivelò sia complesso che costoso, poiché la creazione, la copia e la manipolazione delle capability richiedeva la collaborazione del sistema operativo e di un adeguato supporto hardware. Gli elaboratori più recenti supportano un modello di protezione più semplice basato sulla memoria virtuale, simile a quello discusso nel Paragrafo 7.4.

Letture consigliate

Conti, C., D. H. Gibson e S. H. Pitowski [1968]. "Structural aspects of the System/360 Model 85, part I: General organization", *IBM Systems J.* 7:1. 2-14.
Presenta il primo calcolatore commerciale a far uso di una cache e ne descrive le prestazioni.

Hennessy, J. e D. Patterson [1996]. Capitolo 5 di "*Computer Architecture: A Quantitative Approach*", Second edition, Morgan Kauffmann Publishers, San Francisco.
Fornisce un maggiore approfondimento di diversi argomenti tra cui la protezione, il miglioramento delle prestazioni per le operazioni di scrittura, le cache indirizzate virtualmente, le cache a più livelli, i meccanismi aggiuntivi per la tolleranza della latenza e la coerenza delle cache.

Kilburn, T., D. B. Edwards, M. J. Lanigan e F. H. Sumner [1962]. "One-level storage system", *IRE Transactions on Electronic* Computers EC-11 (Aprile) 223-35. Apparso anche in D. P. Siewiorek, C. G. Bell e A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 135-48, 1982.
Questo articolo è un classico: la prima proposta della memoria virtuale.

Przybylski, S. A. [1990]. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, Morgan Kauffmann Publishers, San Francisco.
Un'esplorazione approfondita delle gerarchie di memorie a più livelli e delle loro prestazioni.

Silberschatz, A. e P. Galvin [1994]. *Operating Systems Concepts*, Addison-Wesley, Reading, MA.
Un libro di testo sui sistemi operativi con una discussione approfondita della memoria virtuale, dei processi, della gestione dei processi e degli aspetti legati alla protezione.

Smith, A. J. [1982]. "Cache memories", *Computing Surveys* 14:3 (Settembre) 473-530.
L'articolo classico di introduzione alle cache. Questo articolo ha definito la terminologia di quest'area ed è servito come punto di riferimento per molti progettisti di calcolatori.

Tanenbaum, A. [1991]. *Operating Systems Principles*, Addison-Wesley, Reading, MA.
Un libro di testo sui sistemi operativi con una buona trattazione della memoria virtuale.

Wilkes, M. [1965]. "Slave memories and dynamic storage allocation", *IEEE Trans. Electronic Computers* EC-14:2 (Aprile) 270-71.
Il primo, classico, articolo sulle cache.

7.10 Parole chiave

I progettisti hanno sviluppato una vasta gamma di strategie che sfruttano il principio di località per superare il divario tra il processore e la memoria principale, portando ad un'ampia terminologia per descrivere tali soluzioni.

abilitazione delle interruzioni
aliasing
bit di riferimento (o di uso)
bit di validità
blocco
buffer delle scritture
cache a corrispondenza diretta
cache a più livelli
cache completamente associativa
cache divisa
cache indirizzata fisicamente
cache indirizzata virtualmente
cache non bloccante
cache set-associative
cambio di contesto
chiamata di sistema
frequenza di hit
frequenza di miss
frequenza di miss globale
frequenza di miss locale
gerarchia delle memorie
hit nella cache (o successo nell'accesso alla cache)
indirizzo fisico
indirizzo virtuale
istruzione rieseguibile
località spaziale
località temporale
memoria virtuale
miss certo o miss di partenza a freddo
miss nella cache (o fallimento nell'accesso alla cache)
miss per capacità
miss per conflitti o per collisioni
modello delle tre C
modo kernel o supervisore
page fault
penalità di miss
pre-caricamento
protezione
segmentazione
tabella delle pagine
tag
tempo di hit
traduzione (o corrispondenza) degli indirizzi
translation-lookaside buffer (TLB)
usato meno di recente (LRU)
write-back
write-through

7.11 Esercizi

7.1 [10] <§7.2> Si descrivano le caratteristiche generali di un programma avente località spaziale e temporale molto basse negli accessi ai dati. Si fornisca un programma di esempio (anche sotto forma di pseudo-codice).

7.2 [10] <§7.2> Si descrivano le caratteristiche generali di un programma avente una forte località temporale ma una limitata località spaziale negli accessi ai dati. Si fornisca un programma di esempio (anche sotto forma di pseudo-codice).

7.3 [10] <§7.2> Si descrivano le caratteristiche generali di un programma avente una limitata località temporale ma una forte località spaziale negli accessi ai dati. Si fornisca un programma di esempio (anche sotto forma di pseudo-codice).

7.4 [10] <§7.2> Si descrivano le caratteristiche generali di un programma avente località spaziale e temporale molto basse negli accessi alle istruzioni. Si fornisca un programma di esempio (anche sotto forma di pseudo-codice).

7.5 [10] <§7.2> Si descrivano le caratteristiche generali di un programma avente una forte località temporale ma una limitata località spaziale negli accessi alle istruzioni. Si fornisca un programma di esempio (anche sotto forma di pseudo-codice).

7.6 [10] <§7.2> Si descrivano le caratteristiche generali di un programma avente una limitata località temporale ma una forte località spaziale negli accessi alle istruzioni. Si fornisca un programma di esempio (anche sotto forma di pseudo-codice).

7.7 [10] <§7.2> Sia data la seguente sequenza di indirizzi a cui si intende fare accesso, espressi come indirizzi di parola: 1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17. Disponendo di una cache a corrispondenza diretta con blocchi di una parola di 16 bit, inizialmente vuota, si determini se ciascuno degli accessi sia un hit o un miss e si mostri il contenuto finale della cache.

7.8 [10] <§7.2> Usando la stessa sequenza di accessi dell'Esercizio 7.7 si identifichino gli hit ed i miss e si calcoli il contenuto finale della cache nel caso di una cache a corrispondenza diretta con blocchi di quattro parole ed una *dimensione totale* di 16 parole.

7.9 [10] <§7.2> Si calcoli il numero totale di bit necessari ad implementare la cache di figura 7.10. Il numero ottenuto è diverso dalla dimensione della cache, con la quale si intende normalmente il numero di byte di dati in essa memorizzati. Il numero di bit necessari ad implementare la cache rappresenta la quantità complessiva di memoria per poter memorizzare tutti i dati, i campi tag ed i bit di validità.

7.10 [10] <§7.2> Si trovi una tecnica in grado di eliminare la porta AND sul bit di validità mostrata in figura 7.7. Suggerimento: occorre lavorare sulle operazioni di confronto.

7.11 [10] <§7.2> Si considerino le gerarchie delle memorie basate sulle tre organizzazioni della memoria principale riportate in figura 7.13. Si ipotizzi che la dimensione dei blocchi della cache sia pari a 16 parole, che la larghezza della memoria nello schema b della figura sia di quattro parole e che il numero di banchi dello schema c sia pari a quattro. Se la latenza di un accesso alla memoria principale è di 10 cicli ed il tempo di trasferimento è di un ciclo, quali sono le penalità di miss nei tre casi?

7.12 [10] <§7.2> {Es. 7.11} Si supponga che un processore abbia una frequenza di miss effettiva pari a 0,5% miss per istruzione usando blocchi di 16 parole, e che il valore di CPI in assenza di miss nella cache sia pari a 1,2. Usando le memorie descritte nella figura 7.13 e nell'Esercizio 7.11, di quanto sarebbe più veloce il processore se fosse collegato alla memoria di maggior larghezza invece che alla memoria più stretta o con interleaving?

7.13 [15] <§7.2> La cache C1 è del tipo a corrispondenza diretta con 16 blocchi di una parola ciascuno; la cache C2 è del tipo a corrispondenza diretta con 4 blocchi di quattro parole ciascuno. Si supponga che la penalità di miss sia di 8 cicli di clock per C1 e di 11 cicli per C2. Ipotizzando che le cache siano inizialmente vuote, si trovi una sequenza di accessi per cui C2 abbia una frequenza di miss inferiore ma richieda più cicli di clock dovuti ai miss rispetto a C1. Si esprimano gli indirizzi come indirizzi di parola.

7.14 [15] <§7.2> Per le cache dell'Esercizio 7.13 si trovi una sequenza di accessi per cui C2 abbia più miss di C1. Si esprimano gli indirizzi come indirizzi di parola.

In maggior dettaglio

Tempo medio di accesso alla memoria

Per tenere conto del fatto che le prestazioni sono influenzate dal tempo necessario ad accedere ai dati sia nel caso di miss che nel caso di hit, i progettisti usano spesso il tempo medio di accesso alla memoria (average memory access time, AMAT) come metrica alternativa per valutare i progetti delle cache. Il tempo medio di accesso alla memoria è calcolato tenendo conto sia degli hit che dei miss, considerando anche la diversa frequenza degli accessi di tipo diverso; il suo valore è pari a

AMAT = tempo necessario per un hit + frequenza di miss × penalità di miss

Il parametro AMAT si rivela un'utile cifra di merito per diversi sistemi basati su cache.

7.15 [5] <§7.2> Si trovi il valore di AMAT per un calcolatore con periodo di clock di 2 ns, penalità di miss di 20 cicli di clock, frequenza di miss pari a 0,05 miss per istruzione e tempo di accesso alla cache (inclusa la rilevazione dell'eventuale hit) di 1 ciclo di clock. Si assuma che le penalità di miss per le operazioni di lettura e di scrittura siano uguali e si ignorino gli altri stalli dovuti alle operazioni di scrittura.

7.16 [5] <§7.2> {Es. 7.15} Si ipotizzi di poter migliorare la frequenza di miss fino ad un valore pari a 0,03 miss per istruzione raddoppiando la dimensione della cache. Questa modifica fa sì che il tempo di accesso alla cache aumenti fino a 1,2 cicli di clock. Utilizzando il valore di AMAT come metrica si determini se la modifica porti a risultati positivi.

7.17 [10] <§7.2> {Es. 7.16} Se il tempo di accesso alla cache determinasse la frequenza di clock del processore, come spesso avviene, il valore di AMAT potrebbe non indicare correttamente le prestazioni di diverse organizzazioni della cache. Se il clock di un processore deve essere modificato per poter corrispondere alla velocità della cache, si è di fronte ad un buon compromesso? Si ipotizzi che due calcolatori siano identici eccezion fatta per la frequenza di clock e per il numero di cicli per i miss nella cache; si ipotizzi che vi siano 1,5 accessi in memoria per istruzione ed un valore di CPI in assenza di miss pari a 2. La penalità di miss è di 20 cicli in entrambi i calcolatori.

7.18 [10] <§§7.2, B.5> Un progettista ha a disposizione 18 SRAM di 32K × 8 bit ciascuna per costruire una cache per un processore con indirizzi di 32 bit. Qual è la cache a corrispondenza diretta di maggiori dimensioni (ossia quella che permette di memorizzare il maggior numero di byte di dati) che si può costruire usando blocchi di una parola (32 bit)? Si mostri la suddivisione degli indirizzi nelle diverse componenti relative all'accesso alla cache (in modo simile alla figura 7.8) e si descriva come saranno usati i chip di SRAM. Suggerimento: potrebbe non essere necessario usarli tutti.

7.19 [10] <§§7.2, B.5> Si svolga un esercizio simile al 7.18, in cui si voglia costruire una cache a corrispondenza diretta con blocchi di quattro parole come quella di figura 7.10. Ancora una volta si mostri la suddivisione degli indirizzi nelle diverse componenti e si descriva come saranno usati i chip.

7.20 [10] <§7.3> Usando la stessa sequenza di accessi dell'Esercizio 7.7 si identifichino gli hit ed i miss e si calcoli il contenuto finale della cache nel caso di una cache set-associative a due vie con blocchi di una parola ed una *dimensione totale* di 16 parole. Si ipotizzi una sostituzione di tipo LRU.

7.21 [10] <§7.3> Usando la stessa sequenza di accessi dell'Esercizio 7.7 si identifichino gli hit ed i miss e si calcoli il contenuto finale della cache nel caso di una cache completamente associativa con blocchi di una parola ed una *dimensione totale* di 16 parole. Si ipotizzi una sostituzione di tipo LRU.

7.22 [10] <§7.3> Usando la stessa sequenza di accessi dell'Esercizio 7.7 si identifichino gli hit ed i miss e si calcoli il contenuto finale della cache nel caso di una cache completamente associativa con blocchi di quattro parole ed una *dimensione totale* di 16 parole. Si ipotizzi una sostituzione di tipo LRU.

7.23 [5] <§7.3> L'associatività tende in generale a migliorare la frequenza di miss, ma non sempre ciò avviene. Si fornisca una breve sequenza di indirizzi per i quali una cache set-associative a due vie con sostituzione LRU soffra di un maggior numero di miss rispetto ad una cache a corrispondenza diretta di pari dimensioni.

7.24 [15] <§7.3> Si ipotizzi che la dimensione degli indirizzi di un calcolatore sia pari a k bit (con un indirizzamento a byte), che la dimensione della cache sia di S byte, i blocchi della cache siano di B byte e la cache sia di tipo set-associative ad A vie. Si ipotizzi che B sia una potenza di due, per cui $B=2^b$. Si determinino le seguenti quantità in funzione di S, B, A, b e k: il numero di insiemi nella cache, il numero di bit di indice nell'indirizzo, il numero di bit necessari ad implementare la cache (si veda anche l'Esercizio 7.9).

7.25 [10] <§7.3> Questo esercizio tratta di cache con dimensioni insolite. È possibile costruire una cache completamente associativa che contenga esattamente 3K parole di dato? E una cache di tipo set-associative, o a corrispondenza diretta delle stesse dimensioni? In ciascuno dei casi si motivi la risposta. Si ricordi che $1K=2^{10}$.

7.26 [10] <§7.3> Questo esercizio è simile al 7.25, sostituendo il valore 3K con 300. Si ricordi che $300 = 3 * 10^2$.

7.27 [20] <§7.3> Si considerino tre calcolatori con diverse configurazioni della cache:

- *Cache 1*: a corrispondenza diretta con blocchi di una parola
- *Cache 2*: a corrispondenza diretta con blocchi di quattro parole

- *Cache 3*: set-associative a due vie con blocchi di quattro parole.

Sono state fatte le seguenti misure della frequenza di miss:

- *Cache 1*: frequenza di miss per le istruzioni: 4%; per i dati: 8%
- *Cache 2*: frequenza di miss per le istruzioni: 2%; per i dati: 5%
- *Cache 3*: frequenza di miss per le istruzioni: 2%; per i dati: 4%.

In questi calcolatori metà delle istruzioni contengono un accesso ai dati. Si ipotizzi che la penalità di miss introdotta dalla cache sia pari a 6 + dimensione del blocco (in parole). Il valore di CPI per questo carico di lavoro, misurato sul calcolatore con la cache 1, è risultato pari a 2,0. Si determini quale dei calcolatori perde più tempo per i miss nella cache.

7.28 [5] <§7.3> {Es. 7.27} I periodi di clock per i calcolatori dell'Esercizio 7.27 sono di 2 ns per i primi due casi e di 2,4 ns per il terzo. Si determini quale calcolatore sia più veloce e quale più lento.

7.29 [10] <§§7.2, 7.3> Il seguente programma in C viene eseguito (senza alcuna ottimizzazione) su un calcolatore con una cache contenente 256 byte di dati e dotata di blocchi di quattro parole (16 byte):

```
int i, j, c, passo, matrice[256] ;
. . .
for (i=0; i<10000; i++)
  for (j=0; j<256; j=j+passo)
   c = matrice[j]+5;
```

Considerando la sola attività sulla cache generata dall'accesso agli elementi della matrice ed ipotizzando che le variabili intere corrispondano a singole parole, quale frequenza di miss ci si deve aspettare quando la cache è a corrispondenza diretta e passo=132? E se passo=131? I risultati cambierebbero se la cache fosse di tipo set-associative a due vie?

7.30 [10] <§§7.3, B.5> Si risolva un esercizio simile al 7.18, con la differenza che in questo caso si vuole costruire una cache set-associative a tre vie con blocchi di una parola. Si mostri ancora una volta la suddivisione degli indirizzi nelle diverse componenti (si veda la figura 7.19 per un esempio relativo ad una cache set-associative a quattro vie) e si descriva come saranno usati i chip. Si noti che ciascuna SRAM può effettuare una sola operazione di lettura per ciascun accesso alla cache.

7.31 [5] <§§7.2-7.4> Si ordinino le combinazioni di eventi della figura 7.27 secondo la rispettiva frequenza.

7.32 [15] <§7.4> Si consideri un sistema di memoria virtuale con le seguenti caratteristiche:

- indirizzi virtuali di 40 byte
- pagine di 16 KB
- indirizzi fisici di 26 bit.

Qual è la dimensione complessiva della tabella delle pagine per ciascun processo di questo calcolatore, ipotizzando che le informazioni di validità, di modifica, di protezione e di uso richiedano 4 bit e che siano utilizzate tutte le pagine virtuali? Si assuma che gli indirizzi su disco non vengano memorizzati nella tabella delle pagine.

7.33 [15] <§7.4> Si ipotizzi che il sistema di memoria virtuale dell'Esercizio 7.32 sia implementato con un TLB set-associative a due vie di 256 elementi. Si mostri la traduzione di indirizzi virtuali in indirizzi fisici con una figura simile alla 7.25. Si specifichi l'ampiezza di tutti i campi e di tutti i segnali.

7.34 [15] <§7.3> Si ipotizzi che la cache del calcolatore descritto nell'Esercizio 7.32 sia di tipo set-associative a due vie ed abbia blocchi di otto parole ed una dimensione totale di 16 KB. Si mostri l'organizzazione della cache e la modalità di accesso nello stesso formato della figura 7.19.

7.35 [15] <§7.4> Le tabelle delle pagine richiedono notevoli quantità di memoria (come descritto in un approfondimento del Paragrafo 7.4), anche se spesso la maggior parte degli elementi sono inutilizzati. Il numero di pagina virtuale, illustrato nella figura 7.21, può essere suddiviso in due parti, un "numero di tabella delle pagine" ed un "offset interno alla tabella delle pagine". Il numero di tabella delle pagine può essere usato per indicizzare una tabella delle pagine di primo livello che fornisca l'indirizzo fisico di una seconda tabella delle pagine, se questa risiede in memoria (altrimenti si verificherà un page fault di primo livello ed occorrerà prelevare la tabella delle pagine stessa dal disco). L'offset interno alla tabella delle pagine verrà quindi usato come indice nella tabella delle pagine di secondo livello per determinare il numero di pagina fisica. Un modo naturale per definire questo schema è fare in modo che ciascuna tabella delle pagine di secondo livello occupi esattamente una pagina di memoria. Ipotizzando uno spazio di indirizzamento virtuale di 32 bit con pagine di 4 KB e 4 byte per ciascun elemento della tabella delle pagine, quanti byte sono necessari per memorizzare la tabella delle pagine di primo livello (che deve sempre risiedere in memoria) di un programma? Si forniscano delle figure simili alle figure 7.20, 7.21 e 7.22 per dimostrare la corretta comprensione della tecnica.

7.36 [15] <§7.4> Ipotizzando di usare la tabella delle pagine gerarchica descritta nell'Esercizio 7.35, che in memoria vi sia esattamente una tabella delle pagine di secondo livello e che metà dei suoi elementi siano validi, quanti byte di memoria dello spazio di indirizzamento virtuale risiedono effettivamente nella memoria fisica? Suggerimento: la tabella delle pagine di secondo livello occupa esattamente una pagina di memoria fisica.

7.37 [10] <§7.4> Alcuni programmi, come le simulazioni complesse di dati meteorologici, vengono caricati su un calcolatore su cui lavorano ininterrottamente per lunghi periodi di tempo; spesso per queste applicazioni si acquistano dei costosi supercomputer. Si discutano alcuni motivi per cui la memoria virtuale può essere desiderabile o meno nei sistemi progettati per questo tipo di applicazioni. Sarà necessaria una cache?

7.38 [5] <§7.5> Se si classificano tutti i miss secondo una delle tre categorie (Certi, Capacità, Conflitti), quali miss si ridurranno con maggior probabilità riscrivendo il programma in modo da usare meno memoria? Che cosa succede se si aumenta la frequenza di clock del processore che sta eseguendo il programma? E se si aumenta il grado di associatività della cache esistente?

7.39 [5] <§7.5> Il seguente programma C può servire come spunto per scrivere un simulatore di cache. Molti tipi di dato non sono stati definiti, ma il codice descrive accuratamente le azioni che si svolgono durante un accesso in lettura ad una cache a corrispondenza diretta.


```
parola LeggiCacheCorrispondenzaDiretta(Indirizzo i)
{
  static Elemento cache[DIMENSIONE_CACHE_IN_PAROLE];
  Elemento e = cache[i.indice];
  if (e.valido == FALSO || e.tag != i.tag) {
    e.valido = VERO;
    e.tag = i.tag;
    e.dato = carica_da_memoria(i);
  }
  return e.dato;
}
```

L'esercizio consiste nel modificare il codice in modo da descrivere accuratamente le azioni che si svolgono in un accesso in lettura ad una cache a corrispondenza diretta con blocchi di più parole.

7.40 [8] <§7.5> Si svolga un esercizio simile al 7.39, scrivendo questa volta il codice per l'accesso in lettura ad una cache set-associative ad *n* vie con blocchi di una parola. Si noti che il codice potrebbe far sembrare che i confronti abbiano natura sequenziale, quando in realtà sono calcolati in parallelo dall'hardware.

7.41 [8] <§7.5> {Es. 7.39} Si estenda la soluzione data all'Esercizio 7.39 includendo la specifica di una nuova procedura per la gestione degli accessi in scrittura, assumendo una metodologia di tipo write-through. Verificare se la soluzione data per la gestione degli accessi in lettura deve essere modificata.

7.42 [8] <§7.5> {Es. 7.39} Si estenda la soluzione data all'Esercizio 7.39 includendo la specifica di una nuova procedura per la gestione degli accessi in scrittura, assumendo una metodologia di tipo write-back. Verificare se la soluzione data per la gestione degli accessi in lettura deve essere modificata.

7.43 [8] <§7.5> {Es. 7.40} Si svolga un esercizio simile al 7.41, estendendo però la soluzione data all'Esercizio 7.40. Si ipotizzi che la cache usi la sostituzione casuale.

7.44 [8] <§7.5> {Es. 7.40} Si svolga un esercizio simile al 7.42, estendendo però la soluzione data all'Esercizio 7.40. Si ipotizzi che la cache usi la sostituzione casuale.

7.45 [5] <§§7.7-7.8> Perché un compilatore dovrebbe compiere la seguente ottimizzazione?

```
/* Prima */
for (j=0; j<20; j++)
  for (i=0; i<200; i++)
    x[i][j] = x[i][j] + 1;
/* Dopo */
for (i=0; i<200; i++)
  for (j=0; j<20; j++)
    x[i][j] = x[i][j] + 1;
```

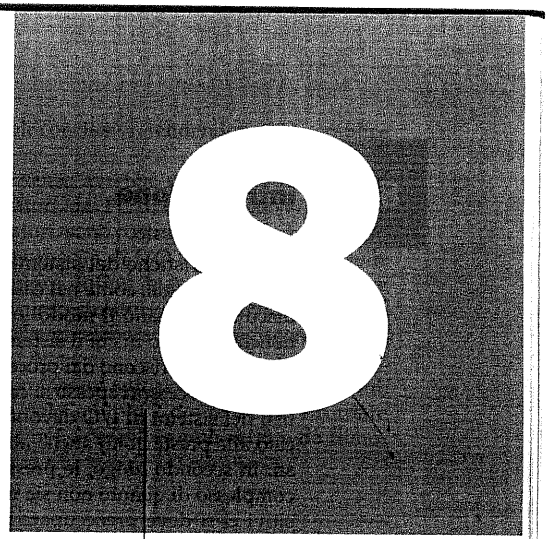
7.46 [3 ore] <§7.2> Si usi un simulatore di cache per simulare diverse organizzazioni della cache per il primo milione di accessi in memoria durante un'esecuzione del programma gcc. È possibile reperire *dinero* (un simulatore di cache) ed i file contenenti alcune tracce di esecuzione di gcc come illustrato nella prefazione del libro. Si ipotizzi di disporre di una cache di 32 KB per le istruzioni ed una cache di

32 KB per i dati aventi la stessa organizzazione. Si devono sperimentare almeno due diversi gradi di associatività e due dimensioni dei blocchi. Si disegni uno schema come quello di figura 7.19 che descriva l'organizzazione della cache per i dati che ha fornito la miglior frequenza di hit.

7.47 [4 ore] <§§7.2-7.4> Si vuole usare un simulatore di cache per simulare diverse organizzazioni del TLB e della memoria virtuale; si usi a questo scopo il primo milione di accessi di gcc. Si vuole conoscere la frequenza di miss del TLB per le combinazioni di TLB e di pagine seguenti:

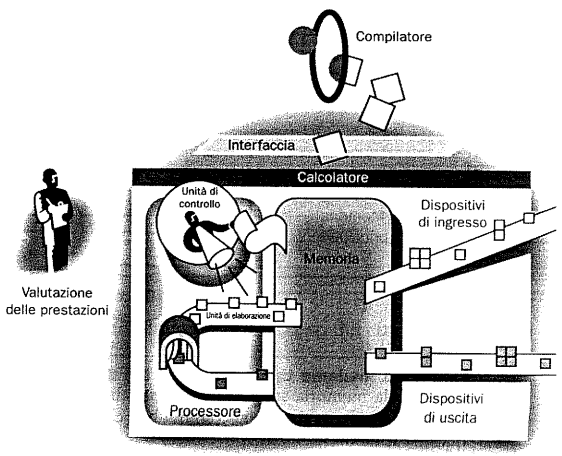
1. TLB con 64 elementi, completamente associativo e con pagine di 4 KB
2. TLB con 32 elementi, completamente associativo e con pagine di 8 KB
3. TLB con 64 elementi, set-associative ad otto vie e con pagine di 4 KB
4. TLB con 128 elementi, set-associative a quattro vie e con pagine di 4 KB.

7.48 [1 giorno] <§7.2> È stato commissionato il progetto di una cache per un nuovo calcolatore. Esso ha indirizzi fisici di 32 bit e richiede cache separate per istruzioni e dati. Le SRAM hanno tempo di accesso di 1,5 ns e dimensioni pari a 32K × 8 bit; si possono usare fino a 16 SRAM. La penalità di miss per il sottosistema di memoria è pari a 8 + 2 × dimensione dei blocchi in parole. L'uso della tecnica set-associative comporta un tempo aggiuntivo di accesso alla cache di 0,2 ns. Usando il primo milione di accessi di gcc si trovino le migliori organizzazioni delle cache istruzioni e dati, date le SRAM disponibili.



L'interfaccia tra processori e periferiche

Le cinque componenti classiche del calcolatore



L'I/O è sicuramente rimasto indietro nell'ultimo decennio.

Seymour Cray
Lezione pubblica, 1976

32 KB per i dati aventi la stessa organizzazione. Si devono sperimentare almeno due diversi gradi di associatività e due dimensioni dei blocchi. Si disegni uno schema come quello di figura 7.19 che descriva l'organizzazione della cache per i dati che ha fornito la miglior frequenza di hit.

7.47 [4 ore] <§§7.2-7.4> Si vuole usare un simulatore di cache per simulare diverse organizzazioni del TLB e della memoria virtuale; si usi a questo scopo il primo milione di accessi di gcc. Si vuole conoscere la frequenza di miss del TLB per le combinazioni di TLB e di pagine seguenti:

1. TLB con 64 elementi, completamente associativo e con pagine di 4 KB
2. TLB con 32 elementi, completamente associativo e con pagine di 8 KB
3. TLB con 64 elementi, set-associative ad otto vie e con pagine di 4 KB
4. TLB con 128 elementi, set-associative a quattro vie e con pagine di 4 KB.

7.48 [1 giorno] <§7.2> È stato commissionato il progetto di una cache per un nuovo calcolatore. Esso ha indirizzi fisici di 32 bit e richiede cache separate per istruzioni e dati. Le SRAM hanno tempo di accesso di 1,5 ns e dimensioni pari a 32K × 8 bit; si possono usare fino a 16 SRAM. La penalità di miss per il sottosistema di memoria è pari a 8 + 2 × dimensione dei blocchi in parole. L'uso della tecnica set-associative comporta un tempo aggiuntivo di accesso alla cache di 0,2 ns. Usando il primo milione di accessi di gcc si trovino le migliori organizzazioni delle cache istruzioni e dati, date le SRAM disponibili.

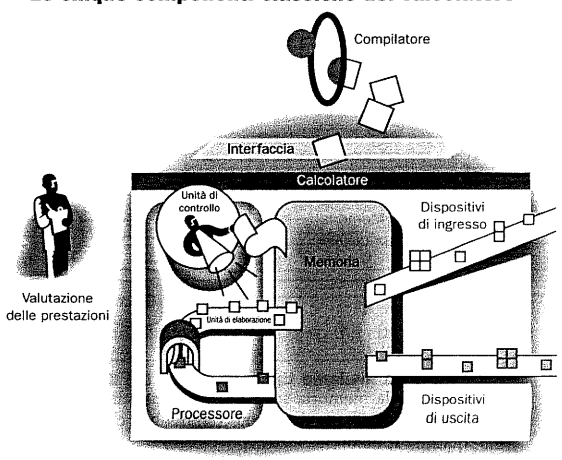
8

L'interfaccia tra processori e periferiche

L'I/O è sicuramente rimasto indietro nell'ultimo decennio.

Seymour Cray
Lezione pubblica, 1976

Le cinque componenti classiche del calcolatore



8.1 Introduzione

Le caratteristiche dei sistemi di ingresso/uscita (I/O) sono spesso determinate dalla tecnologia, come nel caso dei processori. Ad esempio, le proprietà delle unità disco influenzano il modo in cui i dischi possono essere connessi al processore, nonché il modo in cui il sistema operativo interagisce con i dischi. I sistemi di I/O, tuttavia, differiscono dai processori per alcuni punti fondamentali. Mentre i progettisti dei processori spesso si concentrano principalmente sulle prestazioni, i progettisti dei sistemi di I/O devono tenere in considerazione (con lo stesso rilievo dedicato alle prestazioni) anche altri aspetti, quali l'espandibilità e la tolleranza ai guasti. In secondo luogo, le prestazioni di un sistema di I/O sono un parametro più complesso di quanto non sia nel caso dei processori. Ad esempio, per alcuni dispositivi ci si deve preoccupare principalmente della latenza di accesso, mentre per altri è fondamentale il throughput. Inoltre, le prestazioni dipendono da molti aspetti del sistema: le caratteristiche del dispositivo, le modalità di connessione tra il dispositivo ed il resto del sistema, la gerarchia delle memorie, il sistema operativo. La figura 8.1 mostra la struttura di un sistema completo del suo I/O. Le varie componenti, dai singoli dispositivi di I/O, al processore ed al software di sistema, influenzeranno le prestazioni complessive del sistema di I/O.

Le difficoltà nell'identificare chiari schemi di valutazione e le criticità di progetto hanno spesso messo in secondo piano l'importanza architetturale dei sistemi di I/O. La ricerca si concentra sul progetto dei processori: le industrie presentano valutazioni delle prestazioni che utilizzano principalmente misure orientate ai processori; i corsi sui diversi aspetti dell'informatica, dalla programmazione all'architettura dei calcolatori, spesso ignorano l'I/O o gli dedicano uno spazio limitato; i

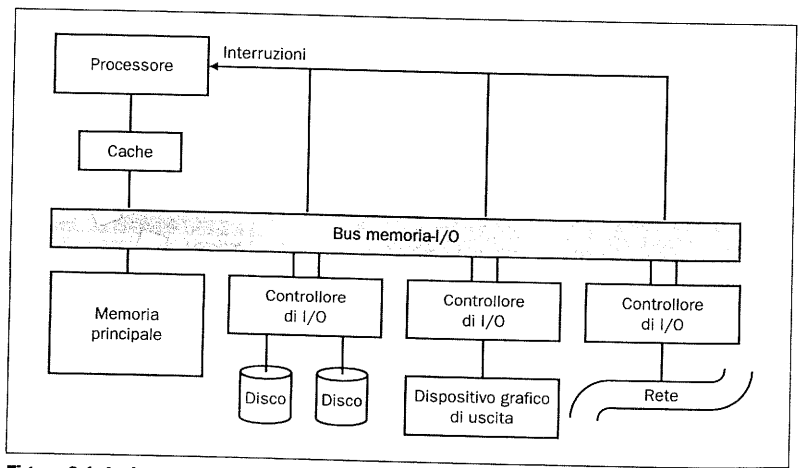


Figura 8.1 Insieme tipico di dispositivi di I/O. Le connessioni tra i dispositivi di I/O, il processore e la memoria sono normalmente chiamate bus. La comunicazione tra i dispositivi ed il processore utilizza sia i protocolli del bus che il meccanismo delle interruzioni, come si vedrà in questo capitolo.

libri di testo relegano l'argomento ai capitoli conclusivi, rendendone più semplice l'omissione a studenti ed insegnanti.

Questa situazione è priva di senso: che senso avrebbe un calcolatore privo di I/O? Inoltre, in un periodo in cui gli elaboratori, dai PC di basso livello ai mainframe più veloci, ed addirittura ai supercalcolatori, sono costruiti utilizzando la stessa tecnologia di base per i microprocessori, le capacità di I/O sono spesso una delle caratteristiche peculiari di ciascun elaboratore. Infine, mano a mano che cresce l'importanza delle connessioni in rete e delle infrastrutture informative, l'I/O gioca un ruolo sempre più importante: si ricordi che gli elaboratori interagiscono con gli utenti tramite l'I/O.

Se tutte queste argomentazioni ancora non fossero sufficientemente convincenti, la discussione della legge di Amdahl fatta nel Capitolo 2 dovrebbe far ricordare quanto sia pericoloso ignorare l'I/O. Un semplice esempio può servire da dimostrazione.

Impatto dell'I/O sulle prestazioni di un sistema

Esempio

Si supponga di avere un programma campione che viene eseguito in 100 secondi (in termini di tempo trascorso), dei quali 90 secondi sono costituiti da tempo di CPU ed il resto da tempo di I/O. Se il tempo di CPU si riduce del 50% all'anno per cinque anni di seguito, ma il tempo di I/O non si riduce, quanto più veloce sarà il programma alla fine dei cinque anni?

Soluzione

È noto che

$$\text{tempo trascorso} = \text{tempo di CPU} + \text{tempo di I/O}$$

$$100 = 90 + \text{tempo di I/O}$$

$$\text{tempo di I/O} = 10 \text{ secondi}$$

I nuovi tempi di CPU ed il conseguente tempo trascorso sono forniti nella seguente tabella:

Dopo *n* anni	Tempo di CPU	Tempo di I/O	Tempo trascorso	Tempo di I/O (in %)
0	90 secondi	10 secondi	100 secondi	10%
1	90/1,5 = 60 secondi	10 secondi	70 secondi	14%
2	60/1,5 = 40 secondi	10 secondi	50 secondi	20%
3	40/1,5 = 27 secondi	10 secondi	37 secondi	27%
4	27/1,5 = 18 secondi	10 secondi	28 secondi	36%
5	18/1,5 = 12 secondi	10 secondi	22 secondi	45%

Il miglioramento nelle prestazioni della CPU nel corso dei cinque anni è pari a

$$\frac{90}{12} = 7,5$$

Tuttavia il miglioramento nel tempo trascorso è soltanto

$$\frac{100}{22} = 4,5$$

ed il tempo di I/O è aumentato dal 10% al 45% del tempo trascorso.

Il modo di valutare le prestazioni dell'I/O spesso dipende dall'applicazione. In alcuni ambiti, ci si preoccupa principalmente del throughput del sistema; in questi casi l'ampiezza di banda di I/O sarà l'aspetto più importante. La stessa ampiezza di banda di I/O può essere misurata in due modi diversi:

1. Quanti dati possono essere trasferiti attraverso il sistema in un certo tempo?
2. Quante operazioni di I/O possono essere eseguite nell'unità di tempo?

Quale sia la misura più appropriata dipende dai casi; per esempio in molte applicazioni dei supercalcolatori gran parte delle richieste di I/O sono relative a lunghe sequenze di dati, e l'ampiezza di banda di trasferimento è un parametro importante. In altri casi si desidera invece eseguire un numero elevato di accessi piccoli e non correlati ad uno stesso dispositivo di I/O. Si può trovare un esempio di una tale situazione nell'ufficio per l'elaborazione dei dati fiscali del Ministero delle Finanze. Il Ministero delle Finanze mira principalmente ad elaborare il maggior numero di cartelle nell'unità di tempo; ciascuna cartella fiscale è memorizzata indipendentemente dalle altre ed è relativamente piccola. Un sistema orientato al trasferimento di file di grandi dimensioni può essere adeguato, ma un sistema di I/O in grado di supportare il trasferimento simultaneo di molti file di piccole dimensioni può essere più economico e più efficiente nell'elaborazione di milioni di cartelle fiscali.

In altre applicazioni ciò che importa è principalmente il tempo di risposta, che come si ricorderà è il tempo totale trascorso prima del completamento di una certa attività. Se le richieste di I/O sono estremamente pesanti, il tempo di risposta dipende in maniera determinante dall'ampiezza di banda, ma in molti casi la maggior parte degli accessi è di piccole dimensioni, ed il sistema di I/O con la minor latenza per accesso sarà quello in grado di fornire il minor tempo di risposta. Su elaboratori a singolo utente come le workstation ed i personal computer il tempo di risposta è la caratteristica chiave in termini di prestazioni.

Un elevato numero di applicazioni, specialmente nel vasto mercato dell'informatica, richiede sia alte prestazioni, sia bassi tempi di risposta. Ricadono ad esempio in questa categoria i Bancomat, i sistemi per la prenotazione dei voli, i sistemi di contabilità e di gestione dei magazzini, i file server e gli elaboratori che adottano il timesharing. Per tali applicazioni è importante sia la durata di ciascuna operazione, sia il numero di operazioni che possono essere elaborate in un secondo. Il numero di transazioni Bancomat contemporanee che possono essere elaborate in un'ora non sarebbe importante se ciascuna di esse richiedesse 15 minuti, dal momento che i clienti scapperebbero! Analogamente se si è in grado di elaborare rapidamente ciascuna transazione, ma se ne possono smaltire solo un piccolo numero nell'unità di tempo, allora non si potrà gestire un elevato numero di terminali Bancomat, oppure il costo del calcolatore richiesto per ciascun Bancomat sarà estremamente elevato.

Se davvero l'I/O è così importante, come si possono confrontare i sistemi di I/O? Si tratta di un problema complesso, in quanto le prestazioni dell'I/O dipendono da molti aspetti del sistema, ed applicazioni diverse sollecitano aspetti diversi del sistema di I/O. Inoltre ciascun progetto si basa su complesse valutazioni per

individuare il compromesso adeguato tra tempo di risposta e throughput, rendendo impossibile la valutazione di un singolo aspetto in maniera isolata. Ad esempio il tempo di risposta viene normalmente minimizzato se si gestisce ciascuna richiesta il prima possibile, mentre per massimizzare il throughput è possibile che convenga gestire insieme richieste tra loro connesse. Di conseguenza è possibile aumentare il throughput di un disco raggruppando le richieste che fanno accesso a locazioni tra loro prossime. Un tale approccio farà aumentare il tempo di risposta per alcune richieste, causando probabilmente una maggiore variabilità nel tempo di risposta. Benché il throughput sia maggiore, alcuni benchmark prevedono un tempo massimo di risposta per ciascuna richiesta, rendendo potenzialmente critiche tali ottimizzazioni.

Prima di discutere le diverse caratteristiche dei dispositivi di I/O, e come questi siano connessi tra loro, verranno brevemente riviste alcune misure di prestazioni per i sistemi di I/O.

8.2 La misura delle prestazioni dell'I/O: alcuni esempi su dischi e file system

La valutazione di un sistema di I/O deve tener conto di una vasta gamma di fattori. Le prestazioni sono uno di questi, ed in questo paragrafo verranno forniti alcuni esempi di metodi di misura utilizzabili per determinare le prestazioni di un sistema di dischi. Si considererà un insieme di problematiche tipiche dipendenti da molteplici elementi, quali la tecnologia utilizzata per i dischi, il modo in cui i dischi sono connessi, il sistema di memoria, il processore, ed il file system fornito dal sistema operativo. Nel complesso i metodi di valutazione dei sottosistemi di I/O restano alquanto primitivi se confrontati con la vasta attività registrata negli ultimi anni nel campo della valutazione dei processori. Questa situazione cambierà forse quando i progettisti capiranno l'importanza dell'I/O e l'inadeguatezza delle tecniche a disposizione per la sua valutazione.

Prima di discutere i casi campione considerati, è necessario affrontare un punto critico relativo alla terminologia e alle unità di misura. Le prestazioni dei sistemi di I/O dipendono dalla velocità con cui il sistema trasferisce i dati. La velocità di trasferimento dipende dalla frequenza del clock, che normalmente è espressa in MHz, ossia in milioni di cicli per secondo. La velocità di trasferimento è normalmente espressa in MB/s. Nei sistemi di I/O, i MB sono misurati usando la base 10 (quindi, 1 MB = 10^6 = 1000000 byte), mentre quando si parla di memoria principale si usa la base 2 (quindi 1 MB = 10^{20} = 1048576 byte). Inoltre, ad aumentare la confusione, tale differenza introduce la necessità di una conversione tra la base 10 (1K = 1000) e la base 2 (1K = 1024), dal momento che molti accessi di I/O sono relativi a blocchi di dati la cui dimensione è una potenza di 2. Anziché complicare tutti gli esempi convertendo le misure secondo uno dei due sistemi, si evidenzia ora questa distinzione ed il fatto che confondendo le due misure (come se le unità usate fossero identiche) si introduce un piccolo errore. Tale errore verrà preso in considerazione nel Paragrafo 8.8.

Valutazione dell'I/O nei supercomputer

Nei supercomputer l'I/O è dominato dagli accessi a file di grandi dimensioni memorizzati su dischi magnetici. Molti supercomputer eseguono delle applicazioni

batch, ciascuna delle quali può durare varie ore. In questi casi le operazioni di I/O consistono in una lettura di dati di grandi dimensioni, seguita da una sequenza di scritture che servono a fotografare il progressivo evolversi dei calcoli al fine di salvare il risultato parziale nel caso il calcolatore dovesse bloccarsi. Di conseguenza, le operazioni di I/O dei supercomputer consistono spesso più di operazioni di output che di input. La misura di I/O dominante per i supercomputer è il throughput dei dati: il numero di byte al secondo che può essere trasferito dalla memoria principale del supercomputer verso i dischi nel corso dei trasferimenti di grandi dimensioni.

Valutazione dell'I/O nell'elaborazione di transazioni

Le applicazioni *basate su transazioni* (*transaction processing o TP*) implicano sia vincoli sul tempo di risposta, sia misure di prestazioni basata sul throughput; inoltre la maggior parte degli accessi di I/O sono di piccole dimensioni. Di conseguenza le applicazioni TP sono essenzialmente dipendenti dalla *frequenza di I/O*, misurata in termini di numero di accessi a disco per secondo, piuttosto che dalla *frequenza di trasferimento di dato*, misurata in termini di byte di dato per secondo. Le applicazioni TP generalmente implicano modifiche a basi di dati di grandi dimensioni e prevedono che il sistema rispetti alcuni vincoli sul tempo di risposta, nonché sulla capacità di gestire in maniera indolore alcuni tipi di guasti; tali applicazioni sono estremamente critiche e sensibili ai costi. Ad esempio le banche usano i sistemi TP poiché richiedono che tutte le elaborazioni soddisfino a precise caratteristiche, quali la certezza di non perdere nessuna transazione, la velocità nella gestione delle transazioni e la minimizzazione del costo di *elaborazione* di ciascuna transazione. Anche se l'affidabilità a fronte di un guasto è un vincolo irrinunciabile, sia il tempo di risposta che il throughput sono fattori critici nella costruzione di sistemi attraenti in termini di costo.

Nella valutazione dell'elaborazione di transazioni sono stati sviluppati diversi casi tipici assunti come problemi campione (benchmark): il più noto di questi è stato sviluppato dal *Transaction Processing Council* (TPC). Le versioni più recenti di questi benchmark sono denominate TPC-C e TPC-D: ambedue comprendono l'elaborazione di interrogazioni relative ad una base dati. TPC-C contiene interrogazioni di costo basso e medio in un ambiente per la gestione degli ordini, ma che possono essere considerate tipiche anche delle transazioni utilizzate in un sistema di prenotazione o in un sistema bancario on-line. TPC-D prevede interrogazioni complesse tipiche di applicazioni di supporto alle decisioni.

TPC-C è significativamente più complesso dei precedenti insiemi di benchmark TPC-A e TPC-B. TPC-C comprende nove diversi tipi di record di base dati, cinque diversi tipi di transazioni ed un modello di richiesta di transazioni inteso a simulare un utente reale che genera delle transazioni ad un terminale. Le specifiche dei benchmark, comprese le regole per documentare i risultati ottenuti, sono lunghe 128 pagine! Le prestazioni su TPC-C sono misurate in transazioni per minuto o per secondo (TPM o TPS) e comportano la realizzazione di misure relative all'intero sistema, includendo l'I/O relativo ai dischi, l'I/O relativo ai terminali ed il calcolo. Una descrizione dettagliata di come sia organizzato il TPC e dei suoi problemi campione è disponibile attraverso il link al sito del TPC all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*.

Valutazione dell'I/O nei file system

I file system, memorizzati su disco, presentano diverse tipologie di accesso. Per esempio, misure effettuate su un file system Unix in un ambiente di tipo ingegneristico hanno rivelato che l'80% degli accessi è relativo a file di dimensioni minori di 10 KB e che il 90% di tutti gli accessi a file sono relativi a dati posti ad indirizzi consecutivi sul disco. Inoltre il 67% degli accessi corrisponde a letture, il 27% a scritture, ed il 6% ad accessi del tipo leggi-modifica-scrivi, che eseguono la lettura di un dato, la sua modifica, e infine la sua riscrittura nella stessa locazione. Tali misure hanno condotto alla creazione di alcuni benchmark specifici per i file system. Uno dei più diffusi tra questi si compone di cinque fasi ed utilizza 70 file, per una dimensione complessiva di 200 KB:

- *MakeDir*: costruzione di un sottoalbero di directory identico in termini di struttura ad un dato sottoalbero di directory.
- *Copy*: copia di tutti i file del sottoalbero sorgente nel sottoalbero destinazione.
- *ScanDir*: attraversamento ricorsivo di un sottoalbero di directory e verifica dello stato di ogni file in esso contenuto.
- *ReadAll*: lettura di tutti i byte di tutti i file di un sottoalbero.
- *Make*: compilazione e link di tutti i file in un sottoalbero.

Come si vedrà nel Paragrafo 8.6, il progetto di un sistema di I/O implica la conoscenza del carico di lavoro previsto.

8.3 Tipi e caratteristiche dei dispositivi di I/O

I dispositivi di I/O sono incredibilmente diversi. Per organizzare in maniera sistematica una tale varietà è utile fare riferimento a tre caratteristiche:

- *Comportamento*: input (singola lettura), output (sola scrittura, senza possibilità di lettura), o memorizzazione (può essere riletto e normalmente anche riscritto).
- *Partner*: è possibile che all'altro capo del dispositivo di I/O vi sia un essere umano o una macchina ad immettere i dati in input o a leggere i dati in output.
- *Frequenza di trasferimento dati*: la frequenza di picco alla quale i dati possono essere trasferiti tra il dispositivo di I/O e la memoria principale o il processore. È utile conoscere la massima richiesta che il dispositivo può generare.

Per esempio una tastiera è un dispositivo di input utilizzato da un essere umano con una frequenza di dato di picco di circa 10 byte al secondo; la figura 8.2 elenca alcuni dei dispositivi di I/O che possono essere collegati ad un calcolatore.

Nel Capitolo 1 sono stati descritti quattro dispositivi di I/O particolarmente importanti e rappresentativi: i mouse, i video grafici, i dischi e le reti. I mouse, i dischi e le reti verranno utilizzati come esempi per illustrare come i dispositivi di I/O si interfacciano verso i processori e le memorie; prima è però utile discutere questi dispositivi in maggior dettaglio rispetto a quanto fatto nel Capitolo 1.

Dispositivo	Comportamento	Partner	Frequenza di dato (KB/s)
Tastiera	input	essere umano	0,01
Mouse	input	essere umano	0,02
Input vocale	input	essere umano	0,02
Scanner	input	essere umano	400,00
Output vocale	output	essere umano	0,60
Stampante	output	essere umano	1,00
Stampante laser	output	essere umano	200,00
Video grafico	output	essere umano	60000,00
Modem	input o output	macchina	2,00 – 8,00
Rete locale	input o output	macchina	500,00 – 6000,00
Floppy disk	memorizzazione	macchina	100,00
Disco ottico	memorizzazione	macchina	1000,00
Nastro magnetico	memorizzazione	macchina	2000,00
Disco magnetico	memorizzazione	macchina	2000,00 – 10000,00

Figura 8.2 La varietà dei dispositivi di I/O. I dispositivi di I/O possono essere distinti a seconda che fungano da dispositivi di input, output o memorizzazione e a seconda del partner (persona o altro calcolatore) e della frequenza di comunicazione di picco. Le frequenze di dato coprono sei ordini di grandezza. Si noti che una rete può essere un dispositivo di input o di output, ma non può essere usata per la memorizzazione. Le dimensioni dei dischi, così come le frequenze di trasferimento dei dispositivi, sono sempre espresse in base 10, così che 1 MB = 1000000 byte, e 10 Mbit/s = 10000000 bit/s.

Il mouse

L'interfaccia tra un mouse ed il sistema può assumere una tra le seguenti due forme: o il mouse genera una serie di impulsi quando viene mosso (usando per generare gli impulsi il LED ed il rivelatore descritti nel Capitolo 1), o incrementa e decrementa dei contatori. La figura 8.3 mostra come i contatori si modificano quando il mouse viene mosso e descrive come funzionerebbe l'interfaccia se invece generasse degli impulsi. Il processore può leggere periodicamente i contatori oppure contare gli impulsi, e determinare così di quanto si è mosso il mouse dall'ultima volta che è stato esaminato; il sistema provvede poi a muovere in maniera appropriata il cursore sullo schermo. Il movimento appare graduale in quanto la velocità alla quale viene mosso il mouse è bassa rispetto alla velocità alla quale il processore può leggere lo stato del mouse e muovere il cursore sullo schermo.

Molti mouse possiedono inoltre uno o più pulsanti, ed il sistema deve essere in grado di rilevare quando un pulsante è stato premuto. Osservando lo stato del pulsante, il sistema può inoltre distinguere tra il caso in cui il pulsante viene premuto ed immediatamente rilasciato (clic) ed il caso in cui viene tenuto premuto. Ovviamente il legame tra i contatori, lo stato del pulsante e ciò che avviene sullo schermo è totalmente sotto il controllo del software: per questo motivo, ad esempio, la velocità con la quale il mouse si muove attraverso lo schermo e la velocità con la quale i clic singoli e doppi vengono rilevati sono normalmente decise dall'utente.

Analogamente il modo in cui il software interpreta la posizione del mouse fa sì che il cursore non esca completamente dallo schermo quando il mouse viene mosso di molto sempre nella stessa direzione. Il metodo in base al quale il sistema osserva lo stato del mouse leggendo i segnali che da questo provengono è quello comunemente usato nell'interfacciamento con i dispositivi con basse prestazioni e viene chiamato *polling* (*interrogazione*). Il metodo verrà nuovamente descritto nel Paragrafo 8.5.

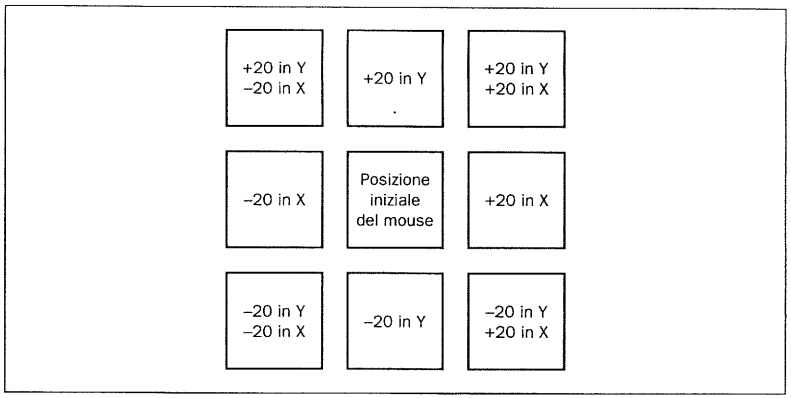


Figura 8.3 Lo spostamento del mouse in direzione orizzontale o verticale fa sì che i contatori X e Y (rispettivamente) vengano incrementati e decrementati. Se il mouse viene mosso in diagonale cambiano entrambi i contatori. Poiché la sfera non si muove quando il mouse non tocca la superficie, esso può essere sollevato e mosso senza che i contatori vengano modificati. Quando il mouse usa gli impulsi per comunicare i propri movimenti, si usano quattro tipi di impulsi: +X, –X, +Y e –Y. Anziché provocare un cambiamento nel valore del contatore, il mouse genera l'appropriato numero di impulsi su ciascuna delle quattro linee di segnale impulsive. Il valore 20 è una misura arbitraria di quanto il mouse si sia spostato.

Dischi magnetici

Come detto nel Capitolo 3, vi sono principalmente due tipi di dischi magnetici: i floppy disk e gli hard disk. Ambedue i tipi di dischi si basano su un piatto rotante rivestito di una superficie magnetica ed usano una testina mobile di lettura/scrittura per accedere al disco. La memorizzazione su disco è *non volatile*, intendendo con questo che i dati vengono conservati anche quando viene tolta l'alimentazione. Poiché i piatti di un hard disk sono metallici (o recentemente in vetro), esistono alcuni significativi vantaggi rispetto ai floppy disk:

- l'hard disk può essere più grande, dato che è rigido;
- l'hard disk ha una densità più elevata, dato che può essere controllato con maggiore precisione;
- l'hard disk ha una maggiore velocità di dato, dato che ruota più velocemente;
- l'hard disk può contenere più di un piatto.

Nel resto di questo capitolo l'attenzione verrà concentrata sugli hard disk, e verrà usato il termine *dischi magnetici* per indicarli.

Un disco magnetico consiste di un insieme di piatti (in numero compreso tra 1 e 15), ognuno dei quali possiede due superfici utilizzabili per la scrittura, come mostrato nella figura 8.4. La pila di piatti viene fatta ruotare ad una velocità compresa tra i 3600 ed i 7200 giri al minuto (revolutions per minute, RPM), ed ha un diametro che può variare da poco più di un pollice (circa 2,5 cm) fino ad oltre otto pollici (circa 20 cm). Ciascuna superficie del disco è divisa in cerchi concentrici, chiamati *tracce*. Su ciascuna superficie vi sono tipicamente da 1000 a 5000 tracce. Una traccia è a sua volta divisa in *settori* che contengono le informazioni; ciascuna traccia può contenere da 64 a 200 settori, ed il settore è l'unità minima di lettura e

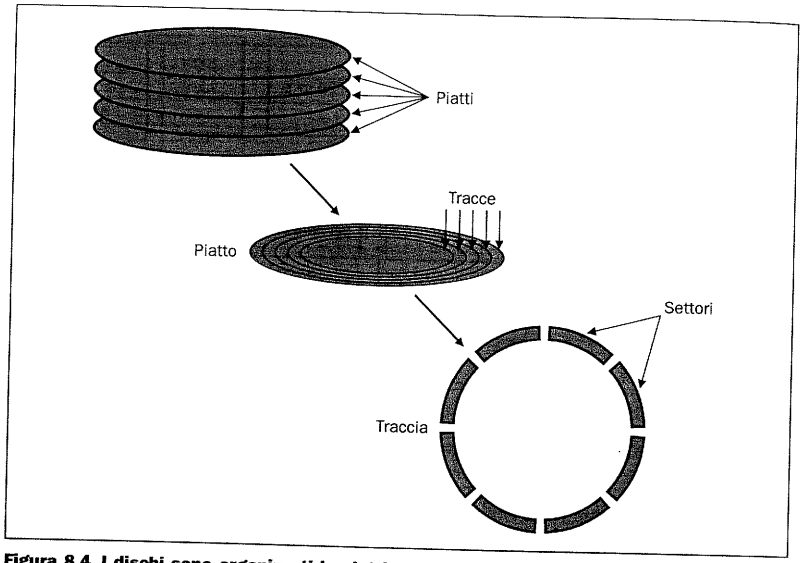


Figura 8.4 I dischi sono organizzati in piatti, tracce e settori. Entrambe le facce di un piatto sono rivestite di materiale magnetico, così che le informazioni possono essere memorizzate su tutte e due le superfici. I floppy disk hanno la stessa organizzazione, ma sono composti da un solo piatto.

scrittura. Nel 1997 la dimensione tipica dei settori era pari a 512 byte. La sequenza di dati memorizzati sul supporto magnetico comprende il numero del settore, un intervallo, le informazioni contenute nel settore, compreso un codice di correzione dell'errore (si veda l'Appendice B), un intervallo, il numero del settore successivo, e così via. Normalmente tutte le tracce contengono lo stesso numero di settori e quindi lo stesso numero di bit.

Come si è visto nel Capitolo 1, per leggere o scrivere delle informazioni le testine di lettura e scrittura devono essere mosse in maniera che vengano a trovarsi nella posizione corretta. Le testine dei dischi relative alle diverse superfici sono tra loro solidali e si muovono insieme, così che tutte le testine si trovano sulla stessa traccia, ciascuna sulla propria superficie. Si usa il termine *cilindro* per fare riferimento a tutte le tracce che in un certo istante si trovano al di sotto delle testine sulle varie superfici.

Per accedere ai dati il sistema operativo deve pilotare i dischi attraverso un processo composto da tre fasi. La prima fase consiste nel posizionare la testina al di sopra della traccia opportuna. Tale operazione viene denominata *seek* (letteralmente *ricerca*) ed il tempo necessario per spostare la testina sulla traccia desiderata è chiamato *tempo di seek*.

I produttori di dischi riportano nei loro manuali il tempo di seek minimo, massimo e medio. I primi due sono facilmente misurabili, mentre la media si presta a diverse interpretazioni, poiché dipende dalla distanza di posizionamento. Le industrie si sono accordate sul calcolare il tempo di seek medio come la somma di tutti i possibili tempi di seek divisa per il numero di possibili posizionamenti. I tempi di seek medi che vengono normalmente dichiarati oscillano tra 8 ms e 20 ms, ma a causa della località dei riferimenti al disco i valori medi reali del tempo di seek pos-

sono essere anche solo il 25%-33% dei valori dichiarati, a seconda delle applicazioni e delle modalità di ordinamento delle richieste. Il fenomeno della località si manifesta sia a causa di accessi consecutivi allo stesso file, sia perché il sistema operativo tenta di far eseguire insieme tali accessi.

Una volta che la testina ha raggiunto la traccia corretta, è necessario aspettare che il settore desiderato si muova (ruotando) sotto la testina di lettura/scrittura. Questo tempo viene chiamato *latenza di rotazione* o *ritardo di rotazione*. La latenza media per raggiungere l'informazione desiderata corrisponde a mezza rotazione del disco. Poiché i dischi ruotano ad una velocità compresa tra 3600 RPM e 7200 RPM, la latenza di rotazione media varia tra

$$\text{latenza di rotazione media} = \frac{0,5 \text{ rotazioni}}{3600 \text{ RPM}} = \frac{0,5 \text{ rotazioni}}{3600 \text{ RPM} \ / \ \left(60 \dfrac{\text{secondi}}{\text{minuti}}\right)} =$$

$$= 0,0083 \text{ secondi} = 8,3 \text{ ms}$$

e

$$\text{latenza di rotazione media} = \frac{0,5 \text{ rotazioni}}{7200 \text{ RPM}} = \frac{0,5 \text{ rotazioni}}{7200 \text{ RPM} \ / \ \left(60 \dfrac{\text{secondi}}{\text{minuti}}\right)} =$$

$$= 0,0042 \text{ secondi} = 4,2 \text{ ms}$$

Dischi caratterizzati da un minore diametro sono particolarmente attraenti in quanto possono ruotare a velocità maggiori senza un eccessivo consumo, riducendo così la latenza di rotazione.

L'ultima parte dell'accesso ad un disco, denominata *tempo di trasferimento*, corrisponde al tempo necessario a trasferire un blocco di bit, solitamente un settore. Il tempo di trasferimento è una funzione della dimensione del settore, della velocità di rotazione e della densità di memorizzazione di una traccia. Nel 1997 i tempi di trasferimento erano compresi tra 2 e 15 MB/s. La situazione è resa più complicata dal fatto che la maggior parte dei dischi di medio ed alto livello possiedono una cache integrata che memorizza i settori mano a mano che questi vengono acquisiti; la velocità di trasferimento dalla cache è di solito più elevata, e nel 1997 poteva arrivare sino a 40 MB/s. Oggi molti trasferimenti da disco hanno una lunghezza pari a più settori.

Il controllo del disco a basso livello ed il trasferimento dei dati tra il disco e la memoria vengono normalmente gestiti da un *controllore di disco*. Il controllore è responsabile dell'aggiunta di un'ulteriore componente al tempo di accesso al disco, nota come *tempo del controllore*, corrispondente al ritardo che il controllore impone all'esecuzione degli accessi di I/O. Il tempo medio per eseguire una operazione di I/O è determinato dalle quattro componenti introdotte, oltre agli eventuali tempi di attesa dovuti al fatto che altri processi stanno utilizzando il disco.

Esempio

Tempi di lettura da disco

Qual è il tempo medio per la lettura o la scrittura di un settore di 512 byte nel caso di un disco tipico, che ruota a 5400 RPM? Il tempo medio di seek indicato

dal costruttore è pari a 12 ms, il tempo di trasferimento a 5 MB/s, ed il tempo aggiuntivo causato dal controllore è 2 ms. Si assuma che il disco non sia utilizzato da altri, così che non vi siano tempi di attesa.

Soluzione

Il tempo medio di accesso a disco è pari al tempo medio di seek + il ritardo di rotazione medio + il tempo di trasferimento + il tempo del controllore. Utilizzando il tempo di seek medio indicato, la soluzione è

$$12\ \text{ms} + 5{,}6\ \text{ms} + \frac{0{,}5\ \text{KB}}{5\ \text{MB/s}} + 2\ \text{ms} = 12 + 5{,}6 + 0{,}1 + 2 = 19{,}7\ \text{ms}$$

Se il tempo medio di seek misurato fosse pari al 25% del tempo medio indicato, la risposta sarebbe

$$3\ \text{ms} + 5{,}6\ \text{ms} + 0{,}1\ \text{ms} + 2\ \text{ms} = 10{,}7\ \text{ms}$$

Si noti che quando si prende in considerazione il tempo medio di seek misurato, anziché quello indicato dal costruttore, la latenza di rotazione può divenire la componente principale del tempo di accesso.

La densità dei dischi ha continuato a crescere per oltre 40 anni. Gli effetti di questi progressi in termini di densità e la riduzione della dimensione fisica delle unità disco sono stati sorprendenti, come illustrato dalla figura 8.5. L'impegno di proget-

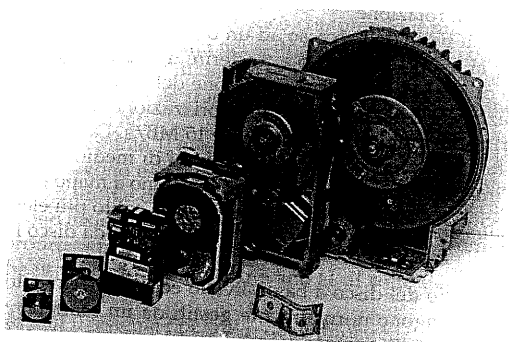


Figura 8.5 Sei dischi magnetici di diametro variabile tra 14 pollici (circa 35 cm) e 1,8 pollici (circa 4 cm). I dischi mostrati sono stati introdotti sul mercato più di 10 anni fa e non sono quindi da considerare come rappresentativi delle capacità massime dei dischi di pari diametro nel 1998. La fotografia riporta tuttavia con fedeltà le loro pari a dimensioni fisiche relative. Il disco più grande è il DEC R81, che contiene 4 piatti con diametro pari a 14 pollici (circa 35 cm) e capacità di immagazzinamento di 456 MB; veniva costruito nel 1985. Il disco da 8 pollici (circa 20 cm) di diametro proviene dalla Fujitsu, risale al 1984, e memorizza 130 MB su sei piatti. Il Micropolis RD53 ha cinque piatti da 5,25 pollici (circa 11 cm) e memorizza 85 MB. L'IBM 0361 possiede anch'esso cinque piatti, ma con un diametro di soli 3,5 pollici (circa 7 cm). Il disco risale al 1988 e contiene 320 MB. Nel 1997 il disco da 3,5 pollici con la maggiore densità comprende 10 piatti e nello stesso spazio contiene 9,1 GB, con un aumento in termini di densità pari a 30 volte! Il Conner CP 2045 ha due piatti da 2,5 pollici (circa 5,5 cm) contenenti 40 MB, ed era prodotto nel 1990. Il disco più piccolo nella fotografia è l'Integral 1820. Comprende un solo piatto da 1,8 pollici (circa 4 cm), contiene 20 MB ed era prodotto nel 1992. Fotografia di Peg Skorpinski.

Caratteristiche	Seagate ST423451	Seagate ST19171	Seagate ST92255
Diametro del disco (in pollici)	5,25	3,50	2,50
Capacità dopo la formattazione (MB)	23200	9100	2250
MTBF (ore)	500000	1000000	300000
Numero di superfici dei dischi	28	20	10
Velocità di rotazione (RPM)	5400	7.200	4500
Velocità di trasferimento interna (Mbit/s)	86-124	80-124	fino a 60,8
Interfaccia esterna	Fast SCSI-2 (8-16 bit)	Fast SCSI-2 (8-16 bit)	Fast ATA
Velocità di trasferimento esterna (MB/s)	20-40	20-40	fino a 16,6
Tempo minimo di seek (da traccia a traccia) (ms)	0,9	0,6	4
Tempo medio di seek + ritardo di rotazione (ms)	11	9	14
Consumo/contenitore (watt)	26	13	2,6
MB/w	892	700	865
Volume (cm^3)	5250	603	130
MB/ cm^3	1152	3936	4368

Figura 8.6 Caratteristiche di tre dischi magnetici di uno stesso produttore. I tre dischi corrispondono ai prodotti con massima densità per ciascuna dimensione all'interno della famiglia di prodotti Seagate (nel 1997). I dischi qui riportati richiedono un'interfaccia SCSI (il bus standard di I/O discusso nel Paragrafo 8.4) o ATA (un'interfaccia standard per i dischi in ambiente PC). Se confrontati con i dischi riportati nell'analoga tabella che comparve nella prima edizione di questo libro (nel 1994), i dischi qui considerati presentano valori tra le 25 e le 40 volte superiori in termini di MB/w e tra le 90 e le 450 volte in termini di MB/cm^3. MTBF sta per *Mean Time Between Failure* (tempo medio tra due guasti) e rappresenta la misura standard di affidabilità. I due dischi più grandi contengono delle cache che memorizzano il contenuto dei settori che vengono trasferiti. La velocità di trasferimento interna è la velocità alla quale i bit sono letti dalla superficie del disco, mentre la velocità di trasferimento esterna tiene conto della velocità alla quale può essere trasferito un settore che all'atto della richiesta già si trova nella cache. Per avere maggiori informazioni su queste unità, nonché sulla moderna tecnologia dei dischi, si veda il link alla Seagate all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*.

tisti diversi ha portato alla disponibilità in ogni momento di un'ampia varietà di dischi. La figura 8.6 mostra le caratteristiche di tre diversi dischi magnetici di uno stesso produttore. Le unità caratterizzate da un diametro elevato contengono molti più megabyte per ammortizzare il costo della parte elettronica, così che il buon senso farebbe pensare che queste unità siano caratterizzate da un minor costo per megabyte. In realtà questo vantaggio è compensato per le unità di piccole dimensioni da volumi di vendita molto più elevati, che abbassano i costi di produzione: nel 1997, i dischi costavano tra 10 e 20 centesimi di dollaro per megabyte, quasi indipendentemente dalla dimensione. I dischi più piccoli presentano inoltre dei vantaggi in termini di consumo e volume occupato per byte, come mostrato nella figura 8.6.

Approfondimento: Molti dischi recenti comprendono una cache inclusa direttamente nell'unità disco. Tale cache permette accessi veloci ai dati che sono stati letti recentemente in seguito alle richieste fatte dalla CPU. Ovviamente tale situazione complica la valutazione delle prestazioni del disco ed aumenta l'importanza del carico di lavoro prescelto. L'unità da 5,25 pollici della Seagate mostrata nella figura 8.6 viene fornita con una cache integrata.

Approfondimento: Ciascuna traccia contiene lo stesso numero di bit, ma le tracce esterne sono più lunghe. Le tracce esterne quindi memorizzano le informazioni con una minore densità per centimetro rispetto alle tracce più vicine al centro del disco. Grazie all'avvento di

interfacce standard intelligenti quali lo SCSI (si veda il Paragrafo 8.4) si sta oggi diffondendo una tecnica basata sulla memorizzazione di un numero maggiore di settori nelle tracce esterne rispetto alle tracce interne (tecnica definita *densità di bit costante*). La velocità alla quale l'unità di lunghezza della singola traccia si muove sotto la testina è quindi variabile, ed è maggiore per le tracce esterne. Conseguentemente, se il numero di bit per unità di lunghezza è costante, la frequenza alla quale i bit devono essere letti o scritti cambia, e la parte elettronica del dispositivo deve tener conto di questo fattore se viene adottata la tecnica della densità di bit costante.

Le reti

Le reti sono il mezzo più usato per la comunicazione tra calcolatori; alcune delle caratteristiche chiave di una rete tipica sono:

- la *distanza*: tra 0,01 e 10000 chilometri;
- la *velocità*: tra 0,001 MB/s e 100 MB/s;
- la *topologia*: a bus, ad anello, a stella, ad albero;
- la *condivisione di linee*: nessuna condivisione (connessione punto-punto) o condivisione (multidrop).

Tali caratteristiche verranno illustrate con tre esempi.

Lo standard RS-232 permette la realizzazione di una *rete di terminali* con velocità compresa tra 0,3 e 19,2 Kbit/s. Un calcolatore centrale è connesso a diversi terminali tramite connessioni lente ma dedicate. Queste connessioni punto-punto costituiscono una stella che si diparte dal calcolatore centrale, nella quale ciascun terminale si trova ad una distanza compresa tra 10 e 100 metri dal calcolatore.

La *rete locale* (Local Area Network o LAN) è ciò a cui normalmente si pensa quando si fa comunemente riferimento ad una rete, ed Ethernet è ciò che spesso la gente intende quando parla di LAN. In pratica infatti Ethernet è divenuta un termine così comune da essere spesso usata come sinonimo di LAN. La versione base di Ethernet è essenzialmente un bus costituito da un singolo filo, con velocità di 10 Mbit/s, senza alcun controllo centralizzato. I messaggi, o *pacchetti*, sono inviati su Ethernet in blocchi di dimensione variabile tra 64 e 1518 byte. Di recente alcune società hanno sviluppato una versione più veloce (chiamata normalmente Fast Ethernet) che offre velocità 10 volte maggiori (cioè intorno ai 100 Mbit/s) e per il 1998 è stato annunciato il rilascio di una Gigabit Ethernet.

Una Ethernet è essenzialmente un bus con master multipli ed uno schema per definire chi debba ottenere il controllo del bus: negli esercizi verrà discusso come sia implementato il controllo distribuito. Poiché Ethernet è un bus, ad ogni istante può esservi un solo trasmettitore, e questo limita l'ampiezza di banda. Nella pratica normalmente questo non è un problema, in quanto l'utilizzo del bus è relativamente basso. Ovviamente alcune LAN possono divenire sovraccariche per scarse capacità di pianificazione, così che il tempo di risposta ed il throughput possono rapidamente degradare al crescere dell'utilizzo.

Un modo in cui sono stati superati i limiti della Ethernet originale basata su bus è attraverso le reti a commutazione. In una *rete a commutazione* si introducono dei commutatori che permettono di ridurre il numero di dispositivi collegati (host) a ciascun segmento di Ethernet. Nel caso limite vi è un solo host per ciascun segmento, e l'host è direttamente connesso ad un commutatore. Le reti a commutazione sono diffuse per reti su lunghe distanze, che costituiscono l'argomento successivo, ma sono di recente divenute comuni anche in applicazioni di reti locali, dal momento che l'uso di elaboratori ad elevate prestazioni e la diffusione di informazioni di tipo multimediale ha imposto dei carichi notevoli alle reti Ethernet condivise.

Le *reti a lunga distanza* coprono distanze comprese tra 10 e 10000 kilometri. La prima e più famosa rete a lunga distanza fu ARPANET (così chiamata a ragione dell'ente che la finanziò, ossia la Advanced Research Projects Agency del governo americano). Essa trasferiva dati a 56 Kbit/s ed usava linee dedicate punto-punto noleggiate dalle società telefoniche. Il calcolatore host colloquiava con un *processore di interfaccia per i messaggi* (Interface Message Processor o IMP), che comunicava sulla linea telefonica. Lo IMP prendeva l'informazione e la spezzava in pacchetti da 1 Kbit, che potevano prendere anche cammini diversi verso il nodo destinazione. Ad ogni passo il pacchetto veniva memorizzato (in maniera da poter essere recuperato in caso di guasto) e poi inviato all'opportuno IMP sulla base dell'indirizzo contenuto nel pacchetto. Lo IMP destinazione ricombinava i pacchetti a formare un messaggio e lo passava all'host. La maggior parte delle reti di oggi utilizzano questo approccio, denominato *a commutazione di pacchetto* (packet-switched), nel quale i singoli pacchetti provenienti dalla sorgente sono instradati separatamente verso la destinazione.

ARPANET è stata il precursore di Internet. La chiave per interconnettere le diverse reti stava nella standardizzazione di una singola famiglia di protocolli, ossia TCP/IP (Transmission Control Protocol/Internet Protocol). La parte IP del protocollo fornisce il supporto per l'indirizzamento di due host su Internet, ma non garantisce l'affidabilità della consegna. TCP costituisce un protocollo che può garantire che tutti i pacchetti siano ricevuti e che i pacchetti non soffrano di errori di trasmissione. I due protocolli lavorano congiuntamente costituendo uno *stack di protocolli*, nella quale i pacchetti TCP sono incapsulati nei pacchetti IP. La standardizzazione del formato del pacchetto TCP/IP è ciò che permette a host e reti diversi di comunicare tra loro.

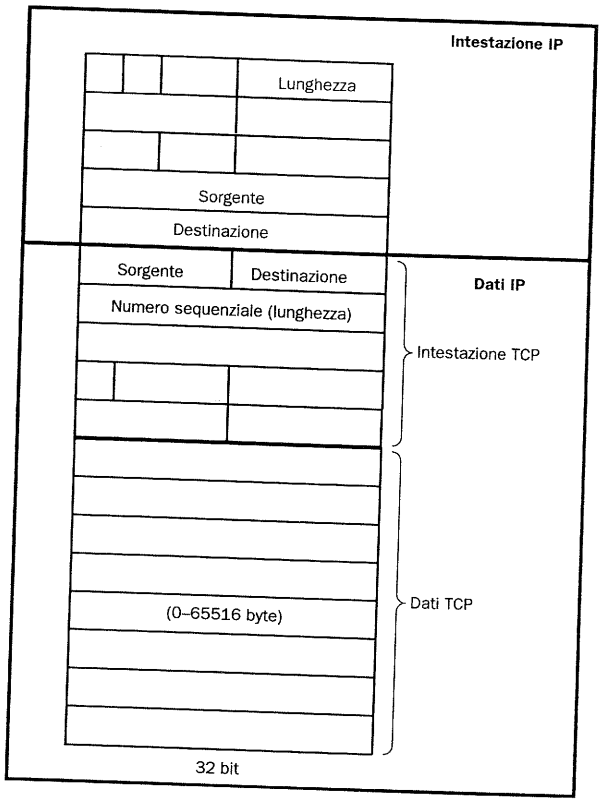
L'ampiezza di banda offerta dalle reti sta probabilmente crescendo oggi più velocemente di quella di qualunque altro tipo di dispositivo. Le reti ad alta velocità che utilizzano cavi di rame e cavi coassiali offrono bande di 100 Mbit/s, mentre la fibra ottica offre bande fino ad 1 Gbit/s. In futuro sembra prevedibile che tecnologie del tipo di quella di Internet possano essere estese sino a velocità dell'ordine di 1 Gbit/s. È probabile che queste super-reti ad alta velocità saranno a commutazione di pacchetto anziché su canali condivisi.

Un'altra tecnologia di successo tra quelle emergenti nel campo delle reti è ATM (Asynchronous Transfer Method). ATM è una tecnologia di rete scalabile (da 155 Mbit/s a 2,5 Gbit/s) che nacque originariamente per le reti a commutazione per fonia e dati su lunghe distanze. Già oggi è utilizzata in molte applicazioni del tipo a commutazione e giocherà un ruolo importante, insieme con approcci quali Fast Ethernet, nel campo della connessione dei calcolatori da tavolo.

La difficoltà nel mettere in servizio queste reti risiede principalmente nella costruzione di sistemi in grado di supportare efficacemente i nuovi mezzi di comunicazione, e nell'offrire l'ampiezza di banda richiesta ogni volta che due programmi intendono comunicare. Per vincere questa sfida è necessario che tutte le parti del sistema di I/O, dal sistema operativo al sistema di memoria, al bus, all'interfaccia dei dispositivi, siano in grado di fornire tale ampiezza di banda. Si tratta davvero di una sfida che coinvolge per intero il sistema.

Interfaccia Hardware Software

Per permettere la comunicazione tra reti diverse e con diverse caratteristiche, il TCP/IP definisce un formato standard di pacchetto. Un pacchetto IP, che contiene le informazioni sull'indirizzo Internet, comprende un pacchetto TCP che contiene sia le informazioni di indirizzo interpretate dall'host che i dati che devono venire trasmessi.



Per capire l'importanza, quando si parla di prestazioni, di considerare il sistema in maniera complessiva, sia dal punto di vista hardware che software, si consideri l'esempio seguente.

Esempio

Prestazioni di due reti

Si considerino le seguenti misure, eseguite su una coppia di SPARCstation 10 equipaggiate con Solaris 2.3, e collegate a due tipi diversi di rete, utilizzanti TCP/IP per la comunicazione:

Caratteristiche	Ethernet	ATM
Ampiezza di banda dal nodo alla rete	1,125 MB/s	10 MB/s
Latenza di interconnessione	15 ms	50 ms
Latenza HW da/verso la rete	6 ms	6 ms
Ritardo introdotto dal SW nell'invio alla rete	200 ms	207 ms
Ritardo introdotto dal SW nella ricezione da rete	241 ms	360 ms

Si calcoli la latenza tra i due host per un messaggio di 250 byte trasmesso utilizzando ciascuna delle due reti.

Soluzione

Il valore di tempo richiesto può essere stimato come somma delle latenze fisse più il tempo per la trasmissione del messaggio. Il tempo per trasmettere il messaggio è dato semplicemente dalla lunghezza del messaggio divisa per l'ampiezza di banda della rete.

I tempi di trasmissione sono

$$\text{tempo di trasmissione}_{\text{Ethernet}} = \frac{250\ \text{byte}}{1{,}125 \times 10^6\ \text{byte/s}} = 222\ \mu\text{s}$$

$$\text{tempo di trasmissione}_{\text{ATM}} = \frac{250\ \text{byte}}{10 \times 10^6\ \text{byte/s}} = 25\ \mu\text{s}$$

Ne consegue che il tempo di trasmissione per la rete ATM è circa 9 volte inferiore.
La latenza totale nell'invio e nella ricezione del pacchetto è la somma del tempo di trasmissione e dei ritardi introdotti dall'hardware e dal software:

$$\text{tempo totale}_{\text{Ethernet}} = 15 + 6 + 200 + 241 + 222 = 684\ \mu\text{s}$$

$$\text{tempo totale}_{\text{ATM}} = 50 + 6 + 207 + 360 + 25 = 648\ \mu\text{s}$$

La latenza complessiva della rete Ethernet è quindi solo 1,06 volte maggiore, anche se il tempo di trasmissione è quasi 9 volte più elevato!

8.4 I bus: la connessione tra i dispositivi di I/O, il processore e la memoria

In un sistema di elaborazione i vari sottosistemi devono essere interfacciati l'uno con l'altro; ad esempio processore e memoria devono comunicare, così come devono farlo il processore ed i dispositivi di I/O. Tale obiettivo è normalmente ottenuto

tramite l'uso di un *bus*; un bus è un canale di comunicazione condiviso, che usa un insieme di fili per collegare diversi sottosistemi. I due maggiori vantaggi dell'organizzazione a bus sono la versatilità ed il basso costo. Tramite la definizione di un unico schema di connessione, è facile aggiungere nuovi dispositivi, eventualmente limitando il throughput massimo di I/O; quando l'I/O deve transitare attraverso un singolo bus, l'ampiezza di banda del bus ne limita il throughput massimo. Nei sistemi per applicazioni commerciali, nei quali l'I/O è molto frequente, e nei supercalcolatori, nei quali le velocità di I/O devono essere molto alte perché le prestazioni del processore sono elevate, il progetto di un sistema a bus capace da un lato di soddisfare le richieste del processore e dall'altro di connettere all'elaboratore un elevato numero di dispositivi di I/O rappresenta un problema reale.

Una delle ragioni per cui il progetto di un bus è così complesso è che la velocità massima del bus è in larga misura limitata da fattori fisici, quali la lunghezza del bus ed il numero dei dispositivi; questi limiti fisici impediscono di far lavorare il bus ad una velocità arbitraria. Pur non superando questi limiti, esiste un'ampia gamma di tecniche che possono essere usate per aumentare le prestazioni del bus; queste tecniche possono tuttavia influenzare negativamente altre misure di prestazioni. Ad esempio per ottenere un tempo di risposta veloce per le operazioni di I/O è necessario minimizzare il tempo richiesto per un accesso a bus ottimizzando il canale di comunicazione. D'altra parte per avere con continuità delle elevate prestazioni di I/O si deve massimizzare l'ampiezza di banda del bus. L'ampiezza di banda del bus può aumentare se si utilizza un numero maggiore di buffer e trasferendo blocchi di dati di dimensioni maggiori, ma entrambe le tecniche aumentano il ritardo nel completare l'accesso a bus; è chiaro che i due obiettivi, accesso veloce a bus e ampiezza di banda elevata, possono portare a vincoli di progetto tra loro contrastanti. Infine la necessità di supportare un'ampia gamma di dispositivi con valori di latenza e velocità di trasferimento dati ampiamente variabili complica ulteriormente il progetto di un bus.

Un bus contiene normalmente un insieme di linee di controllo ed un insieme di linee di dato. Le linee di controllo sono utilizzate per avanzare delle richieste e segnalarne il ricevimento, nonché per indicare il tipo di informazione presente sulle linee di dato. Le linee di dato del bus portano le informazioni dalla sorgente alla destinazione; tali informazioni possono consistere di dati, comandi complessi, o indirizzi. Ad esempio se un disco vuole scrivere in memoria dei dati provenienti da un settore, le linee di dato verranno utilizzate per specificare l'indirizzo in memoria nel quale collocare i dati, nonché per trasportare i dati effettivi provenienti dal disco; le linee di controllo saranno usate per indicare quale tipo di informazione sia contenuta sulle linee di dato del bus in ciascun momento del trasferimento. Alcuni bus possiedono due insiemi di linee di segnale per trasferire separatamente dati ed indirizzi nel corso di un'unica fase di trasmissione su bus. In entrambi i casi le linee di controllo sono utilizzate per indicare ciò che è presente sul bus e per implementare il protocollo del bus stesso. Dal momento che il bus è condiviso, è inoltre necessario un protocollo per decidere chi lo utilizzerà nella fase successiva; questo problema verrà affrontato tra breve.

Si consideri una tipica *transazione su bus*. Una transazione su bus comprende due fasi: l'invio dell'indirizzo e la ricezione o l'invio del dato. Le transazioni sul bus sono tipicamente caratterizzate dall'operazione che eseguono sulla memoria: una transazione di *lettura* trasferisce dati *dalla* memoria (verso il processore o verso un dispositivo di I/O), mentre una transazione di *scrittura* scrive i dati *nella* memoria. La terminologia riportata è chiaramente ambigua, e per superare il problema si cer-

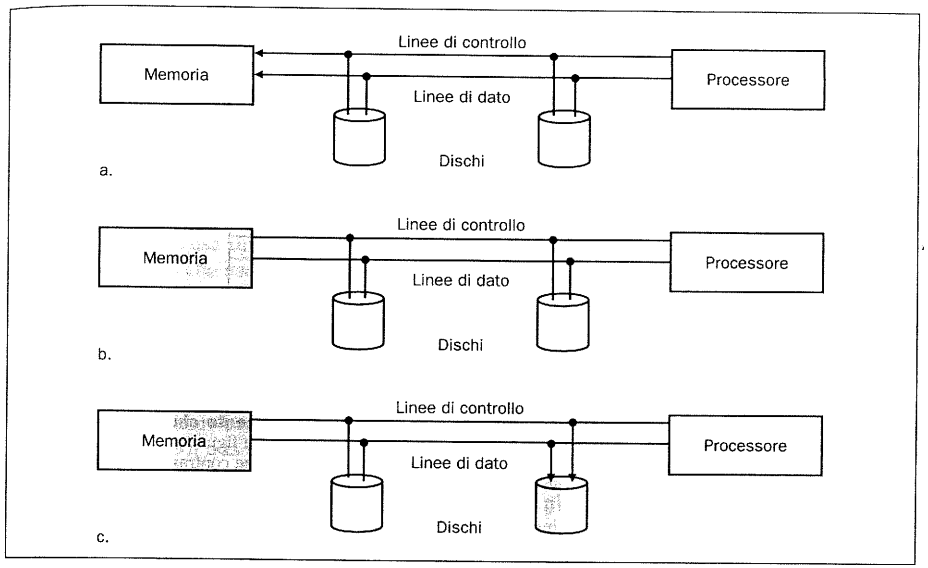


Figura 8.7 I tre passi che costituiscono un'operazione di output. Ad ogni passo i componenti che partecipano attivamente alla comunicazione sono mostrati in colore contrastante; la parte destra è ombreggiata se il dispositivo sta eseguendo un'operazione di lettura, mentre è ombreggiata quella di sinistra se il dispositivo sta eseguendo un'operazione di scrittura. Si noti che le linee di dato del bus possono trasportare sia un indirizzo (come nel caso a), sia un dato (come nel caso c). (a) Il primo passo in un'operazione di output corrisponde ad una lettura dalla memoria. Le linee di controllo segnalano una richiesta di lettura alla memoria, mentre le linee di dato contengono l'indirizzo. (b) Nel secondo passo la memoria fa accesso al dato. (c) Nel terzo ed ultimo passo la memoria trasferisce i dati utilizzando le linee di dato del bus e segnala al dispositivo di I/O che il dato è disponibile usando le linee di controllo. Il dispositivo memorizza il dato così come appare sul bus.

cherà di usare i termini *input* e *output*, che sono sempre definiti rispetto al processore: un'operazione di input corrisponde ad acquisire dati dal dispositivo verso la memoria, dove il processore può poi leggerli; un'operazione di output invia dati ad un dispositivo prelevandoli dalla memoria, dove sono stati scritti dal processore. La figura 8.7 mostra i passi che compongono una tipica operazione di output, nella quale i dati vengono letti dalla memoria ed inviati al dispositivo. La figura 8.8 mostra i passi che compongono un'operazione di input, nella quale i dati sono letti dal dispositivo e scritti nella memoria. In entrambe le figure le parti attive del bus e della memoria sono evidenziate in colore contrastante, ed un'operazione di lettura o scrittura è indicata evidenziando con l'ombreggiatura l'unità coinvolta, come era stato fatto nel Capitolo 6. In queste figure si concentrerà l'attenzione sul modo in cui i dati sono trasferiti tra il dispositivo di I/O e la memoria; nel Paragrafo 8.5 si vedrà come vengono attivate le operazioni di I/O.

Tipi di bus

I bus sono tradizionalmente classificati in tre categorie: *bus processore-memoria*, *bus di I/O*, *bus di backplane*. I bus processore-memoria sono corti, normalmente caratterizzati da un'alta velocità, e ottimizzati rispetto alle caratteristiche del sistema di memoria, in maniera tale da massimizzare l'ampiezza di banda memoria-processo-

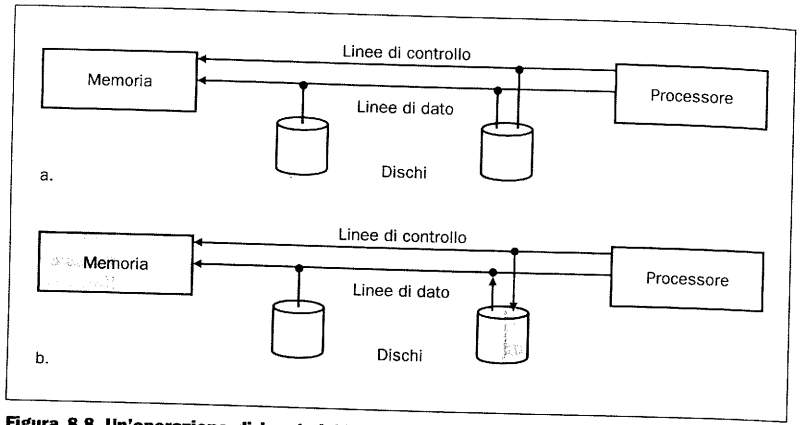


Figura 8.8 Un'operazione di input richiede meno tempo, dal momento che il dispositivo non deve aspettare la memoria per fare accesso ai dati. Come nella figura precedente, i partecipanti attivi di ogni passo della comunicazione sono mostrati in colore contrastante, e la parte destra è ombreggiata se il dispositivo sta eseguendo un'operazione di lettura, mentre è la parte sinistra ad essere ombreggiata se il dispositivo sta eseguendo un'operazione di scrittura. (a) Nel primo passo di una operazione di input le linee di controllo indicano un'operazione di scrittura per la memoria, mentre le linee di dato contengono un indirizzo. (b) Il secondo passo si svolge quando la memoria è pronta e lo comunica al dispositivo, che a quel punto trasferisce il dato; la memoria memorizzerà il dato non appena l'avrà ricevuto. Il dispositivo non ha la necessità di attendere che l'operazione di memorizzazione sia stata completata. Nei passi mostrati si assume che il dispositivo debba attendere che la memoria segnali di essere pronta, anche se questo non si verifica in alcuni sistemi che utilizzano dei buffer o possiedono un sotto-sistema di memoria veloce.

re. Diversamente da questi, i bus di I/O sono caratterizzati da una maggiore lunghezza, permettono di collegare dispositivi di tipo diverso, e tollerano spesso un ampio intervallo della banda dei dati dei dispositivi connessi. Tipicamente i bus di I/O non si interfacciano direttamente con la memoria, ma usano o un bus processore-memoria, o un bus di backplane per connettersi ad essa. I bus di backplane sono progettati in maniera da permettere ai processori, alla memoria ed ai dispositivi di I/O di coesistere su un singolo bus; essi sono un compromesso tra le esigenze della comunicazione tra processore e memoria e quelle della comunicazione tra dispositivi di I/O e memoria. I bus di backplane derivano il loro nome dal fatto che sono spesso ricavati all'interno del backplane, che è la struttura di interconnessione esistente all'interno del telaio; i processori, la memoria e le schede di I/O sono quindi inserite nel backplane ed utilizzano il bus per la comunicazione.

I bus processore-memoria sono spesso specifici di un singolo progetto, mentre sia i bus di I/O che i bus di backplane sono spesso utilizzati in elaboratori diversi. I bus di backplane e di I/O sono spesso dei bus *standard*, utilizzati da calcolatori diversi costruiti da aziende diverse. Al contrario i bus processore-memoria sono spesso proprietari (ossia definiti ed utilizzati da una singola azienda). In molti recenti elaboratori il bus processore-memoria può coincidere con il bus di backplane, ed i bus standard, o quelli di I/O, si connettono al bus processore-memoria. In molti elaboratori recenti la distinzione tra i vari tipi di bus, in particolare tra i bus di backplane ed i bus processore-memoria, è divenuta di poco conto.

Durante la fase di progetto il progettista di un bus processore-memoria conosce esattamente il tipo di tutti i dispositivi che devono essere connessi al bus, mentre il

progettista di un bus di I/O o di backplane deve progettare il bus in modo da poter accettare dispositivi sconosciuti con caratteristiche diverse in termini di latenza e ampiezza di banda. Normalmente un bus di I/O presenta un'interfaccia relativamente semplice e di basso livello verso il dispositivo, in maniera da richiedere l'aggiunta di circuiteria minima per consentire l'interfaccia al bus. Di norma un bus di backplane richiede invece della logica aggiuntiva per implementare l'interfaccia tra il bus ed un dispositivo, o tra il bus di backplane ed un bus di I/O di livello inferiore; in termini di costo un bus di backplane offre il vantaggio di avere un unico bus. La figura 8.9 mostra un sistema che usa un singolo bus di backplane, un sistema che usa un bus processore-memoria cui sono connessi dei bus di I/O, ed un sistema che usa tutti e tre i tipi di bus; gli elaboratori che possiedono un bus processore-memoria separato normalmente usano un adattatore di bus per collegare il bus di I/O con il bus processore-memoria. Alcuni sistemi espandibili ad alte prestazioni usano un'organizzazione in cui sono combinati i tre bus: il bus processore-memoria ha uno o più adattatori di bus che interfacciano un bus di backplane standard con il bus processore-memoria. I sistemi IBM RS/6000 ed i sistemi multiprocessore Silicon Graphics utilizzano questo tipo di organizzazione; essa offre il vantaggio che il bus processore-memoria può essere reso molto più veloce di un bus di backplane o di I/O, e che il sistema di I/O può essere esteso inserendo diversi controllori di I/O o diversi bus nel bus di backplane senza che venga influenzata l'efficienza del bus processore-memoria.

Bus sincroni e asincroni

Le differenze condizioni nelle quali viene progettato un bus processore-memoria, un bus di I/O o di backplane fanno sì che esistano due diversi schemi di comunicazione sul bus: *sincrono* e *asincrono*. Se un bus è sincrono, comprende tra le linee di controllo un segnale di clock ed un protocollo fisso per la comunicazione basato sulla tempistica di clock. Ad esempio se si considera un bus processore-memoria che esegue un'operazione di lettura dalla memoria, si potrebbe pensare ad un protocollo che trasmetta l'indirizzo ed il comando di lettura nel primo ciclo di clock, usando le linee di controllo per indicare il tipo di richiesta; la memoria potrebbe poi rispondere producendo la parola di dato nel quinto ciclo di clock. Questo tipo di protocollo può essere facilmente implementato utilizzando una piccola macchina a stati finiti. Poiché il protocollo è predeterminato e richiede una quantità ridotta di logica, il bus può lavorare molto velocemente e la logica di interfaccia sarà ridotta. I bus sincroni presentano tuttavia due principali svantaggi: il primo è che tutti i dispositivi sul bus devono lavorare alla stessa frequenza di clock. Il secondo è la ridotta lunghezza del bus necessaria a mantenere alte velocità per problemi di disallineamento del clock (si veda l'Appendice B per una discussione sui problemi di disallineamento del clock). I bus processore-memoria sono spesso sincroni, dal momento che i dispositivi che essi collegano sono tra loro vicini, in numero ridotto e predisposti ad operare con frequenze di clock elevate.

Un bus asincrono non possiede un segnale di clock; quindi può supportare un'ampia gamma di dispositivi ed il bus può essere reso lungo a piacere senza che insorgano problemi di disallineamento del clock o di sincronizzazione. Per coordinare la trasmissione dei dati tra il trasmettitore ed il ricevitore un bus asincrono usa un protocollo detto di *handshaking* (letteralmente *stretta di mano*). Un protocollo di handshaking consiste di una serie di passi nei quali il dispositivo che invia i dati e quello che li riceve accedono al passo successivo solo quando ambedue le parti

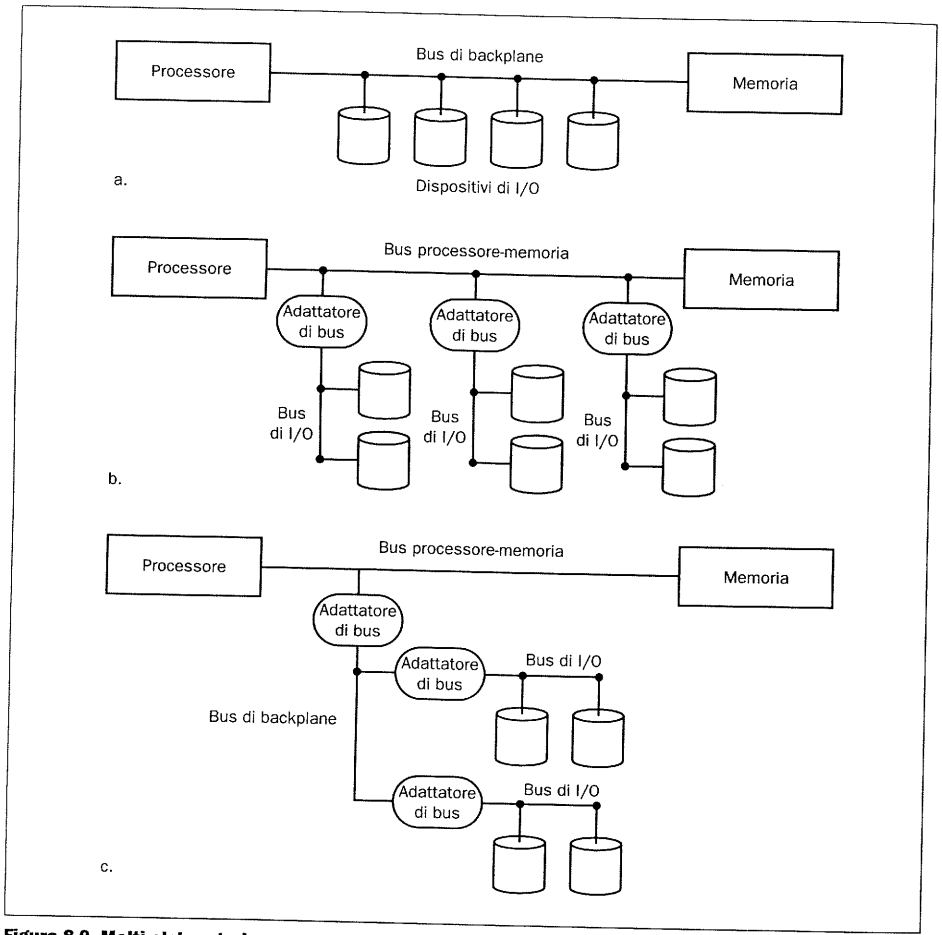


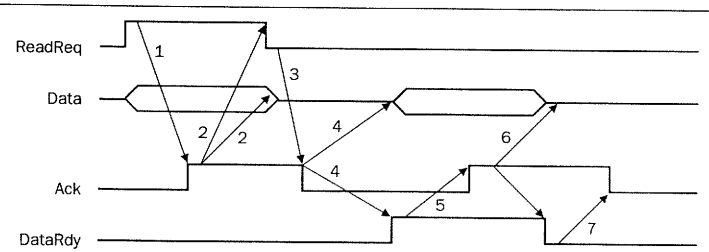
Figura 8.9 Molti elaboratori usano un unico bus di backplane sia per il traffico tra processore e memoria che per quello di I/O. Alcuni elaboratori ad alte prestazioni usano un bus processore-memoria separato al quale si collegano i bus di I/O. Alcuni sistemi fanno uso di tutti e tre i tipi di bus, organizzati in maniera gerarchica. (a) Si usa un singolo bus per la comunicazione tra processore e memoria, nonché per la comunicazione tra dispositivi di I/O e memoria. I bus usati nei vecchi PC adottavano questa soluzione. (b) Si usa un bus separato per il traffico tra processore e memoria. Per trasferire dati tra la memoria ed i dispositivi di I/O, i bus di I/O sono interfacciati con il bus processore-memoria attraverso un adattatore di bus. Tale adattatore permette uno scambio di dati veloce tra i bus. In molti PC recenti il bus processore-memoria è un bus PCI (un bus di backplane) che permette di avere sia dei dispositivi di I/O che si interfacciano direttamente sul bus, sia un bus di I/O (il bus SCSI) che si connette al bus PCI. (c) Si usa un bus a parte per il traffico processore-memoria. Un numero limitato di bus di backplane si collegano al bus processore-memoria; i bus processore-memoria si interfacciano con il bus di backplane. Questa operazione viene normalmente svolta da un controllore integrato su un chip apposito, ad esempio un controllore di bus SCSI. Un vantaggio di tale organizzazione è il numero limitato di connessioni nel bus ad alta velocità tra processore e memoria.

sono d'accordo. Il protocollo è implementato con un insieme aggiuntivo di linee di controllo.

Un semplice esempio illustra come funzionino i bus asincroni: si consideri un dispositivo che richiede una parola di dato al sistema di memoria e si assuma che esistano tre linee di controllo:

1. *ReadReq*: utilizzata per indicare una richiesta di lettura da memoria. L'indirizzo viene contemporaneamente messo sulle linee di dato.
2. *DataRdy*: utilizzata per indicare che la parola di dato è pronta sulle linee di dato. In una transazione di output la memoria attiva questo segnale nel momento in cui fornisce il dato. In una transazione di input un dispositivo di I/O attiva questo segnale nel momento in cui vuole fornire un dato. In ambedue i casi il dato è messo contemporaneamente sulle linee di dato.
3. *Ack*: utilizzata per segnalare la ricezione dei segnali *ReadReq* e *DataRdy*.

In un protocollo asincrono i segnali di controllo ReadReq e DataRdy restano attivi sino a che l'altra parte (la memoria o il dispositivo) non indica che il valore sulle linee di controllo è stato ricevuto e che le linee di dato sono state lette; questa informazione viene trasmessa attivando la linea Ack. Questo processo nel suo complesso viene chiamato *handshaking*. La figura 8.10 mostra come opera un tale protocollo illustrando i passi attraverso i quali si svolge la comunicazione.



Il ciclo di passi previsto dal protocollo inizia non appena il dispositivo avanza una richiesta attivando il segnale ReadReq e mettendo l'indirizzo sulle linee Data:

1. Quando la memoria rileva il nuovo valore sulla linea ReadReq, legge l'indirizzo dal bus dati ed attiva il segnale Ack per indicare che ha rilevato il cambiamento.
2. Il dispositivo di I/O rileva che la linea Ack è attiva e disattiva il segnale ReadReq e le linee di dato.
3. La memoria rileva che la linea ReadReq è bassa e disattiva la linea Ack per segnalare il rilevamento del cambiamento su ReadReq.
4. Questo passo viene iniziato allorché la memoria ha i dati pronti: si mettono i dati richiesti per la lettura sulle linee di dato e si attiva DataRdy.
5. Il dispositivo di I/O rileva il cambiamento su DataRdy, legge i dati dal bus e segnala che ha acquisito i dati attivando Ack.
6. La memoria vede il segnale Ack, disattiva DataRdy e rilascia le linee di dato.
7. Infine il dispositivo di I/O, vedendo che DataRdy ritorna basso, disattiva la linea Ack, indicando che la trasmissione è stata completata.

A questo punto si può iniziare una nuova transazione sul bus.

Figura 8.10 Il protocollo asincrono di handshaking consiste di sette passi per la lettura di una parola dalla memoria e la sua ricezione da parte di un dispositivo di I/O. I segnali in colore contrastante sono quelli attivati dal dispositivo di I/O, mentre la memoria attiva i segnali che appaiono in nero. Le frecce specificano i sette passi e l'evento che scatena ogni passo. Il simbolo che rappresenta due linee (una in alto ed una in basso) nello stesso istante sulle linee di dato indica che le linee di dato contengono a quel punto un dato valido, ma che il suo valore non è noto.

Il funzionamento di un protocollo di bus asincrono è analogo a quello di una coppia di macchine a stati finiti che comunicano in maniera tale che una macchina non procede sino a quando non sa che l'altra macchina ha raggiunto un certo stato; in questa maniera le due macchine lavorano in modo coordinato.

Il protocollo di handshaking non risolve tutti i problemi legati alla comunicazione tra un dispositivo che invia ed un altro che riceve che possiedono clock diversi. Un problema ulteriore si verifica quando si campiona un segnale asincrono, quale ReadReq: tale problema, denominato *errore di sincronizzazione*, può portare ad un comportamento imprevedibile, e può essere superato grazie a dispositivi chiamati *sincronizzatori*, che saranno descritti nell'Appendice B.

Macchina a stati finiti per il controllo dell'I/O

Esempio Si mostri come il controllo di una transazione di output per il trasferimento dati da un dispositivo di I/O alla memoria (come nella figura 8.7) possa essere implementato come una coppia di macchine a stati finiti.

Soluzione La figura 8.11 mostra i due controllori basati su macchine a stati finiti che implementano il protocollo di handshaking della figura 8.10.

Se è possibile utilizzarlo, un bus sincrono è normalmente più veloce di un bus asincrono a causa del costo aggiuntivo introdotto dal meccanismo di handshaking. Il seguente esempio illustra questo punto.

Analisi delle prestazioni di un bus sincrono rispetto ad un bus asincrono

Esempio Si desidera confrontare la massima ampiezza di banda fornita da un bus sincrono con quella di uno asincrono. Il bus sincrono ha un periodo di clock di 50 ns, e ciascuna trasmissione sul bus richiede un ciclo di clock; il bus asincrono richiede 40 ns per l'handshaking. La parte relativa ai dati dei due bus ha ampiezza pari a 32 bit. Si trovi l'ampiezza di banda di ciascun bus quando questo esegue la lettura di una parola da una memoria da 200 ns.

Soluzione Innanzitutto si consideri il bus sincrono, che possiede dei cicli di bus di lunghezza 50 ns. I passi e i tempi richiesti dal bus sincrono sono i seguenti:

1. invio dell'indirizzo alla memoria: 50 ns
2. lettura da memoria: 200 ns
3. invio del dato al dispositivo: 50 ns.

In conclusione il tempo totale è pari a 300 ns; ne consegue un'ampiezza di banda massima del bus corrispondente a 4 byte ogni 300 ns, ossia

$$\frac{4 \text{ byte}}{300 \text{ ns}} = \frac{4 \text{ MB}}{0,3 \text{ secondi}} = 13,3 \frac{\text{MB}}{\text{secondo}}$$

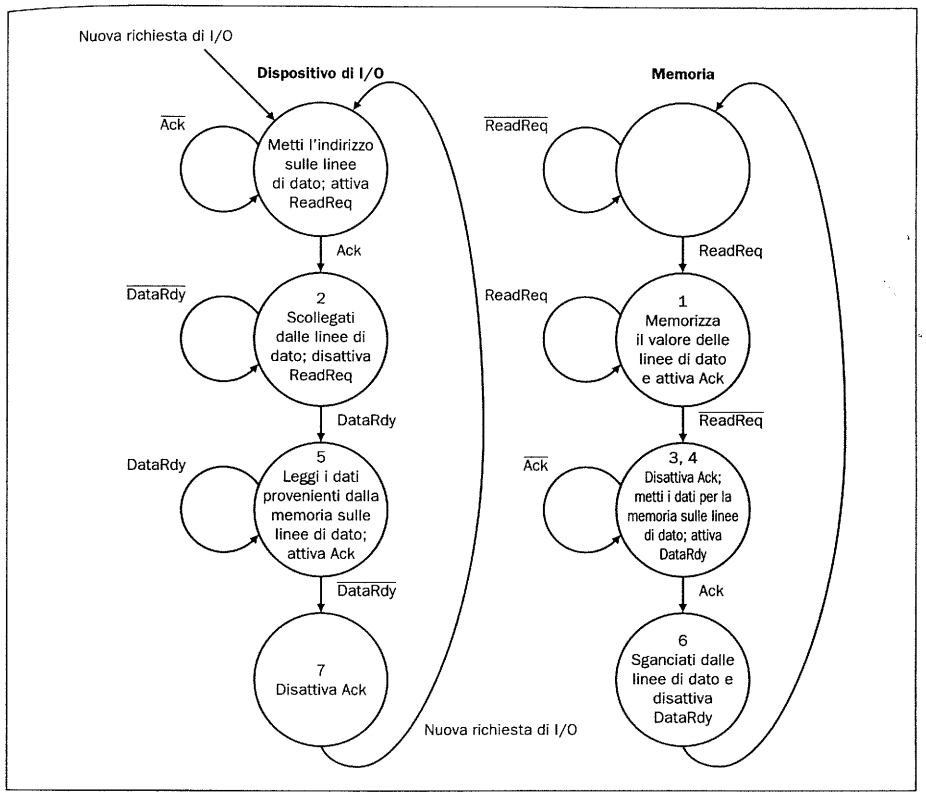


Figura 8.11 Le macchine a stati finiti che implementano il controllo del protocollo di handshaking riportato in figura 8.10. I numeri che compaiono in ciascuno stato corrispondono ai passi della figura 8.10. Il primo stato del dispositivo di I/O (in alto a sinistra) attiva il protocollo allorquando viene generata una nuova richiesta di I/O, esattamente come nella figura 8.10. Ciascuno stato nella macchina a stati finiti memorizza lo stato sia del dispositivo che della memoria, che in questo modo restano sincronizzati durante la transazione. Dopo il completamento della transazione la parte del dispositivo di I/O può rimanere nell'ultimo stato sino a che non giunge una nuova richiesta da soddisfare.

A prima vista potrebbe sembrare che il bus asincrono sia molto più lento poiché richiede sette passi, ognuno dei quali lungo almeno 40 ns, oltre al passo corrispondente all'accesso in memoria che richiede 200 ns. Se però si osserva con attenzione la figura 8.10 ci si accorge che alcuni dei passi possono essere contemporanei all'accesso alla memoria. In particolare la memoria riceve l'indirizzo alla fine del passo 1 e non è necessario che metta i dati sul bus fino alla fine del passo 5; i passi 2, 3 e 4 possono quindi essere sovrapposti al tempo necessario per l'accesso in memoria. Questo porta ai seguenti parametri di tempo:

- Passo 1: 40 ns
- Passi 2, 3 e 4: valore massimo tra 3×40 ns e 200 ns = 200 ns
- Passi 5, 6 e 7: 3×40 ns = 120 ns

Il tempo totale per eseguire il trasferimento è quindi 360 ns, e la massima ampiezza di banda corrisponde a

$$\frac{4 \text{ byte}}{360 \text{ ns}} = \frac{4 \text{ MB}}{0,36 \text{ secondi}} = 11,1 \frac{\text{MB}}{\text{secondo}}$$

Ne consegue che il bus sincrono è più veloce di un fattore pari a circa il 20% soltanto. Ovviamente queste velocità possono essere ottenute solo se il dispositivo e il sotto-sistema di memoria connessi al bus asincrono sono sufficientemente veloci da eseguire ciascun passo del meccanismo di handshaking in 40 ns.

Anche se un bus sincrono può essere più veloce, la scelta tra un bus sincrono ed uno asincrono ha delle implicazioni che riguardano non solo la banda dati, ma anche la capacità del sistema di I/O in termini di distanza fisica e numero di dispositivi che possono essere collegati al bus. I bus asincroni sono più facilmente scalabili con la tecnologia e possono supportare una più ampia varietà di dispositivi, anche dotati di diverse velocità di risposta. È per queste ragioni che i bus di I/O sono spesso asincroni malgrado il costo aggiuntivo che ne consegue.

Come si aumenta l'ampiezza di banda di un bus

Benché una buona parte dell'ampiezza di banda di un bus sia determinata una volta che si è scelto un protocollo sincrono o asincrono e sono stati stabiliti i parametri temporali caratteristici del bus, esistono alcuni altri fattori che influenzano l'ampiezza di banda ottenibile durante una singola operazione di trasferimento. I più importanti tra questi fattori sono:

1. *Ampiezza del bus dati*: aumentando l'ampiezza del bus dati il trasferimento di più parole richiede un numero minore di cicli di clock.
2. *Linee di indirizzo e di dato distinte oppure condivise*: nell'esempio della figura 8.8 venivano utilizzati gli stessi fili per gli indirizzi e per i dati; se si introducono delle linee distinte per gli indirizzi le prestazioni dell'operazione di scrittura saranno migliori, dal momento che gli indirizzi e i dati potranno essere trasferiti in un singolo ciclo di bus.
3. *Trasferimento a blocchi*: se si permette al bus di trasferire più parole in cicli di bus consecutivi senza inviare un indirizzo o rilasciare il bus, si riduce il tempo necessario per il trasferimento di un blocco di grandi dimensioni.

Ciascuna delle soluzioni di progetto riportate è in grado di migliorare le prestazioni del bus durante una singola operazione di trasferimento. Il costo per l'implementazione di questi miglioramenti corrisponde ad uno o più dei seguenti punti: un numero maggiore di linee di bus, una complessità superiore, un tempo di risposta più alto per le richieste che devono aspettare mentre viene eseguito il trasferimento di un blocco di grandi dimensioni.

Esempio

Analisi delle prestazioni di due schemi di bus

Si supponga di avere un sistema con le seguenti caratteristiche:

1. Un sistema composto da memoria e bus in grado di eseguire l'accesso a blocchi di dimensione variabile tra 4 e 16 parole di 32 bit.

2. Un bus sincrono di 64 bit con clock a 200 MHz, in cui i trasferimenti su 64 bit richiedono ciascuno un ciclo di clock, oltre ad un ciclo di clock per l'invio di un indirizzo alla memoria.
3. Due cicli di clock tra due operazioni successive sul bus. Si assuma che il bus sia inattivo prima dell'inizio di un accesso.
4. Un tempo di accesso alla memoria per le prime quattro parole pari a 200 ns; ciascun ulteriore gruppo di quattro parole può essere letto in 20 ns. Si assuma che possano essere sovrapposti il trasferimento su bus del dato letto più di recente e la lettura delle successive quattro parole.

Si calcoli l'ampiezza di banda a regime e la latenza per una lettura di 256 parole ipotizzando dei trasferimenti che utilizzano blocchi di 4 parole e trasferimenti che utilizzano blocchi di 16 parole. Si calcoli inoltre il reale numero di transazioni sul bus per secondo in ciascun caso. Si ricordi che una singola transazione sul bus consiste nella trasmissione di un indirizzo seguita dai dati.

Soluzione

Per il trasferimento di un blocco di quattro parole è necessario:

1. Un ciclo di clock per inviare l'indirizzo alla memoria
2. $\frac{200 \text{ ns}}{5 \text{ ns/ciclo}} = 40$ cicli di clock per la lettura da memoria
3. Due cicli di clock per inviare i dati alla memoria
4. Due cicli di clock vuoti tra un trasferimento ed il successivo.

Tutto questo corrisponde ad un totale di 45 cicli e 256/4 = 64 transazioni, cosicché l'intero trasferimento richiede 45 × 64 = 2880 cicli di clock. La latenza è quindi pari a 2880 cicli × 5 ns/ciclo = 14400 ns. Il numero di transazioni su bus per secondo è

$$64 \text{ transazioni} \times \frac{1 \text{ secondo}}{14400 \text{ ns}} = 4,44 \text{ Mtransazioni/secondo}$$

L'ampiezza di banda del bus è

$$(256 \times 4) \text{ byte} \times \frac{1 \text{ secondo}}{14400 \text{ ns}} = 71,11 \text{ MB/s}$$

Per il trasferimento di un blocco di 16 parole il primo blocco richiede:

1. Un ciclo di clock per inviare un indirizzo alla memoria
2. 200 ns (o 40 cicli) per leggere le prime 4 parole in memoria
3. Due cicli per inviare i dati del blocco; contemporaneamente viene iniziata la lettura delle 4 parole del blocco successivo
4. Due cicli vuoti per separare due trasferimenti, durante i quali viene completata la lettura del blocco successivo.

Ciascuno dei restanti tre blocchi di quattro parole richiede la ripetizione degli ultimi due passi soltanto.
Il numero totale di cicli per ciascun blocco di 16 parole è quindi 1 + 40 + 4 × (2 + 2) = 57 cicli, e sono necessarie 256/16 = 16 transazioni; l'intero trasferimento

richiede 57 × 16 = 912 cicli. La latenza è quindi pari a 912 cicli × 5 ns/ciclo = 4560 ns, che corrisponde a circa un terzo della latenza trovata nel caso di blocchi di 4 parole. Il numero di transazioni su bus per secondo con blocchi di 16 parole è

$$16 \text{ transazioni} \times \frac{1 \text{ secondo}}{4560 \text{ ns}} = 3{,}51 \text{ Mtransazioni/secondo}$$

che è inferiore al caso di blocchi di 4 parole, dal momento che ciascuna transazione richiede un tempo maggiore (57 cicli anziché 45).
L'ampiezza di banda del bus con blocchi di 16 parole è

$$(256 \times 4) \text{ byte} \times \frac{1 \text{ secondo}}{4560 \text{ ns}} = 224{,}56 \text{ MB/s}$$

che è 3,16 volte maggiore del valore trovato per blocchi di 4 parole. È chiaro il vantaggio derivante dall'uso di trasferimenti che coinvolgono blocchi più grandi.

Approfondimento: Un altro metodo per aumentare l'ampiezza della banda efficace del bus quando più dispositivi vogliono comunicare consiste nel rilasciare il bus quando non è utilizzato per la trasmissione di informazioni. Si consideri ad esempio il caso della lettura da memoria esaminato nella figura 8.10. Cosa succede sul bus quando è in corso l'accesso alla memoria? Nella versione semplice del protocollo il dispositivo e la memoria continuano ad occupare il bus durante il tempo di accesso alla memoria, anche se in effetti non ha luogo alcun trasferimento. Un protocollo alternativo, che rilascia il bus, potrebbe funzionare come segue:

1. Il dispositivo invia un segnale alla memoria e trasmette la richiesta e l'indirizzo.
2. Dopo che la memoria ha segnalato la ricezione della richiesta, sia la memoria che il dispositivo rilasciano tutte le linee di controllo
3. Ha luogo l'accesso alla memoria, ed il bus è disponibile per altri usi durante il periodo corrispondente.
4. La memoria invia un segnale sul bus al dispositivo per indicare che i dati sono disponibili.
5. Il dispositivo riceve i dati attraverso il bus e segnala che ha eseguito l'operazione, così che il sistema di memoria può rilasciare il bus.

Nel caso del bus sincrono che esegue trasferimenti a blocchi di 16 parole visto nell'esempio precedente, un tale schema occuperebbe il bus per soli 272 dei 912 cicli richiesti dalla transazione completa su bus.

Un protocollo di questo tipo viene chiamato *protocollo a transazione divisa* (split transaction protocol). Il vantaggio di un simile protocollo sta nel fatto che liberando il bus nel periodo in cui non vengono trasmessi dati, il protocollo permette ad un altro dispositivo di utilizzare il bus. Questo accorgimento può migliorare l'ampiezza di banda effettiva del bus dell'intero sistema, a patto che la memoria sia sufficientemente sofisticata da gestire delle transazioni tra loro sovrapposte.

Se si considera una transazione divisa, il tempo per completare un trasferimento risulta con ogni probabilità maggiore, dal momento che il bus deve essere acquisito due volte. I protocolli a transazione divisa sono inoltre più costosi da implementare, principalmente per la necessità di tener traccia dell'altro dispositivo che comunica durante la transazione. Nel protocollo a transazione divisa il sistema di memoria deve infatti contattare il dispositivo che aveva fatto la richiesta per iniziare la parte di risposta della transazione sul bus, e l'identità del richiedente deve essere quindi trasmessa e memorizzata dal sistema di memoria.

Come ottenere l'accesso al bus

Ora che si sono riviste alcune delle molte possibilità di progetto per i bus, ci si può concentrare su uno dei punti più importanti: il dispositivo che desidera utilizzare il bus per la comunicazione, come può ottenerlo? Si è già fatto cenno a questa questione in diversi punti nel corso delle precedenti discussioni: essa è cruciale allorquando si progettano sistemi di I/O di grandi dimensioni che permettono alle operazioni di I/O di aver luogo senza un coinvolgimento continuo e a basso livello del processore.

Perché è necessario uno schema per controllare l'accesso al bus? Se non vi fosse alcun controllo, dispositivi diversi che volessero comunicare potrebbero ciascuno cercare di forzare sulle linee di controllo e di dato i valori corrispondenti a trasferimenti diversi! Così come in un'aula di scuola regna il caos quando tutti gli studenti cercano di parlare contemporaneamente, così se più dispositivi usassero il bus simultaneamente si otterrebbe una confusione totale.

Per evitare il caos si introducono nel sistema uno o più *bus master* (letteralmente *padroni del bus*). Un bus master controlla l'accesso al bus: è questo dispositivo che dà l'autorizzazione e controlla tutte le richieste al bus. Il processore deve essere in grado di attivare una richiesta di bus per accedere alla memoria, ed è quindi sempre un bus master. La memoria è normalmente uno *slave* (letteralmente schiavo), dal momento che risponde alle richieste di lettura e scrittura ma non genera mai proprie richieste.

Il sistema più semplice tra quelli possibili possiede un singolo bus master, coincidente con il processore. Quando vi è un unico bus master si è in una situazione simile a quella di una classe, in cui tutte le comunicazioni tra gli studenti richiedono un'autorizzazione da parte dell'insegnante: in un sistema a master singolo tutte le richieste di bus devono essere controllate dal processore. I passi che costituiscono una transazione sul bus nel caso di bus a master singolo sono mostrati nella figura 8.12. Il principale svantaggio di un simile approccio risiede nel fatto che il processore deve essere coinvolto in ciascuna transazione sul bus. La lettura da disco di un solo settore può richiedere il coinvolgimento del processore per centinaia di migliaia di volte, a seconda della dimensione di ciascun trasferimento. Poiché i dispositivi sono divenuti più veloci e sono ora capaci di eseguire trasferimenti con ampiezza di banda molto più elevata, il coinvolgimento del processore in ciascuna transazione sul bus è divenuto via via sempre meno accettabile.

Lo schema alternativo a questo consiste nell'avere più master del bus, ognuno dei quali in grado di iniziare un trasferimento. Se si vuole permettere a più persone in una classe di parlare senza che l'insegnante debba dare a ciascuno l'autorizzazione, si deve avere un protocollo per decidere chi dovrà parlare al prossimo turno. Analogamente quando si hanno più master del bus deve esistere un meccanismo per arbitrare l'accesso al bus così che questo avvenga in modo coordinato anziché caotico.

L'arbitraggio del bus

L'operazione di decidere quale master utilizzerà il bus al prossimo ciclo di bus viene chiamata *arbitraggio del bus*. Esiste un'ampia varietà di schemi di arbitraggio del bus: essi possono richiedere hardware specifico o protocolli di bus estremamente sofisticati. In un qualunque schema di arbitraggio, quando un dispositivo (o il processore) vuole usare il bus invia un segnale di *richiesta del bus* (bus request) e riceve poi un segnale di *assegnazione del bus* (bus grant). Dopo aver ricevuto il segnale di assegnazione, il dispositivo può utilizzare il bus, segnalando poi all'arbi-

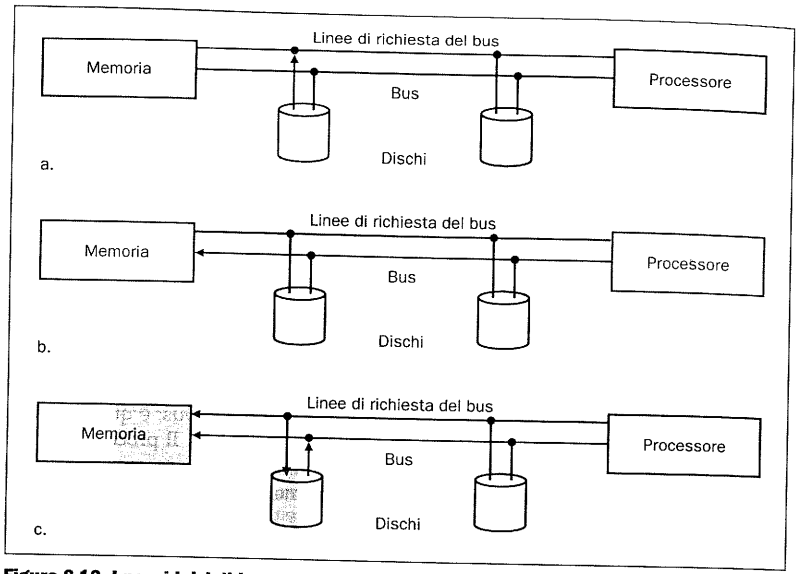


Figura 8.12 I passi iniziali in una transazione sul bus nel caso in cui il master sia unico, e coincidente con il processore. Il dispositivo usa un insieme di linee di richiesta del bus per comunicare con il processore, che inizia poi il ciclo di bus per conto del dispositivo richiedente. Le linee e le unità attive in ciascun passo sono mostrate in colore contrastante. L'ombreggiatura è utilizzata per indicare la sorgente di una lettura (la memoria) o la destinazione di una scrittura (un disco). Dopo il passo c, il ciclo di bus continua come una normale transazione di lettura, come nella figura 8.7. (a) Innanzitutto il dispositivo genera una richiesta di bus per segnalare al processore che desidera utilizzare il bus. (b) Il processore risponde e genera degli appropriati segnali di controllo del bus. Ad esempio, se il dispositivo desidera eseguire l'output da memoria, il processore attiva le linee di richiesta della lettura verso la memoria. (c) Il processore informa inoltre il dispositivo che la sua richiesta di bus è in corso di elaborazione; come risultato il dispositivo sa di poter usare il bus e mette sul bus gli indirizzi relativi alla richiesta.

tro quando il bus non gli è più necessario; l'arbitro può quindi assegnare il bus ad un altro dispositivo. La maggior parte dei bus a master multiplo possiedono un insieme di linee di bus per eseguire le richieste e le assegnazioni. È inoltre necessaria una linea di rilascio del bus, a meno che ciascun dispositivo non abbia la propria linea di richiesta. A volte i segnali utilizzati per l'arbitraggio del bus corrispondono a linee fisicamente separate, mentre in altri sistemi si usano a questo scopo le linee di dato del bus (anche se questo impedisce di eseguire insieme le fasi di arbitraggio e di trasferimento).

Quando devono scegliere a quale dispositivo assegnare il bus, gli schemi di arbitraggio tentano normalmente di trovare un compromesso tra due fattori. Innanzitutto ogni dispositivo possiede una *priorità* per quanto riguarda il bus, ed il dispositivo con la priorità più elevata deve essere servito per primo. D'altro canto si vorrebbe che tutti i dispositivi, anche quelli con bassa priorità, non fossero mai completamente esclusi dal bus: questa proprietà, chiamata *fairness* (letteralmente equità) garantisce che ciascun dispositivo che vuole usare il bus prima o poi lo otterrà. In aggiunta a questi fattori, gli schemi più sofisticati mirano a ridurre il tempo necessario per eseguire l'arbitraggio sul bus. Dal momento che il tempo per

l'arbitraggio rappresenta un costo addizionale, aumentando il tempo di accesso al bus, esso dovrebbe essere per quanto possibile ridotto e sovrapposto con il tempo per il trasferimento su bus.

Gli schemi di arbitraggio del bus possono essere suddivisi in quattro grandi categorie:

- *Arbitraggio in daisy chain* (in catena): in questo schema, la linea di assegnazione del bus passa da un dispositivo all'altro a partire da quello con priorità più alta (la priorità è determinata dalla posizione sul bus). Un dispositivo con priorità elevata che desidera fare accesso al bus intercetta semplicemente il segnale di assegnazione del bus, impedendo ai dispositivi con priorità inferiore di ricevere il segnale. La figura 8.13 mostra l'organizzazione di un bus in daisy chain. Il vantaggio di un bus con daisy chain è la semplicità; gli svantaggi stanno nel fatto che non si può garantire la fairness (una richiesta a bassa priorità può restare bloccata indefinitamente) e nel fatto che la propagazione del segnale di assegnazione in daisy chain limita la velocità del bus.
- *Arbitraggio centralizzato parallelo*: gli schemi appartenenti a questa categoria usano linee multiple di richiesta, ed i dispositivi avanzano indipendentemente le loro richieste di bus. Un arbitro centralizzato sceglie uno tra i dispositivi che hanno avanzato richiesta di accesso al bus e gli comunica che a quel punto diventa master del bus. Lo svantaggio di questo schema sta nel fatto che richiede un arbitro centralizzato, che può divenire il collo di bottiglia nell'uso del bus. Il bus PCI, che rappresenta uno standard nell'ambito dei bus di backplane, utilizza uno schema di arbitraggio centralizzato.
- *Arbitraggio distribuito con auto-selezione*: questi schemi utilizzano anch'essi delle linee di richiesta multiple, ma i dispositivi che richiedono l'accesso al bus determinano essi stessi chi riceverà l'assegnazione del bus. Ciascun dispositivo che vuole accedere al bus scrive sul bus un codice che lo identifica. Dall'esame del bus, i dispositivi possono determinare qual è il dispositivo con priorità più elevata tra quelli che hanno fatto richiesta. Non vi è necessità di un arbitro centralizzato: ciascun dispositivo determina indipendentemente se è il richiedente con più alta priorità. Questo schema non richiede comunque un numero più elevato di linee di richiesta. Il NuBus, che costituisce il bus di backplane per i Macintosh II di Apple, utilizza questo schema.

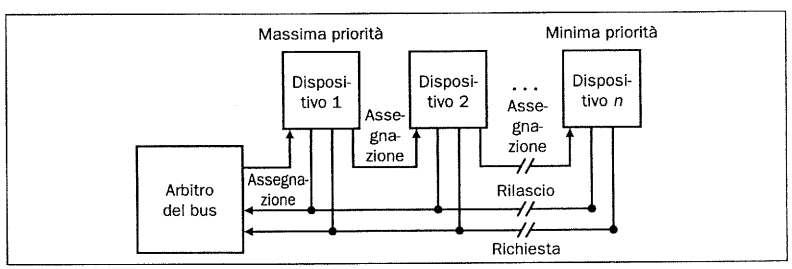


Figura 8.13 Un bus con daisy chain usa una linea di assegnazione del bus che attraversa i vari dispositivi a partire da quelli con più elevata priorità e proseguendo con quelli a minore priorità. Se il dispositivo ha richiesto l'accesso al bus, utilizza la linea di assegnazione per sapere se l'accesso gli è stato accordato. Dal momento che la linea di assegnazione viene passata oltre solo se un dispositivo non desiderava fare accesso, la priorità è implicita nello schema. I dettagli del protocollo usato della tecnica daisy chain sono descritti nel seguente approfondimento.

- *Arbitraggio distribuito con rilevamento delle collisioni*: in questo schema ciascun dispositivo avanza indipendentemente la sua richiesta di bus. Richieste multiple simultanee provocano una *collisione*. La collisione viene rilevata e si usa uno schema per la selezione di un dispositivo tra quelli che hanno causato la collisione. Il protocollo della rete Ethernet, che usa questo schema, è descritto in maggiore dettaglio nell'Esercizio 8.28.

La scelta dello schema di arbitraggio più adatto è determinata da una varietà di fattori, tra cui il livello di espandibilità che deve essere garantito dal bus in termini di numero di dispositivi di I/O e di lunghezza del bus, la velocità richiesta per l'arbitraggio, ed il grado di fairness richiesto.

Approfondimento: il protocollo seguito da un dispositivo in un bus di tipo daisy chain è il seguente:

1. Attivazione della linea di richiesta.
2. Attesa di una transizione da basso ad alto sulla linea di assegnazione, che indica che il bus sta per essere riassegnato.
3. Rilevamento del segnale di assegnazione senza permettere ai dispositivi con priorità inferiore di far altrettanto. Fine dell'attivazione della linea di richiesta.
4. Uso del bus.
5. Segnalazione che il bus non è più necessario attraverso l'attivazione della linea di rilascio.

Rendendo il dispositivo sensibile ad una transizione sulla linea di assegnazione piuttosto che ad un livello, si evita che il dispositivo tolga il bus ad un dispositivo con priorità inferiore quando questo già crede di aver ottenuto l'accesso al bus. Per aumentare la fairness in uno schema con daisy chain, si può semplicemente imporre la regola che un dispositivo che abbia appena utilizzato il bus non possa riacquisire il bus sino a che non vede abbassarsi la linea di richiesta del bus. Poiché un dispositivo non disattiva la linea di richiesta sino a che la sua richiesta non è stata soddisfatta, tutti i dispositivi hanno l'opportunità di utilizzare il bus prima che un qualsiasi dispositivo lo abbia utilizzato due volte. Alcuni sistemi di bus (ad esempio il VME) utilizzano catene di daisy chain multiple con un insieme separato di linee di richiesta e di assegnazione per ciascuna catena, ed un codificatore a priorità per eseguire la scelta tra le richieste multiple.

Quadro d'insieme

Le caratteristiche dei diversi bus permettono la creazione di bus ottimizzati per un'ampia gamma di dispositivi diversi, per numeri diversi di dispositivi e per diverse esigenze in termini di ampiezza di banda. La figura 8.14 mostra alcune delle alternative di progetto che sono state discusse e le scelte che possono essere fatte in sistemi di basso costo o ad alte prestazioni. In generale i sistemi ad alto costo usano bus più ampi e più veloci con protocolli più sofisticati (tipicamente un bus sincrono per le ragioni che si sono viste in precedenza). Al contrario un sistema di basso costo tende a privilegiare un bus a più basso parallelismo e che non richiede dispositivi dotati di particolare intelligenza (di conseguenza utilizzerà un master singolo); il bus sarà di tipo asincrono permettendo un costo relativamente basso nell'interfacciamento dei dispositivi a bassa velocità.

Caratteristica	Alte prestazioni	Basso costo
Ampiezza del bus	linee di indirizzo e di dato separate	linee di indirizzo e di dato condivise
Ampiezza dei dati	un bus più ampio è più veloce (ad esempio 32 bit)	un bus più stretto è meno costoso (ad esempio 8 bit)
Dimensione del trasferimento	parole multiple richiedono un ritardo minore per il bus	il trasferimento di singole parole è più semplice
Master del bus	master multipli (richiede arbitraggio)	master singolo (no arbitraggio)
Temporizzazione	sincrono	asincrono

Figura 8.14 Le caratteristiche dei bus di I/O determinano le prestazioni dei trasferimenti di I/O, il numero dei dispositivi di I/O che possono essere connessi ed il costo dei dispositivi di connessione. I bus più corti sono in genere più veloci, ma non possono essere estesi. Analogamente i bus più ampi possono avere ampiezza di banda maggiore, ma sono più costosi. I bus a transazioni divise costituiscono un'altra possibile strada per aumentare l'ampiezza di banda aumentandone il costo (si veda l'approfondimento relativo).

Standard di bus

La maggior parte dei calcolatori permette agli utenti di ampliare le funzioni dell'elaboratore aggiungendo nuove periferiche. Il bus di I/O serve normalmente per raggiungere tale scopo garantendo il collegamento con nuove periferiche. Per rendere questa operazione più semplice l'industria informatica ha sviluppato diversi standard di bus: gli standard vengono utilizzati come specifiche sia dai costruttori di calcolatori che dai costruttori di periferiche. Uno standard di bus garantisce al progettista di calcolatori che per il suo nuovo calcolatore saranno disponibili delle periferiche, e garantisce al costruttore di periferiche che gli utenti saranno in grado di collegare le loro nuove attrezzature.

Gli elaboratori a volte divengono così diffusi che i loro bus di I/O diventano degli standard de facto, come è avvenuto per il bus dell'IBM PC-AT. Una volta che uno standard di bus viene pesantemente usato dai progettisti di periferiche, altri costruttori di calcolatori incorporano tale bus nei loro prodotti, che possono così fruire di un'ampia gamma di periferiche. A volte gli standard sono creati da gruppi di persone che tentano di risolvere uno stesso problema: il bus SCSI e Ethernet sono degli esempi di standard nati dalla cooperazione tra produttori. Anche enti ufficiali come l'ANSI e l'IEEE creano ed approvano degli standard. Lo standard PCI fu inizialmente definito da Intel e poi sviluppato da un comitato di aziende.

La figura 8.15 riassume le caratteristiche fondamentali dei due standard dominanti nel campo dei bus: PCI (un bus di backplane di uso generale) e SCSI (un bus di I/O). Un bus SCSI è normalmente interfacciato con un bus di backplane o con un bus processore-memoria. Un controllore SCSI coordina il trasferimento di dati da un dispositivo che si trova sul bus di I/O verso la memoria tramite il bus processore-memoria. Uno standard di bus emergente è Fibre Channel, proposto come un derivato dello SCSI e basato su canali punto-punto ad alta velocità, che può essere organizzato ad anello per dispositivi multipli.

In generale, l'ampiezza di banda del bus non è riconducibile ad un singolo numero: a causa dei tempi aggiuntivi introdotti dal bus la dimensione del trasferimento influisce in maniera significativa sull'ampiezza della banda. Poiché il bus normalmente esegue trasferimenti da o per la memoria, anche la velocità della memoria influisce sull'ampiezza della banda.

I bus forniscono una continuità elettrica tra i dispositivi di I/O, i processori e la memoria, e definiscono anche il protocollo di più basso livello per le comunicazio-

Caratteristica	PCI	SCSI
Tipo di bus	backplane	I/O
Ampiezza base del bus dati (numero di segnali)	32-64	8-32
Indirizzi/dati condivisi?	condivisi	condivisi
Numero di master del bus	molti	molti
Arbitraggio	centralizzato, arbitraggio parallelo	autoselezione
Temporizzazione	sincrono 33-66 MHz	asincrono o sincrono (5-10 MHz)
Ampiezza di banda di picco teorica	133-512 MB/s	5-40 MB/s
Ampiezza di banda stimata raggiungibile per bus base	80 MB/s	2,5-40 MB/s (sincrono) o 1,5 MB/s (asincrono)
Massimo numero di dispositivi	1024 (con segmenti di bus multipli; al più 32 dispositivi per segmento di bus)	7-31 (ampiezza del bus – 1)
Massima lunghezza del bus	0,5 metri	25 metri
Nome dello standard	PCI	ANSI X3.131

Figura 8.15 Caratteristiche chiave di due standard di bus dominanti. Sia lo standard di bus PCI che lo standard SCSI sono stati estesi in maniera significativa. PCI ha una versione con ampiezza doppia (64 bit rispetto a 32) ed una versione veloce (66 MHz rispetto a 33). Il bus SCSI originale era asincrono. Versioni sincrone più veloci sono state poi sviluppate, seguite da estensioni per un bus più ampio (16 e 32 bit rispetto ad 8; queste versioni prendono il nome di *wide* SCSI) e un clock più veloce (10 MHz nella versione *fast* SCSI, rispetto ai 5 MHz del bus SCSI sincrono originale). Il bus fast wide SCSI combina una più alta frequenza di clock con una maggiore ampiezza. Inoltre una versione a 20 MHz del bus SCSI (chiamata Ultra) è stata sviluppata e rilasciata alla fine del 1996. Le specifiche per questi bus standard sono divenute estremamente complesse: ad esempio il documento che descrive lo standard PCI è lungo 282 pagine, mentre le specifiche dello SCSI-2, che include sia la versione più veloce che quella con maggiore ampiezza, sono lunghe oltre 600 pagine! Le specifiche per lo SCSI-2, una buona panoramica dello SCSI e del suo sviluppo, e la specifica del bus PCI sono disponibili in rete all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*.

ni. Al di sotto di questo livello base è necessario definire dei protocolli hardware e software per il controllo dei trasferimenti di dati tra i dispositivi di I/O e la memoria; dal punto di vista del processore è inoltre necessario definire dei comandi verso i dispositivi di I/O. Tali argomenti saranno trattati nel prossimo paragrafo.

8.5 L'interfacciamento dei dispositivi di I/O con la memoria, il processore ed il sistema operativo

Un protocollo di bus definisce come una parola o un blocco di dati debbano essere trasferiti su un insieme di fili. Questo non risolve tuttavia il problema di varie altre attività che devono essere eseguite per far sì che i dati siano realmente trasferiti da un dispositivo allo spazio di memoria indirizzabile da un qualche programma utente. Questo paragrafo è dedicato a queste attività e fornirà una risposta a problemi quali:

- Come può una richiesta di I/O avanzata da un utente essere trasformata in un comando per il dispositivo appropriato, ed essere poi comunicata al dispositivo?
- Come può il dato essere effettivamente trasferito da o verso una locazione di memoria?
- Qual è il ruolo del sistema operativo?

Come si vedrà dalle risposte a queste domande, il sistema operativo svolge un ruolo fondamentale nella gestione del sistema di I/O, fungendo da interfaccia tra l'hardware ed il programma che ha fatto la richiesta di I/O.

Il sistema operativo svolge compiti diversi in funzione di tre caratteristiche dei sistemi di I/O:

1. Il sistema di I/O è condiviso dai programmi che usano il processore.
2. I dispositivi di I/O spesso usano le interruzioni (ossia le eccezioni di origine esterna) per comunicare informazioni riguardanti le operazioni di I/O. Poiché le interruzioni causano un trasferimento del controllo al modo kernel o supervisore, esse debbono necessariamente essere gestite dal sistema operativo.
3. Il controllo di basso livello di un dispositivo di I/O è estremamente complesso, poiché richiede la gestione di un insieme di eventi concorrenti e poiché i vincoli posti da un corretto controllo del dispositivo sono spesso molto dettagliati.

Interfaccia Hardware Software

Le tre caratteristiche dei sistemi di I/O elencate sopra fanno sì che il sistema operativo debba fornire varie funzioni:

- Il sistema operativo garantisce che ciascun programma utente possa accedere soltanto a quelle parti di un dispositivo di I/O per le quali l'utente ha gli adeguati privilegi. Ad esempio il sistema operativo non deve permettere ad un programma di leggere o scrivere un file su disco se il proprietario del file non ne ha concesso l'accesso al programma stesso. In un sistema che prevede dei dispositivi di I/O condivisi, tale protezione non potrebbe essere fornita se i programmi utenti eseguissero direttamente le operazioni di I/O.
- Il sistema operativo fornisce un livello di astrazione più elevato nell'accesso ai dispositivi, mettendo a disposizione procedure specifiche in grado di gestire le operazioni di basso livello sui dispositivi.
- Il sistema operativo gestisce le interruzioni generate dai dispositivi di I/O, così come gestisce le eccezioni generate dai programmi.
- Il sistema operativo cerca di garantire in modo equo l'accesso alle risorse di I/O condivise e di ordinare gli accessi in maniera da ottimizzare il throughput del sistema.

Per svolgere queste funzioni a vantaggio dei programmi utente il sistema operativo deve essere in grado di comunicare con i dispositivi di I/O e di evitare che i programmi utente comunichino direttamente con essi. È necessario supportare tre tipi di comunicazioni:

1. Il sistema operativo deve essere in grado di inviare comandi ai dispositivi di I/O. Questi comandi includono non solo operazioni quali lettura e scrittura, ma anche altre operazioni che devono essere eseguite sul dispositivo, quali il posizionamento della testina su un disco.
2. Il dispositivo deve essere in grado di informare il sistema operativo quando il dispositivo di I/O ha completato un'operazione o ha incontrato un errore.

Per esempio quando il disco ha completato l'operazione di posizionamento della testina, lo segnala al sistema operativo.

3. I dati devono essere trasferiti tra la memoria ed il dispositivo di I/O. Ad esempio il blocco che si sta leggendo da un disco deve essere trasferito dal disco verso la memoria.

Nei prossimi paragrafi si vedrà come vengono eseguite queste comunicazioni.

Inviare comandi ai dispositivi di I/O

Per inviare un comando ad un dispositivo di I/O il processore deve essere in grado di indirizzare il dispositivo e di produrre una o più parole di comando. I metodi usati per indirizzare un dispositivo sono due: l'I/O mappato in memoria (*memory mapped I/O*) e le istruzioni speciali di I/O. Nell'I/O mappato in memoria alcune parti dello spazio di indirizzamento sono assegnate ai dispositivi di I/O, e le letture e scritture eseguite a questi indirizzi sono interpretate come comandi per il dispositivo di I/O.

Un'operazione di scrittura, ad esempio, può essere usata per inviare dati ad un dispositivo di I/O, che interpreterà il dato come un comando. Quando il processore mette l'indirizzo ed il dato sul bus, il sotto-sistema di memoria ignora l'operazione in quanto l'indirizzo specifica una parte dello spazio di indirizzamento usato per l'I/O. Il controllore del dispositivo invece si accorge dell'operazione, memorizza il dato e lo trasmette al dispositivo sotto forma di comando. I programmi utente non possono eseguire direttamente operazioni di I/O poiché il sistema operativo non permette loro l'accesso allo spazio di indirizzamento ad essi assegnato. Gli indirizzi riservati ai dispositivi di I/O sono protetti dal meccanismo stesso di traduzione degli indirizzi. L'I/O mappato in memoria può inoltre essere usato per trasmettere dati scrivendo o leggendo da indirizzi specificati. Il dispositivo usa l'indirizzo per determinare il tipo di comando, mentre il dato può venir fornito da un'operazione di scrittura o ottenuto attraverso un'operazione di lettura. In ogni caso l'indirizzo codifica sia l'identità del dispositivo sia il tipo di trasmissione tra il processore ed il dispositivo.

L'esecuzione effettiva di una lettura o scrittura per rispondere alla richiesta di un programma richiede normalmente una serie di operazioni di I/O separate; inoltre il processore può dover verificare lo stato del dispositivo tra un comando e l'altro per determinare se il comando è stato completato con successo. Ad esempio la stampante DEC LP11 ha due registri di interfaccia, uno per le informazioni di stato ed uno per i dati da stampare. Il registro di stato contiene un bit di completamento (*done bit*), che viene forzato a 1 dalla stampante quando ha terminato la stampa di un carattere, ed un bit di errore, che indica che la carta della stampante si è inceppata, oppure è finita. Ciascun byte di dato che deve venire stampato è messo nel registro di dato; il processore deve poi aspettare sino a che la stampante non forza a 1 il bit di completamento prima di poter mettere un altro carattere nel buffer. Il processore deve anche verificare il bit di errore per determinare se si è verificato un problema. Ciascuna di queste operazioni richiede l'accesso ad un dispositivo di I/O.

Approfondimento: L'alternativa all'I/O mappato in memoria consiste nell'utilizzare le istruzioni di I/O esistenti nel processore. Tali istruzioni permettono di specificare sia il numero del dispositivo che la parola di comando (o la posizione in memoria della parola di comando). Il processore comunica l'indirizzo del dispositivo attraverso una serie di fili che fa normalmente parte del bus di I/O; il comando effettivo può essere trasmesso sulle linee di dato del bus. Tra i calcolatori che possiedono delle istruzioni di I/O vi sono l'Intel 80x86 e l'IBM 370. Rendendo illegale l'esecuzione delle istruzioni di I/O allorquando il sistema non si trova nel modo kernel o supervisore è possibile impedire ai programmi utente di accedere direttamente ai dispositivi.

La comunicazione con il processore

Il processo di verifica periodica dei bit di stato per controllare se è giunto il momento della successiva operazione di I/O, visto nell'esempio precedente, è chiamato *interrogazione* o *polling* (letteralmente *sondaggio*). Il polling è il modo più semplice per un dispositivo di I/O di comunicare con il processore: il dispositivo di I/O si limita a mettere l'informazione in un registro di stato, ed il processore deve provvedere ad accedere al registro e a leggere l'informazione. Il processore ha il controllo completo ed esegue tutto il lavoro.

Lo svantaggio del polling sta nel fatto che potenzialmente esso è in grado di far sprecare molto tempo al processore, dal momento che i processori sono molto più veloci dei dispositivi di I/O. Può succedere che il processore legga tante volte il registro di stato, soltanto per scoprire che il dispositivo non ha ancora completato una operazione di I/O che al confronto è molto lenta, o che il mouse non è stato mosso dall'ultima volta che è stato controllato. Una volta che un dispositivo ha completato un'operazione, è ancora necessaria la lettura dello stato per determinare se questa ha avuto successo.

Il polling può essere utilizzato in molti modi diversi, a seconda del dispositivo di I/O e a seconda che il dispositivo di I/O possa iniziare le operazioni di I/O in maniera indipendente. Ad esempio il mouse è un dispositivo di solo ingresso che attiva un'operazione di I/O in maniera indipendente nel momento in cui un utente muove il mouse o schiaccia un bottone. Poiché un mouse ha una bassa velocità di I/O, il polling viene spesso usato per gestire l'interfaccia verso il mouse. Molti altri dispositivi di I/O, come i floppy disk o le stampanti, attivano l'I/O soltanto sotto il controllo del sistema operativo: è quindi necessario controllare questi dispositivi soltanto quando il sistema operativo sa che il dispositivo è attivo. Come si vedrà, questo permette di usare il polling anche quando la velocità di I/O è un po' più alta.

Esempio

Costo aggiuntivo del polling in un sistema di I/O

Si determinerà ora l'impatto del polling in termini di costi aggiuntivi per tre diversi dispositivi. Si assuma che il numero di cicli di clock per un'operazione di polling, includendo il trasferimento del controllo alla procedura di polling, l'accesso al dispositivo, ed il ritorno al programma utente, sia pari a 400 e che il processore lavori con una frequenza di clock pari a 500 MHz.
Si determini la frazione di tempo di CPU consumata nei seguenti tre casi, assumendo che si esegua una interrogazione del dispositivo tramite polling con una frequenza sufficiente a far sì che nessun dato venga mai perso, ed assumendo che i dispositivi siano potenzialmente sempre al lavoro:

1. Il mouse deve essere interrogato in polling 30 volte al secondo per garantire di non perdere alcun movimento fatto dall'utente.

2. Il floppy disk trasferisce dati al processore in unità da 16 bit e con una velocità di trasferimento pari a 50 KB/s. Non è permesso perdere dati.
3. L'hard disk trasferisce dati in blocchi di quattro parole e può eseguire il trasferimento a 4 MB/s. Di nuovo non è permesso perdere dei dati.

Soluzione Si inizi dal mouse:

$$\text{cicli di clock per secondo destinati al polling} = 30 \times 400 = 12000 \text{ cicli per secondo}$$

$$\text{frazione di cicli di clock del processore utilizzati} = \frac{12 \times 10^3}{500 \times 10^6} = 0{,}002\%$$

Il polling può chiaramente essere usato per gestire il mouse senza un grande impatto sulle prestazioni del processore.
Per quanto riguarda il floppy disk, la frequenza alla quale è necessario fare il controllo in polling è

$$\frac{50 \dfrac{\text{KB}}{\text{secondo}}}{2 \dfrac{\text{byte}}{\text{accessi in polling}}} = 25\text{K} \frac{\text{accessi in polling}}{\text{secondo}}$$

Di conseguenza, si può calcolare il numero di cicli (ignorando la discrepanza tra base 2 e base 10):

$$\text{cicli per secondo per il polling} = 25\text{K} \times 400$$

$$\text{frazione di processore utilizzata} = \frac{10 \times 10^6}{500 \times 10^6} = 2\%$$

Questo valore di costo aggiuntivo è significativo, ma può essere tollerabile in un sistema di basse prestazioni contenente pochi dispositivi di I/O del tipo del floppy disk considerato.

Nel caso dell'hard disk è necessario eseguire il polling con una frequenza pari alla frequenza di dato nei blocchi di quattro parole, che è uguale a 250K volte per secondo (4 MB per secondo/16 byte per trasferimento). Quindi

$$\text{cicli per secondo per il polling} = 250\text{K} \times 400$$

Ignorando la discrepanza relativa alle basi si ha

$$\text{frazione di processore utilizzata} = \frac{100 \times 10^6}{500 \times 10^6} = 20\%$$

In conclusione un quinto del processore verrebbe utilizzato soltanto per eseguire il polling del disco. È chiaro che il polling sarebbe a questo punto inaccettabile per la gestione di un hard disk sull'elaboratore considerato.

Se si sapesse che il floppy disk e l'hard disk fossero attivi soltanto per il 25% del tempo e si eseguisse il polling solo quando il dispositivo è attivo, allora il costo aggiuntivo medio per il polling si ridurrebbe allo 0,5% e al 5% rispettivamente. Benché in questa maniera si riduca il costo aggiuntivo, si noti che una volta che il sistema operativo inizia un'operazione sul dispositivo il polling deve essere eseguito senza interruzioni, dal momento che il sistema operativo non sa quando il dispositivo risponderà realmente per iniziare un trasferimento.

Il costo aggiuntivo richiesto da un'interfaccia basata sul polling è noto da lungo tempo, ed ha portato all'invenzione del meccanismo delle interruzioni per segnalare al processore quando un dispositivo di I/O richiede la sua attenzione. L'I/O basato su interruzione, che viene utilizzato da quasi tutti i sistemi almeno per alcuni dispositivi, usa le interruzioni di I/O per indicare al processore che un dispositivo di I/O richiede attenzione. Quando un dispositivo vuole informare il processore di aver completato una qualche operazione o di aver bisogno di attenzione, fa sì che il processore venga interrotto.

Un'interruzione di I/O è praticamente uguale alle eccezioni che si sono viste nei Capitoli 5, 6 e 7 con due importanti differenze:

1. Una interruzione di I/O è asincrona rispetto all'esecuzione delle istruzioni. Questo significa che l'interruzione non è associata con alcuna istruzione e non impedisce il completamento dell'istruzione. Questa situazione è molto diversa sia da quella delle eccezioni dovute a page fault, sia da quella delle eccezioni generate ad esempio da un overflow aritmetico. L'unità di controllo deve soltanto verificare se vi è un'interruzione di I/O in attesa nel momento in cui fa partire una nuova istruzione.
2. In aggiunta al fatto che si è verificata un'interruzione di I/O, occorre fornire maggiori informazioni quali l'identità del dispositivo che ha generato l'interruzione. Inoltre le interruzioni corrispondono a dispositivi che possono avere diverse priorità e le cui richieste di interruzione sono associate a diversi livelli di urgenza.

Per trasmettere al processore informazioni quali l'identità del dispositivo che ha richiesto l'interruzione, un sistema può utilizzare o le interruzioni vettorizzate o un registro Causa per le eccezioni. Quando l'interruzione viene rilevata dal processore, il dispositivo può inviare o l'indirizzo nel vettore o un'informazione di stato che verrà scritta nel registro Causa. Come risultato, quando il sistema operativo acquisisce il controllo è nota l'identità del dispositivo che ha causato l'interruzione e il sistema operativo può immediatamente contattarlo. Un meccanismo di interruzione elimina la necessità per il processore di interrogare il dispositivo mediante il polling e gli permette di concentrarsi sull'esecuzione dei programmi.

Approfondimento: Per gestire le diverse priorità dei dispositivi di I/O, la maggior parte dei meccanismi di interruzione possiede diversi livelli di priorità, che indicano l'ordine nel quale il processore deve elaborare le interruzioni. Possiedono un valore di priorità sia le eccezioni generate internamente sia le interruzioni di I/O; di solito le interruzioni di I/O hanno una priorità minore rispetto alle eccezioni interne. Possono esserci diverse priorità per le interruzioni di I/O: i dispositivi ad alta velocità sono associati alle priorità più alte. Se il meccanismo delle eccezioni è vettorizzato (si veda il Paragrafo 5.6), l'indirizzo nel vettore per un dispositivo veloce corrisponderà all'interruzione ad alta priorità. Se invece viene usato un registro Causa, allora il contenuto del registro per un dispositivo veloce identifica un'interruzione ad alta priorità.

Trasferimento dei dati tra un dispositivo e la memoria

Si sono visti due diversi metodi per abilitare un dispositivo a comunicare con il processore: queste due tecniche, il polling e le interruzioni di I/O, sono la base per due metodi che permettono di implementare il trasferimento di dati tra il dispositivo di I/O e la memoria. Entrambi lavorano al meglio con i dispositivi con minor ampiezza di banda, per i quali si è più interessati a ridurre i costi del controllore del dispositivo e dell'interfaccia che a fornire un trasferimento ad elevata ampiezza di banda. Sia il polling che i trasferimenti basati su interruzione addossano il peso del trasferimento dei dati e della gestione del trasferimento al processore. Dopo aver visto questi due schemi si esaminerà uno schema più adatto a dispositivi ad alte prestazioni o a gruppi di dispositivi.

Si può usare il processore per trasferire dati tra un dispositivo e la memoria utilizzando il polling. Si consideri l'esempio del mouse: il processore può leggere periodicamente i valori del contatore del mouse e la posizione dei pulsanti del mouse; se la posizione del mouse o di uno dei suoi pulsanti è cambiata, il sistema operativo informerà il programma incaricato dell'interpretazione dei movimenti del mouse.

Un meccanismo alternativo consiste nel far sì che il trasferimento dei dati sia gestito tramite le interruzioni. In questo caso il sistema operativo provvede comunque a trasferire i dati da o verso il dispositivo in gruppi composti da un ridotto numero di byte. Dal momento che l'operazione è gestita tramite le interruzioni, il sistema operativo può però lavorare ad altri compiti mentre i dati vengono letti o scritti dal dispositivo. Quando il sistema operativo si accorge di una richiesta di interruzione proveniente da un dispositivo, ne legge lo stato per rilevare eventuali errori. Se non ve ne sono, il sistema operativo fornisce il blocco di dati successivo eseguendo, ad esempio, una sequenza di scritture in memoria. Quando l'ultimo byte di una richiesta di I/O è stato trasmesso e l'operazione di I/O è stata completata il sistema operativo provvede ad informare il programma. In questo processo il processore ed il sistema operativo eseguono tutto il lavoro, accedendo al dispositivo ed alla memoria per ciascun dato da trasferire. Si vedrà ora come potrebbe lavorare un'interfaccia di I/O per un floppy disk gestita tramite interruzioni.

Esempio

Costo aggiuntivo di un'operazione di I/O gestita tramite interruzioni

Si supponga di avere lo stesso hard disk e lo stesso processore visti nell'esempio precedente, ma si utilizzi un meccanismo di I/O basato sulle interruzioni. Il costo aggiuntivo per ogni trasferimento, tenendo conto delle interruzioni, è pari a 500 cicli di clock. Si trovi la frazione del processore utilizzata nel caso in cui l'hard disk stia trasferendo dati per il 5% soltanto del suo tempo.

Soluzione

La frequenza delle interruzioni quando il disco lavora è la stessa della frequenza di polling. Quindi

$$\text{cicli per secondo per il disco} = 250\text{K} \times 500 = 125 \times 10^6 \text{ cicli per secondo}$$

$$\text{frazione del processore utilizzata durante il trasferimento} = \frac{125 \times 10^6}{500 \times 10^6} = 25\%$$

Assumendo che il disco trasferisca dati soltanto durante il 5% del tempo si ha

$$\text{frazione del processore utilizzata in media} = 25\% \times 5\% = 1{,}25\%$$

Come si può vedere, l'assenza di costo aggiuntivo relativa al periodo in cui il dispositivo di I/O non sta effettivamente trasferendo produce il maggior vantaggio dato da un'interfaccia basata su interruzioni anziché su polling.

La gestione dell'I/O con meccanismi basati su interruzione solleva il processore dal peso di attendere il verificarsi degli eventi di I/O. Se però si usasse un tale metodo per il trasferimento di dati da o verso un hard disk, il costo aggiuntivo potrebbe comunque essere intollerabile, dal momento che utilizzerebbe il 25% del processore nel periodo in cui avviene il trasferimento da disco. Per dispositivi con elevata larghezza di banda, come gli hard disk, il trasferimento consiste principalmente di blocchi di dati di dimensioni relativamente elevate (centinaia di migliaia di byte). Di conseguenza i progettisti di calcolatori hanno inventato un meccanismo per togliere lavoro al processore e fare in modo che il controllore del dispositivo provveda direttamente a trasferire i dati da o verso la memoria, senza coinvolgere il processore. Questo meccanismo viene chiamato *accesso diretto alla memoria* (Direct Memory Access, DMA). Il meccanismo delle interruzioni viene comunque usato dal dispositivo per comunicare con il processore, ma soltanto al completamento del trasferimento di I/O o quando si verifica un errore.

Il DMA viene implementato attraverso un controllore apposito che trasferisce i dati tra un dispositivo di I/O e la memoria indipendentemente dal processore. Il controllore di DMA diviene il master del bus e comanda le operazioni di lettura e scrittura tra se stesso e la memoria. Un trasferimento tramite DMA comprende tre passi:

1. Il processore programma l'operazione di DMA fornendo l'identità del dispositivo, l'operazione da eseguire sul dispositivo, l'indirizzo di memoria che costituisce la sorgente o la destinazione per i dati da trasferire ed il numero di byte da trasferire.
2. Il DMA fa partire l'operazione sul dispositivo e provvede all'arbitraggio del bus: quando il dato è disponibile (da parte del dispositivo o della memoria) esegue il trasferimento. Il dispositivo di DMA fornisce l'indirizzo di memoria per la lettura o la scrittura; se la richiesta prevede più di un trasferimento sul bus, l'unità DMA genera l'indirizzo di memoria successivo ed inizia il trasferimento successivo. Utilizzando questo meccanismo un'unità DMA può completare un intero trasferimento, costituito anche da migliaia di byte, senza disturbare il processore. Molti controllori di DMA contengono un certo numero di locazioni di memoria che permettono loro di gestire in maniera flessibile i ritardi nel trasferimento, o quelli che si verificano nell'attesa di diventare master del bus.
3. Una volta che il trasferimento in DMA è stato completato, il controllore interrompe il processore, che potrà poi determinare, consultando il dispositivo di DMA o esaminando la memoria, se l'intera operazione è stata completata con successo.

In un sistema di elaborazione possono esserci diversi dispositivi di DMA. Per esempio in un sistema composto da un singolo bus processore-memoria e da diversi bus di I/O, ciascun controllore dei bus di I/O potrà probabilmente comprendere

un controllore di DMA che gestisce i trasferimenti tra un dispositivo sul bus di I/O e la memoria. Si vedrà ora quale porzione del tempo del processore viene utilizzata adottando il DMA per la gestione del solito esempio dell'hard disk.

Esempio

Costo aggiuntivo di un'operazione di I/O gestita tramite DMA

Si supponga di avere lo stesso processore e lo stesso hard disk dell'esempio precedente. Si assuma che l'operazione di inizializzazione del trasferimento in DMA richieda 1000 cicli di clock per il processore, e si assuma che la gestione dell'interruzione al termine della fase di DMA richieda al processore 500 cicli di clock. L'hard disk ha una velocità di trasferimento di 4 MB/s ed utilizza il meccanismo del DMA. Se la dimensione media di un trasferimento dal disco è pari a 8 KB, quale frazione del processore (che lavora a 500 MHz) viene utilizzata, se il disco esegue attività di trasferimento per il 100% del tempo? Si ignorino gli effetti dei conflitti sul bus tra il processore ed il controllore di DMA.

Soluzione

Ciascun trasferimento in DMA richiede

$$\frac{8\ \text{KB}}{4\dfrac{\text{MB}}{\text{secondo}}} = 2\times10^{-3}\ \text{secondi}$$

Se il disco trasferisce in continuazione, richiederà quindi

$$\frac{1000+500\dfrac{\text{cicli}}{\text{trasferimento}}}{2\times10^{-3}\dfrac{\text{secondi}}{\text{trasferimento}}} = 750\times10^{3}\ \frac{\text{cicli di clock}}{\text{secondo}}$$

Se il processore lavora a 500 MHz

$$\text{frazione del processore utilizzata} = \frac{750\times10^{3}}{500\times10^{6}} =$$

$$= 1{,}5\times10^{-3} = 0{,}2\%$$

Diversamente dalle operazioni di I/O che utilizzano il polling o le interruzioni, il DMA può essere adottato per interfacciare un hard disk senza impegnare tutti i cicli del processore per una singola operazione di I/O. Inoltre il disco non trasferirà attivamente dati per la maggior parte del tempo, e quindi il numero reale sarà considerevolmente più basso di quello calcolato. Ovviamente se il processore sta facendo accesso alla memoria verrà messo in attesa quando la memoria è impegnata con il trasferimento in DMA. Utilizzando le cache il processore può evitare di fare accesso alla memoria per buona parte del tempo, lasciando quindi libera la maggior parte dell'ampiezza di banda della memoria per gli usi propri dei dispositivi di I/O.

Approfondimento: Per ridurre ulteriormente la necessità di interrompere il processore e di impegnarlo nella gestione di una richiesta di I/O che può richiedere diverse operazioni, il controllore di I/O può essere reso più intelligente. I controllori intelligenti sono spesso chiamati *processori di I/O* (o anche *controllori di I/O* o *controllori di canale*). Questi processori specializzati eseguono sostanzialmente una serie di operazioni di I/O, chiamate *programmi di I/O*. Il programma può essere memorizzato all'interno del processore di I/O, oppure in memoria, e caricato successivamente dal processore di I/O. Quando si usa un processore di I/O, il sistema operativo di solito prepara un programma di I/O che specifica le operazioni di I/O che devono essere eseguite, nonché la dimensione e l'indirizzo di trasferimento per le operazioni di lettura e scrittura. Il processore di I/O deduce poi le operazioni dal programma di I/O ed interrompe il processore soltanto quando l'intero programma è stato completato. I processori di DMA sono essenzialmente dei processori dedicati (di solito composti da un singolo chip e non programmabili), mentre i processori di I/O sono di solito implementati sotto forma di processori general-purpose, che eseguono un programma speciale per l'I/O.

L'accesso diretto in memoria ed il sotto-sistema di memoria

Quando nel sistema di I/O viene incluso il DMA, i rapporti tra il sistema di memoria ed il processore cambiano. Senza il DMA, tutti gli accessi al sistema di memoria provengono dal processore e quindi procedono attraverso la traduzione degli indirizzi e l'accesso alla cache. Nella gestione della memoria da parte del DMA si utilizza un meccanismo diverso, che non passa attraverso il meccanismo di traduzione degli indirizzi o la gerarchia delle cache. Tale differenza può generare alcuni problemi sia nei sistemi a memoria virtuale, sia nei sistemi con cache; questi problemi sono normalmente risolti utilizzando una combinazione di tecniche hardware e di supporto software.

Le difficoltà che nascono allorquando si ha il DMA all'interno di un sistema a memoria virtuale sono dovute al fatto che le pagine possiedono sia un indirizzo fisico che un indirizzo virtuale. Il DMA può determinare situazioni di criticità nei sistemi dotati di cache poiché possono esserci due copie di ciascun dato, una nella cache ed una in memoria: poiché il controllore di DMA controlla direttamente gli accessi in memoria senza passare attraverso la cache, l'indirizzo di una locazione di memoria può differire a seconda che venga generato dall'unità DMA o dal processore. Si consideri una lettura da disco nella quale l'unità DMA scrive i dati direttamente in memoria: se qualcuna delle locazioni nelle quali il DMA scrive si trova in cache, il processore, eseguendo una lettura, riceverà il vecchio valore. Analogamente, se la cache è di tipo write-back, il DMA può leggere un valore direttamente dalla memoria mentre un valore più aggiornato, che non è ancora stato scritto in memoria, si trova nella cache. Questo fenomeno è chiamato *problema dei dati scaduti* o *problema della coerenza*.

Interfaccia Hardware Software

In un sistema con memoria virtuale il DMA deve lavorare con gli indirizzi virtuali o con quelli fisici? Se si lavora con gli indirizzi virtuali è ovvio che l'unità DMA dovrà tradurli in indirizzi fisici. Il problema principale derivante dall'uso di un indirizzo fisico in un trasferimento mediante DMA sta nel fatto che il trasferimento non può superare facilmente i confini di una pagina. Se una richiesta di I/O supera i confini di una pagina, allora le locazioni di memoria verso le quali è in corso il trasferimento potrebbero non esse-

re contigue nella memoria fisica: le locazioni di memoria corrisponderebbero a diverse pagine virtuali, ciascuna delle quali potrebbe corrispondere ad una qualsiasi pagina fisica. Di conseguenza se si usano indirizzi fisici si deve far sì che tutti i trasferimenti in DMA siano limitati ad una sola pagina.

Un metodo che permette al sistema di iniziare dei trasferimenti in DMA che superino i confini di una pagina consiste nel far sì che il DMA lavori sugli indirizzi virtuali. In un tale sistema l'unità DMA contiene un numero ridotto degli elementi della mappa che permette la trasformazione di indirizzi virtuali in fisici. Il sistema operativo fornisce tali elementi quando viene iniziata l'operazione di I/O: utilizzandoli l'unità di DMA non deve preoccuparsi della posizione delle pagine virtuali coinvolte nel trasferimento.

Un'altra tecnica consiste nel far sì che il sistema operativo suddivida il trasferimento in DMA in una serie di trasferimenti, ciascuno confinato ad una singola pagina fisica. I trasferimenti sono poi concatenati e gestiti da un processore di I/O o da un'unità DMA intelligente che esegue l'intera sequenza di trasferimenti; in alternativa il sistema operativo può richiedere i trasferimenti ad uno ad uno.

Qualunque sia il metodo utilizzato, il sistema operativo deve comunque evitare di rilocare una pagina mentre è in corso un trasferimento tramite DMA che coinvolge quella pagina.

Si sono visti tre diversi modi per trasferire dati tra un dispositivo di I/O e la memoria; nel passare dal polling alle interruzioni al DMA, si è spostato il peso della gestione di un'operazione di I/O dal processore verso un controllore di I/O, via via sempre più intelligente. Questi metodi hanno il vantaggio di liberare cicli del processore; il loro svantaggio sta nel fatto che essi accrescono il costo del sistema di I/O. Nella definizione di un sistema di elaborazione si deve scegliere la modalità più appropriata per la gestione dei suoi dispositivi di I/O.

Interfaccia Hardware Software

Il problema della coerenza dei dati di I/O può essere evitato utilizzando una delle seguenti tre tecniche. Il primo approccio consiste nel far passare l'attività di I/O attraverso la cache: questo garantisce che ciascuna lettura veda il valore più aggiornato del dato, mentre le scritture aggiornano i dati nella cache. Far passare tutte le operazioni di I/O attraverso la cache è costoso e può avere un impatto largamente negativo sulle prestazioni del processore, poiché i dati di I/O sono raramente usati immediatamente dopo il loro arrivo, mentre questa tecnica può ritardare dei dati utili richiesti dal programma in esecuzione. Una seconda tecnica consiste nel far sì che il sistema operativo invalidi in maniera selettiva la cache a fronte di una lettura di I/O o forzi l'esecuzione del write-back per una scrittura di I/O (questa operazione è chiamata scaricamento della cache, o flush). Questo approccio richiede una piccola quantità di hardware di supporto ed è in generale più efficiente se il software può eseguire con facilità ed efficacia questa funzione. Dal momento che lo scaricamento di parti consistenti della cache è richiesto soltanto nel caso di accessi tramite DMA, esso sarà relativamente poco frequente. Il terzo approccio si basa su un meccanismo hardware per

scaricare (o invalidare) in maniera selettiva gli elementi della cache. L'invalidazione via hardware per garantire la coerenza della cache è comune nei sistemi multiprocessore, e la stessa tecnica può essere utilizzata per l'I/O; questo argomento verrà discusso in dettaglio nel Capitolo 9.

8.6 Il progetto di un sistema di I/O

Esistono due tipi principali di specifiche che vengono fornite ai progettisti dei sistemi di I/O: i vincoli sulla latenza e quelli sull'ampiezza di banda. In ambedue i casi la conoscenza delle caratteristiche del traffico da considerare influenza il progetto e l'analisi.

I vincoli sulla latenza mirano a garantire che la latenza per il completamento di un'operazione di I/O sia limitata ad un certo valore massimo. Nel caso più semplice il sistema è scarico, ed il progettista deve garantire che un certo valore massimo di latenza sia soddisfatto o perché questo è critico per l'applicazione o perché il dispositivo deve assolutamente ricevere un certo servizio entro tale tempo per evitare situazioni di errore. Esempi di quest'ultimo caso sono simili a quanto visto nel precedente paragrafo. Calcolare la latenza di un sistema scarico è relativamente facile, dal momento che richiede di tracciare l'evoluzione di un'operazione di I/O e di sommare le singole latenze delle varie fasi.

Calcolare la latenza media (o la distribuzione di latenza) a fronte di un carico è invece un problema molto più complesso. Tali problemi vengono affrontati o ricorrendo alla teoria delle code (allorché il comportamento delle richieste ed i tempi di servizio per l'I/O previsti dal carico di lavoro possano essere approssimati da semplici distribuzioni) o tramite simulazione (quando il comportamento degli eventi di I/O è complesso). Entrambi questi argomenti vanno al di là degli scopi di questo testo.

Il progetto di un sistema di I/O che soddisfi un insieme di vincoli sull'ampiezza di banda, dato un certo carico di lavoro, è l'altro tipico problema che si presenta ai progettisti. In alternativa al progettista può essere fornito un sistema di I/O parzialmente configurato e gli si può chiedere di bilanciare il sistema per mantenere la massima ampiezza di banda raggiungibile secondo le caratteristiche della parte di sistema già configurata. Questo secondo problema in termini di progetto è una versione semplificata del primo.

L'approccio seguito per il progetto di un tale sistema è in generale il seguente:

1. Si determina il collegamento di I/O più critico all'interno del sistema di I/O, ossia quello che vincola il progetto. A seconda delle condizioni di carico, tale criticità può trovarsi nella CPU, nel sistema di memoria, nel bus di backplane, nei controllori di I/O, o nei dispositivi. Sia il carico di lavoro che i vincoli sulla configurazione possono determinare la situazione piùcritica.
2. Si progetta tale collegamento in maniera da poter sostenere l'ampiezza di banda richiesta.
3. Si determinano i vincoli per il resto del sistema e lo si configura in maniera tale da poter supportare tale ampiezza di banda.

Il modo più semplice per capire questa metodologia è di analizzare un esempio.

Il progetto di un sistema di I/O

Esempio Si consideri il seguente sistema di elaborazione:

- Una CPU in grado di lavorare a 300 milioni di istruzioni per secondo, con una media di 50.000 istruzioni del sistema operativo per ciascuna operazione di I/O.
- Un bus di backplane verso la memoria capace di fornire una velocità di trasferimento di 100 MB/s.
- Controllori SCSI-2 con una velocità di trasferimento di 20 MB/s e la possibilità di gestire fino a 7 dischi.
- Unità disco con un'ampiezza di banda per lettura e scrittura di 5 MB/s ed un tempo medio di seek più latenza di rotazione pari a 10 ms.

Se il carico di lavoro consiste in operazioni di lettura di blocchi di 64 KB (il blocco si trova sequenzialmente su una traccia) e il programma utente richiede 100.000 istruzioni per ciascuna operazione di I/O, si trovi la massima velocità di I/O sostenibile e il numero di dischi e di controllori SCSI necessari. Si assuma che le letture possano sempre essere eseguite su un disco scarico se ne esiste uno (si ignorino cioè i conflitti sui dischi).

Soluzione I due componenti fissi di un sistema sono il bus di memoria e la CPU. Si troverà innanzitutto la velocità di I/O sostenibile da parte di questi due componenti, determinando quale di questi costituisce il collo di bottiglia. Ciascuna operazione di I/O richiede 100.000 istruzioni utente e 50.000 istruzioni del sistema operativo; quindi

$$\text{massima velocità di I/O della CPU} = \frac{\text{velocità di esecuzione di un'istruzione}}{\text{istruzioni per l'I/O}} =$$

$$= \frac{300 \times 10^6}{(50+100) \times 10^3} = 2000 \frac{\text{operazioni di I/O}}{\text{secondo}}$$

Ciascun trasferimento di I/O è pari a 64 KB, e quindi

$$\text{massima velocità di I/O del bus} =$$

$$= \frac{\text{ampiezza di banda del bus}}{\text{byte per l'I/O}} = \frac{100 \times 10^6}{64 \times 10^3} = 1562 \frac{\text{operazioni di I/O}}{\text{secondo}}$$

Il collo di bottiglia è costituito dal bus, e quindi si può configurare il resto del sistema in maniera tale che esso abbia prestazioni dettate dal bus, ossia 1562 operazioni di I/O per secondo.

Si determini ora quanti dischi sono necessari per poter eseguire 1562 operazioni di I/O per secondo. Per trovare il numero di dischi si deve prima calcolare il tempo richiesto per ciascuna operazione su disco:

$$\text{tempo per operazione di I/O su disco} =$$
$$= \text{tempo di seek/rotazione} + \text{tempo di trasferimento} =$$

$$= 10 \text{ ms} + \frac{64 \text{ KB}}{5 \text{ MB/s}} = 22{,}8 \text{ ms}$$

Questo significa che ciascun disco può completare 43,9 operazioni di I/O per secondo. Per saturare il bus ci vogliono 1562 operazioni di I/O per secondo, ossia 1562/43,9 = 36 dischi.

Per calcolare il numero di bus SCSI è necessario conoscere la velocità di trasferimento media per disco, che è data da

$$\text{velocità di trasferimento} =$$

$$= \frac{\text{dimensione del trasferimento}}{\text{tempo di trasferimento}} = \frac{64 \text{ KB}}{22{,}8 \text{ ms}} = 2{,}74 \text{ MB/s}$$

Assumendo che gli accessi a disco non siano raggruppati, così che si possa usare tutta la banda del bus, si possono mettere sette dischi per ciascun bus e controllore SCSI. Questo significa che ne saranno necessari 36/7, ossia sei bus e sei controllori.

Si noti l'elevato numero di assunzioni semplificative necessarie per affrontare questo esempio. In pratica molte di queste semplificazioni potrebbero non essere verificate per varie applicazioni critiche dal punto di vista dell'I/O (come ad esempio la gestione di una base dati). Per questa ragione la simulazione è spesso l'unica strada realistica per predire le prestazioni di I/O a fronte di un certo carico di lavoro.

8.7 Un caso reale: un tipico sistema di I/O per un calcolatore da tavolo

L'emergere di due standard dominanti nel mercato dei personal computer da tavolo ha portato ad un livello elevatissimo di omogeneità tra i vari sistemi di I/O. I due standard menzionati sono PCI come bus di backplane e SCSI o SCSI-2 come bus di I/O. Benché sistemi con bus più vecchi (ISA o IDE) continuino ad essere venduti, tali sistemi sono stati rapidamente sostituiti su tutti i modelli, ad esclusione di quelli meno costosi e con le minori prestazioni. È interessante notare come i benefici derivanti dall'esistenza di un singolo standard di bus, in termini di disponibilità di un maggior numero di dispositivi e ad un minore costo, hanno portato all'adozione dello standard per i bus di backplane e di I/O indipendentemente dalle piattaforme (IBM-compatibili e Macintosh), mentre anche un numero crescente di produttori di workstation sta aderendo a questi standard.

La figura 8.16 mostra il sistema di I/O della serie Macintosh 7200, che rappresenta un esempio tipico di un sistema di I/O del 1997 per la fascia di elaboratori che va da quelli di medio livello sino alle macchine da ufficio di alto profilo. Come bus di backplane si utilizza PCI, mentre i dispositivi più lenti condividono un bus di più basse prestazioni, come uno SCSI.

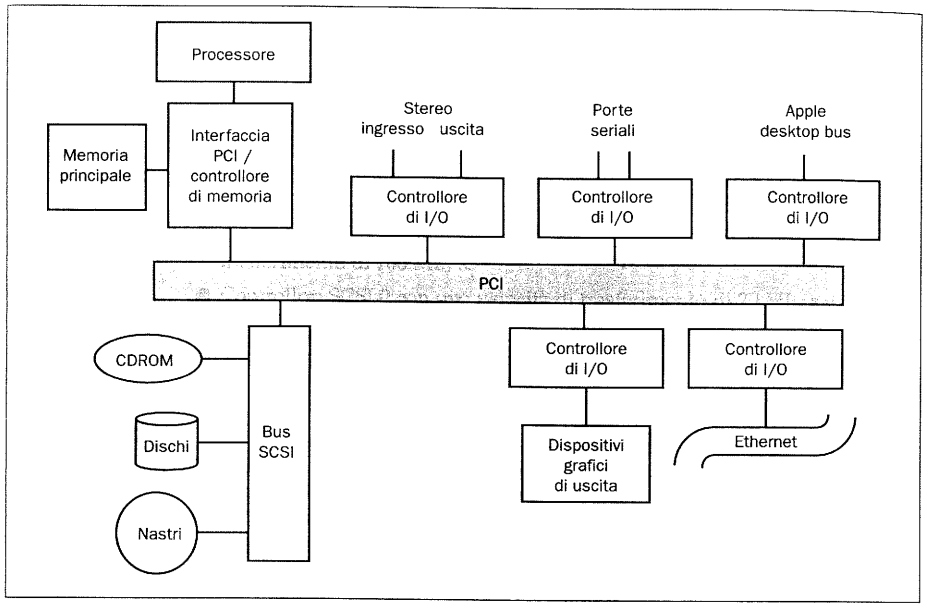


Figura 8.16 Organizzazione del sistema di I/O sugli elaboratori della serie Apple Macintosh 7200. Il bus di backplane PCI è utilizzato per collegare tutti i dispositivi e le interfacce verso il processore ed il sistema di memoria. Le porte seriali permettono le connessioni quali quella verso la rete a bassa velocità Appletalk. Il desktop bus permette di supportare la tastiera ed il mouse. In realtà alcuni dei dispositivi di I/O più lenti (I/O audio, porte seriali e desktop bus) condividono una stessa porta del bus PCI, anche se per semplicità qui vengono mostrati separati.

8.8 Errori e trabocchetti

Errore: un bus a 100 MB/s può trasferire 100 MB di dati in 1 secondo.

Ovviamente si tratta di un errore solo quando la definizione di megabyte di memoria e di megabyte per secondo di ampiezza di banda non coincidono. Come già detto in precedenza, le misure di ampiezza della banda di I/O sono normalmente espresse in base 10 (vale a dire che 1 MB/s = 10^6 byte/s), mentre 1 MB di dato è una tipica misura espressa in base 2 (quindi 1 MB = 2^{20} byte). Quanto è importante questa distinzione? Il tempo per trasferire 100 MB di dati su un bus a 100 MB/s è pari a

$$\frac{100 \times 2^{20}}{100 \times 10^6} = \frac{1048576}{1000000} = 1,048576 \approx 1 \text{ secondo}$$

Un errore simile, ma più piccolo viene introdotto quando si trattano i kilobyte come fossero equivalenti, confondendo 10^3 o 2^{10}; un errore più grande viene invece introdotto quando si trattano i gigabyte, confondendo 10^9 e 2^{30} byte.

Trabocchetto: usare la velocità di trasferimento di picco di una parte del sistema di I/O per fare proiezioni di prestazioni o confronti tra prestazioni.

Molti dei componenti di un sistema di I/O, dai dispositivi ai controllori ai bus, sono descritti fornendone l'ampiezza di banda di picco. Nella pratica le misure di ampiezza di banda di picco sono spesso basate su assunzioni non realistiche a riguardo del sistema, oppure non sono raggiungibili a causa di altri limiti del sistema stesso. Per esempio, nel riportare le prestazioni di un bus, la velocità di trasferimento di picco viene spesso specificata con riferimento a sotto-sistemi di memoria impossibili da costruire.

Un bus PCI ha un'ampiezza di banda di picco di circa 133 MB/s. In pratica anche per trasferimenti di grandi dimensioni è difficile ottenere più di 80 MB/s con sistemi di memoria realistici.

La legge di Amdahl ricorda inoltre che il throughput di un sistema di I/O sarà limitato dal componente sul cammino di I/O con le prestazioni peggiori.

Errore: le memorie magnetiche sono al tramonto e saranno presto rimpiazzate da altre tecnologie.

Questo costituisce sia un errore che un trabocchetto. Questi discorsi sono stati fatti senza interruzione per tutti gli ultimi 20 anni, anche se la serie di fallimenti che ha accompagnato le alternative comparse negli ultimi anni sembra aver ridotto il credito di questi discorsi relativi alla morte delle memorie magnetiche. Tra le alternative proposte per sostituire le memorie magnetiche che non hanno avuto successo vi sono le memorie a bolle magnetiche, i dispositivi ottici e le memorie fotografiche. Nessuno di questi sistemi ha fornito la stessa combinazione di caratteristiche positive che caratterizza i dischi magnetici: non volatilità, basso costo, tempo di accesso ragionevole ed alta affidabilità. La tecnologia delle memorie magnetiche continua a migliorare con una velocità uguale o maggiore a quello mantenuta negli ultimi 25 anni. Infatti il tasso di miglioramento nella densità di memorizzazione è andato aumentando negli ultimi 10 anni, ed anche la velocità di rotazione ed i tempi di seek sono migliorati in maniera significativa negli ultimi anni.

Trabocchetto: spostare funzioni dalla CPU ai processori di I/O, sperando di migliorare le prestazioni, senza eseguire un'analisi attenta.

Ci sono molti esempi di trappole in cui sono caduti i progettisti, anche se i processori di I/O, quando correttamente usati, possono senz'altro migliorare le prestazioni. Un caso frequente di questo trabocchetto consiste nell'uso di interfacce di I/O intelligenti che, a causa dei tempi elevati per iniziare l'operazione di I/O, possono produrre una latenza anche peggiore di quella ottenibile con un'attività di I/O gestita direttamente dal processore (anche se il throughput del sistema può comunque migliorare se il processore viene alleggerito). Di frequente le prestazioni peggiorano quando il processore di I/O ha prestazioni molto minori di quelle del processore principale. Di conseguenza una piccola parte del tempo del processore principale viene sostituita da una grossa parte del tempo del processore di I/O. I progettisti di workstation hanno sperimentato più volte questo fenomeno.

Un problema più serio può verificarsi quando la migrazione di una funzionalità di I/O cambia l'architettura del set di istruzioni o l'architettura del sistema in maniera visibile al programmatore. Questo costringe tutti gli elaboratori successivi a coesistere con una decisione che aveva senso solo nel passato. Se le CPU migliorano in termini di rapporto costo/prestazioni più rapidamente dei processori di

I/O (e questo è il caso più probabile), lo spostamento della funzionalità può corrispondere ad ottenere un elaboratore ancora più lento nella versione successiva della macchina.

L'esempio più istruttivo è quello dell'IBM 360. Si era deciso che le prestazioni del sistema ISAM (un primitivo sistema di base di dati) sarebbero migliorate se alcune delle funzioni di ricerca di record fossero state eseguite dal controllore del disco. Venne associato quindi ad ogni record un campo chiave, ed il dispositivo venne messo in grado di cercare una chiave mentre il disco ruotava, sino a che non si trovava una corrispondenza. A quel punto il disco trasferiva il record desiderato. Quando esiste una chiave, questa tecnica richiede una spaziatura più ampia tra i record.

La velocità alla quale la traccia può essere scandita per la ricerca è limitata dalla velocità del disco e dal numero di chiavi che possono essere inserite in ciascuna traccia. Su un disco IBM 3330 la chiave è composta di solito da 10 caratteri; l'intervallo corrisponde a 131 caratteri se vi è la chiave, e a 135 se la chiave non è presente. Se si assume che il dato sia anch'esso composto da 10 caratteri e che la traccia non contenga altro al suo interno, una traccia composta da 13.165 byte può contenere

$$\frac{13165}{191+10+10} = 62 \text{ record chiave} - \text{dato}$$

Il tempo per la ricerca della chiave è pari a

$$\frac{16{,}7 \text{ ms (1 giro)}}{62} = 0{,}27 \text{ ms/chiave cercata}$$

Al posto di questo schema si possono mettere più coppie chiave-dato in un solo blocco ed utilizzare degli intervalli più piccoli tra i record. Si assuma che vi siano 15 coppie chiave-dato per ciascun blocco e che la traccia non contenga altro; allora

$$\frac{13165}{13+15\times(10+10)} = \frac{13165}{135+300} = 30 \text{ blocchi di coppie chiave} - \text{dato}$$

In questo caso le prestazioni sono quindi

$$\frac{16{,}7 \text{ ms (1 giro)}}{30\times 15} = 0{,}04 \text{ ms/chiave cercata}$$

Ovviamente la ricerca basata sul disco funzionerebbe meglio se le chiavi fossero molto più lunghe.

Quando i processori divennero più veloci, il tempo di CPU per una ricerca divenne trascurabile, mentre il tempo per una ricerca che utilizzava la tecnica hardware migliorò assai poco. Mentre quindi la strategia descritta rese più veloci i primi elaboratori, i programmi che utilizzano l'operazione di ricerca di una chiave utilizzando il processore di I/O sugli elaboratori di oggi lavorano sino a sei volte più lentamente.

8.9 Note conclusive

I sistemi di I/O vengono valutati sulla base di diverse caratteristiche: la varietà di dispositivi di I/O supportati, il massimo numero di dispositivi di I/O, il costo, le

prestazioni (misurate sia in termini di latenza che di throughput). Questi obiettivi hanno prodotto la nascita e l'adozione di schemi molto diversi per l'interfacciamento dei dispositivi di I/O. All'estremo inferiore, schemi quali l'uso di buffer e lo stesso DMA possono essere evitati per minimizzare il costo. Nella fascia dei sistemi di medio livello il DMA con buffer è probabilmente il meccanismo dominante per i trasferimenti. Nella fascia superiore, la latenza e l'ampiezza di banda possono essere entrambi molto importanti, mentre il costo può diventare un parametro secondario. I sistemi di I/O ad elevate prestazioni sono spesso caratterizzati da cammini multipli verso i dispositivi di I/O con una quantità di buffer limitata; se si aumenta l'ampiezza di banda utilizzando un numero più elevato di connessioni con dimensioni maggiori cade la necessità dei buffer ma aumenta il costo. Di solito la possibilità di accedere ai dati su un dispositivo di I/O in un qualunque momento (elevata disponibilità) diventa sempre più importante mano a mano che i sistemi crescono di dimensione. Come risultato, la ridondanza ed i meccanismi di correzione dell'errore diventano sempre più comuni quando il sistema cresce.

Il progetto dei sistemi di I/O è reso più complicato dal fatto che il fattore limitante nelle prestazioni di un sistema di I/O può essere una qualsiasi delle risorse critiche poste sul cammino di I/O, dal sistema operativo al dispositivo. Inoltre, richieste indipendenti provenienti da programmi diversi interagiscono tra loro all'interno del sistema di I/O, facendo sì che le prestazioni di un sistema di I/O dipendano da altre attività che si svolgono contemporaneamente. Infine le tecniche di progetto che migliorano l'ampiezza della banda spesso hanno effetti negativi sulla latenza, e viceversa. Se ad esempio si aggiungono dei buffer, di solito aumenta sia il costo che l'ampiezza di banda del sistema. È anche vero però che, se si mette dell'hardware aggiuntivo tra il dispositivo e la memoria, aumenta la latenza. Questa combinazione di fattori, alcuni dei quali imprevedibili, fa sì che il progetto dei sistemi di I/O ed il miglioramento delle loro prestazioni siano problemi critici e stimolanti non solo per i progettisti dell'hardware, ma anche per i progettisti dei sistemi operativi, e persino per chi programma applicazioni ad elevato I/O.

Quadro d'insieme

Le prestazioni di un sistema di I/O, misurate in termini sia di ampiezza di banda, sia di latenza, dipendono da tutti gli elementi esistenti sul cammino tra il dispositivo e la memoria, compreso il sistema operativo, che genera i comandi di I/O. L'ampiezza di banda dei bus, la memoria ed il dispositivo determinano la massima velocità di trasferimento da o verso il dispositivo. Analogamente la latenza dipende dalla latenza del dispositivo, insieme con ogni altra latenza introdotta dal sistema di memoria o dai bus. L'ampiezza di banda effettiva e la latenza della risposta dipendono inoltre dalle altre richieste di I/O che possono provocare dei conflitti su alcune risorse esistenti sul cammino. Infine il sistema operativo costituisce un collo di bottiglia: in alcuni casi il sistema operativo richiede un tempo lungo per trasferire una richiesta di I/O da un programma utente ad un dispositivo di I/O, causando così una latenza elevata. In altri casi il sistema operativo limita effettivamente l'ampiezza della banda di I/O a causa del ridotto numero di operazioni di I/O concorrenti che può gestire.

Direzioni future nei sistemi di I/O

Cosa vi è nel futuro dei sistemi di I/O? I rapidi miglioramenti nelle prestazioni dei processori mettono in crisi i sistemi di I/O, i cui componenti fisici non possono migliorare in termini di prestazioni così velocemente come i processori. Per colmare il crescente divario tra la velocità dei processori ed il tempo di accesso alla memoria secondaria (principalmente ai dischi) si utilizza la memoria principale come una cache rispetto alla memoria secondaria. Questa *cache dei file*, che si basa sulla località spaziale e temporale nell'accesso alla memoria secondaria, è gestita dal sistema operativo. L'uso delle cache dei file permette di far sì che molti accessi a file siano gestiti dalla memoria piuttosto che dal disco.

I dischi magnetici hanno capacità che crescono rapidamente, ma il tempo di accesso sta migliorando molto più lentamente. Una delle spiegazioni di questo fenomeno sta nel fatto che i dischi magnetici hanno un'evoluzione più veloce nella fascia bassa piuttosto che in quella alta del mercato, e la fascia bassa registra una spinta maggiore verso un limitato costo per megabyte. Questo mercato ha dato il suo contributo alla riduzione nella dimensione dei dischi dai piatti da 14 pollici (circa 35 cm) usati per i dischi dei mainframe ai dischi da 1,3 pollici (circa 3,5 cm) sviluppati per i calcolatori portatili e palmari. Infatti l'elevatissima richiesta di dischi piccoli ha portato ad un'accelerazione nei miglioramenti relativi alla densità dei dischi, così che la densità dei dischi magnetici a partire dal 1990 è cresciuta più rapidamente che in ogni altro periodo precedente! Un aspetto sorprendente consiste nel fatto che questo periodo di crescita è giunto nel momento in cui un certo numero di persone stava predicendo la fine (o almeno un riduzione dell'uso) dei dischi magnetici.

Oltre ad aumentare la densità sono cresciute rapidamente anche le velocità di trasferimento, mano a mano che i dischi aumentavano la propria velocità di rotazione e miglioravano le interfacce. Inoltre pressoché ogni disco ad alte prestazioni prodotto oggi contiene un buffer di traccia o di settore che implementa un meccanismo di cache che memorizza i settori quando la testina passa sopra di essi.

Un'importante novità in termini di organizzazione dei dischi emersa negli ultimi anni è costituita dalle schiere di dischi piccoli ed economici. La motivazione che spinge ad adottare delle schiere di dischi è che, poiché il prezzo per megabyte è indipendente dalla dimensione del disco, il throughput potenziale può essere migliorato utilizzando più unità disco e quindi più testine: la distribuzione dei dati su più dischi fa sì che gli accessi avvengano su più dischi. Mentre l'uso delle schiere migliora il throughput, la latenza non viene però necessariamente ridotta. L'aggiunta di dischi ridondanti offre la possibilità di scoprire un disco guasto e recuperare automaticamente l'informazione perduta. Le schiere possono quindi migliorare l'affidabilità e le prestazioni dei sistemi di elaborazione. Tale ridondanza ha ispirato l'acronimo RAID (*Redundant Arrays of Inexpensive Disks*, o schiere ridondanti di dischi a basso costo), utilizzato per definire tali schiere. Numerose società informatiche offrono dischi RAID per i loro sottosistemi. Ad esempio, IBM offre sia la soluzione basata su RAID (si veda il link a IBM all'indirizzo *www.mkp.com/books_catalog/cod/links.htm*), sia un sottosistema di dischi che utilizza i dischi più grandi prodotti dall'azienda.

Il livello successivo nella gerarchia delle memorie al di sotto dei dischi magnetici (costituito dai nastri magnetici) è stato anch'esso caratterizzato negli ultimi anni da aumenti straordinari in termini di capacità. Questi aumenti sono derivati in parte dagli stessi miglioramenti nelle tecniche di memorizzazione magnetica di cui

hanno beneficiato i dischi, ma anche da una diversa tecnologia di memorizzazione, denominata *nastro a scansione elicoidale*. Sviluppati originariamente per i videoregistratori, le videocamere ed i nastri audio digitali, i nastri a scansione elicoidale memorizzano i dati con una certa angolazione rispetto al nastro, anziché in maniera parallela, come avviene nei nastri scritti longitudinalmente. Il nastro continua a muoversi alla stessa velocità, ma la testina ruota velocemente sul nastro memorizzando i bit in maniera molto più densa (rispetto ai nastri memorizzati longitudinalmente vi è un fattore di densità variabile tra le 50 e le 100 volte). Dal momento che il prodotto è stato creato per la fascia *consumer*, il miglioramento nel tempo in termini di costo per bit è stato più alto che non per i tradizionali nastri magnetici usati soltanto dall'industria informatica.

I miglioramenti nella capacità dei nastri sono stati esaltati dai miglioramenti registrati su due altri fronti: la compressione ed i robot. La disponibilità di processori più veloci ha permesso ai sistemi di iniziare ad adottare la compressione per moltiplicare la capacità di immagazzinamento. Fattori variabili da 2 a 3 sono comunemente raggiunti, e sono possibili compressioni anche di 20:1 per alcuni tipi di dati, come le immagini. Il secondo miglioramento che sta cambiando il rapporto costo-prestazioni dei sistemi di memorizzazione on-line di grandi dimensioni è la disponibilità di robot a basso costo per caricare e scaricare automaticamente i nastri; in tal modo si crea un nuovo livello nella gerarchia delle memorie, tra i dischi magnetici on-line ed i nastri magnetici off-line sui ripiani. Questi sistemi di immagazzinamento robotizzati permettono l'accesso a terabyte di informazione con ritardi di qualche decina di secondi, senza l'intervento di un operatore umano. La figura 8.17 contiene una fotografia di un'unità nastro robotizzata.

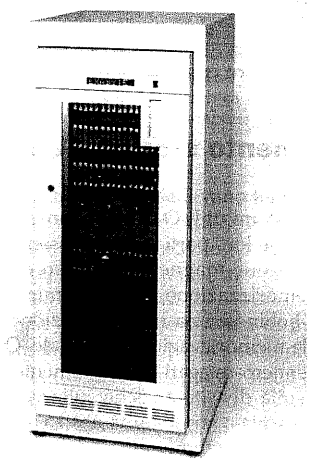


Figura 8.17 L'Exabyte EXB-120 contiene 116 nastri da 8 mm a scansione elicoidale. Ciascun nastro contiene 10 GB, fornendo quindi una capacità complessiva superiore al terabyte. L'EXB-120 ha un costo pari a circa 2-4 workstation. Fotografia gentilmente concessa dalla Exabyte Corporation.

Anche le reti di calcolatori stanno facendo grandi passi in avanti. Le soluzioni basate su Ethernet a 100 Mbit e su Ethernet commutata sono oggi utilizzate nelle nuove reti e nell'aggiornamento di reti esistenti che non possono gestire la tremenda esplosione nell'ampiezza di banda richiesta, conseguente all'uso di applicazioni multimediali e alla crescente importanza del World Wide Web. ATM costituisce un'altra tecnologia potenzialmente in grado di ulteriori espansioni. Per supportare la crescita del traffico le strutture portanti della rete Internet si stanno trasferendo su fibra ottica, che permette un significativo aumento dell'ampiezza di banda per reti a lunga distanza.

Una delle tecnologie di memorizzazione più interessanti tra quelle esplorate è l'olografia. Un progetto di ricerca attualmente in corso mira a provare la realizzabilità di un dispositivo di memorizzazione con capacità dell'ordine del terabyte e velocità di trasferimento di 1 Gbit/s. Questo rappresenterebbe un miglioramento di circa un ordine di grandezza sia nella dimensione della memoria sia nella velocità di trasferimento rispetto ai dischi più grandi disponibili nel 1997. Si veda il relativo link a IBM all'indirizzo *www.mkp.com/books_catalog/cod/links.htm* per una descrizione di questa attività di ricerca congiunta tra università ed industria.

I passi avanti descritti delineano degli scenari da fantascienza informatica che sarebbero sembrati assurdi soltanto qualche anno addietro. Se ad esempio tutti i libri della Biblioteca del Congresso di Washington fossero convertiti in ASCII, occuperebbero soltanto 10 terabyte (anche se le immagini potrebbero richiedere molto più spazio, a seconda del numero e della definizione). I nastri a scansione elicoidale, le unità nastro robotizzate, la compressione e le reti ad alta velocità potrebbero rappresentare gli elementi base di una biblioteca elettronica. Tutte le informazioni contenute in tutti i libri del mondo potrebbero essere a portata di mano al costo di un minicomputer. Il calcolo parallelo, trattato nel successivo capitolo, permetterà di indicizzare queste informazioni in modo tale che tutti i libri potranno essere oggetto di ricerche effettuate sulla base del contenuto anziché del titolo. Le biblioteche elettroniche potrebbero cambiare la vita di tutti grazie ad una carta di accesso, e la tecnologia per la loro creazione è oggi a portata di mano.

8.10 Inquadramento storico e approfondimenti

La storia dei sistemi di I/O è affascinante. I dispositivi di I/O dei primi calcolatori sono spesso tra le loro parti più interessanti. Il nastro magnetico fu il primo sistema di memorizzazione magnetico a basso costo e continua ancor oggi ad essere il mezzo di memorizzazione più economico. Le prime unità a nastro usavano una tecnologia a bobine e una memorizzazione lineare, che furono poi sostituite dalle casette e dalla memorizzazione elicoidale. Quando i dischi divennero più economici, i nastri vennero relegati principalmente a funzioni di archiviazione, spostando l'attenzione sulla densità anziché sulla velocità, nonché su tecnologie di archiviazione su larga scala come le unità nastro robotizzate.

I primi dispositivi di memoria ad accesso casuale furono i tamburi ed i dischi a testina fissa. Un tamburo possedeva una superficie cilindrica rivestita di una pellicola magnetica, ed utilizzava un elevato numero di testine di lettura/scrittura, posizionate sopra ciascuna traccia sul tamburo (si veda la figura 8.18). I tamburi erano dispositivi di I/O con velocità relativamente alta, utilizzati spesso per realizzare i meccanismi di memoria virtuale o per creare una cache dei file per i dispositivi a minore velocità. I tamburi, che non possedevano tempo di seek, sopravvissero sino agli anni '70 in applicazioni ad alta velocità, sino a che i rilevanti vantaggi dei dischi in termini di costo eliminarono la tecnologia dei tamburi. Negli anni '50 si utilizzano anche dischi di grandi dimensioni (fino ad un metro di diametro) composti da un singolo piatto e da testine fisse.

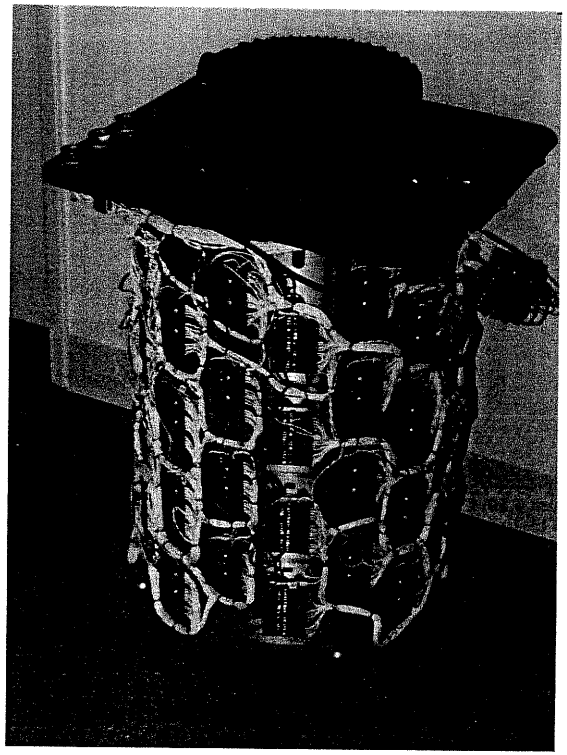


Figura 8.18 Un tamburo magnetico prodotto dalla Digital Development Corporation negli anni '60 e utilizzato in un elaboratore CDC. L'elettronica di supporto alle testine di lettura/scrittura può essere osservata sulla parte esterna del tamburo. Fotografia gentilmente concessa dal Computer Museum of America.

Nel 1956 l'IBM sviluppò a San Josè il primo sistema di memorizzazione basato su dischi, dotato sia di testine mobili che di superfici multiple; in tal modo si diede inizio allo sviluppo dell'industria dei dischi magnetici nella parte sud della Silicon Valley. L'IBM 305 RAMAC (Random Access Method of Accounting and Control, ossia metodo ad accesso casuale per la contabilità ed il controllo) poteva memorizzare 5 milioni di caratteri (5 MB) di dato su 50 dischi, ciascuno dei quali misurava 24 pollici (circa 55 cm) di diametro. Il RAMAC è mostrato nella figura 8.19.

I dischi a testine mobili divennero rapidamente il sistema dominante di memorizzazione magnetica ad alta velocità, anche se il loro elevato costo fece sì che i nastri magnetici continuassero ad essere usati estesamente fino agli anni '70. Il successivo passo chiave nello sviluppo degli hard disk corrispose all'unità hard disk

Figura 8.19 L'unità disco RAMAC dell'IBM prodotta nel 1956 era la prima unità disco con una testina mobile e piatti multipli. Il sito Web sulla tecnologia di memorizzazione IBM contiene una discussione sui più significativi contributi apportati dall'IBM alla tecnologia di memorizzazione. Si veda il link *www.mkp.com/books_catalog/cod/links.htm*. Fotografia gentilmente concessa da IBM.

rimovibile sviluppata da IBM nel 1962; questo rese possibile condividere la costosa circuiteria del controllore e contribuì a far sì che i dischi soppiantassero i nastri quale mezzo di memorizzazione preferito. La figura 8.20 mostra un'unità disco rimovibile ed il disco a più piatti utilizzato nell'unità. L'IBM inventò anche l'unità floppy disk nel 1970, in origine per contenere il microcodice per la serie IBM 370; i floppy disk divennero poi diffusissimi circa dieci anni dopo con i PC.

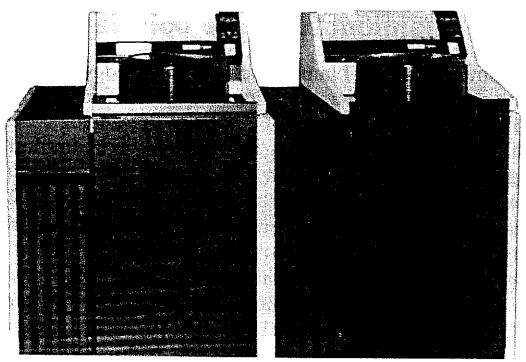


Figura 8.20 Un'unità disco della DEC e l'involucro rimovibile. Questi dischi divennero popolari a cominciare dalla metà degli anni '60 e dominarono la tecnologia dei dischi sino a che non furono introdotte le unità Winchester alla fine degli anni '70. Questa unità fu prodotta a metà degli anni '70. Ciascun disco in questa unità poteva contenere 80 MB. Fotografia gentilmente concessa dal Commercial Computing Museum.

Il disco sigillato Winchester, sviluppato da IBM nel 1973, domina oggi in maniera completa la tecnologia dei dischi: tutti i dischi mostrati nella figura 8.5 sono dischi di tipo Winchester. I dischi Winchester traggono vantaggio da due proprietà tra loro collegate: innanzitutto la riduzione nel costo dell'elettronica del disco rese non più necessaria la condivisione di tale elettronica e quindi economicamente vantaggiosi i dischi non removibili. Poiché il disco era fisso ed era chiuso in un involucro sigillato, i problemi sia ambientali che di controllo venivano significativamente ridotti, permettendo un guadagno elevato in termini di densità. Il primo disco venduto da IBM aveva due assi, ciascuno collegato ad un disco da 30 MB; l'appellativo "30-30" diede origine al nome Winchester. I dischi Winchester crebbero rapidamente in popolarità negli anni '80, rimpiazzando completamente i dischi removibili verso la metà di tale decennio.

Recentemente le unità rimovibili a basso costo sono state riportate in vita per essere usate nelle applicazioni di back-up e portatili. Tali unità sono disponibili di solito sia nella versione floppy disk, che memorizza circa 100 MB nel 1997, sia nella versione hard disk rimovibile, che memorizza 1-2 GB. Questi dischi rimovibili hanno una densità minore e sono più lenti dei dischi non rimovibili, ma i prodotti rimovibili sono particolarmente attraenti in alcuni ambiti.

Gli anni '70 videro l'invenzione di numerosi dispositivi di I/O. Uno di quelli forse più rari fu un dispositivo di memorizzazione su pellicola, che memorizzava con tecnologia ottica i dati su piccole strisce di pellicola fotografica. Tali dispositivi di memorizzazione su pellicola potevano non solo leggere e scrivere la pellicola, ma anche conservare le strisce di pellicola all'interno del dispositivo (che aveva dimensioni pari a circa 150 cm per 120 cm per 90 cm) provvedendo meccanicamente al loro spostamento.

I primi sistemi IBM 360 anticiparono molte delle idee che si usano oggi nei sistemi di I/O. Il 360 fu la prima macchina a fare un uso pesante del DMA ed introdusse il concetto di programmi di I/O che potevano essere interpretati dal dispositivo; una caratteristica chiave era il concatenamento dei programmi di I/O. Il concetto di canale introdotto nel 360 corrisponde all'odierno bus di I/O.

La tendenza per le macchine di classe elevata è stata in seguito quella di utilizzare dei processori di I/O programmabili. Il primo elaboratore a realizzare quest'idea fu il CDC 6600, che usava dei processori di I/O chiamati *processori periferici*.

Il precursore delle attuali workstation e dei personal computer fu Alto, sviluppato nel centro ricerche Xerox di Palo Alto nel 1973 [Thacker et al., 1982], mostrato nella figura 8.21. Questo elaboratore integrava le esigenze delle funzioni di I/O nel microcodice del processore, che includeva il supporto per video grafici a bitmap, dischi e rete. La rete utilizzata da Alto era la prima Ethernet [Metcalfe e Boggs, 1976]. Alto supportava inoltre la prima stampante laser, configurata come un server di stampa accessibile tramite Ethernet; analogamente vennero anche costruiti dei server di disco. Il mouse, inventato in precedenza da Doug Engelbart della SRI, era uno degli elementi chiave di Alto. Il processore a 16 bit utilizzava una memoria di controllo scrivibile, che permetteva ai ricercatori di programmare il supporto per i dispositivi di I/O. Il motore microprogrammato gestiva il video grafico, il mouse, i dischi, la rete e, quando non c'era null'altro da fare, eseguiva il programma utente.

Mentre oggi si associano i microprocessori alla rivoluzione dei personal computer, essi furono originariamente sviluppati per soddisfare le necessità dei controllori dedicati. Dal momento dell'invenzione del microprocessore, i progettisti hanno sviluppato molti controllori di I/O che adattano un microprocessore ad un compito

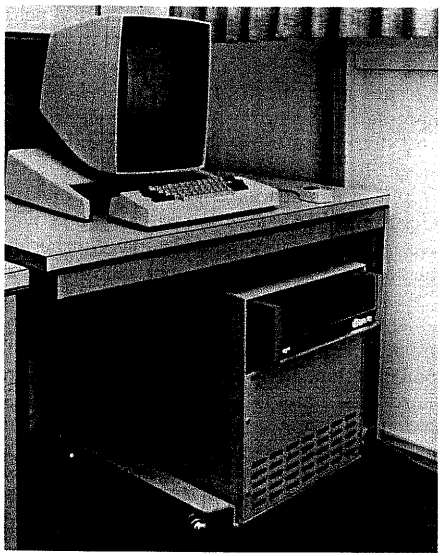


Figura 8.21 Lo Xerox Alto. La Xerox, pur non avendo mai venduto nessuno di questi elaboratori, ne donò un certo numero ad alcune grosse università e li usò internamente in maniera pesante. L'uso del mouse, della rete locale e del video grafico dotato di un sistema a finestre erano le caratteristiche chiave di Alto, successivamente adottate dalle aziende produttrici di workstation e di PC. Fotografia gentilmente concessa dal Computer History Center.

specifico. Tra questi vi sono vari dispositivi, dai controllori di DMA ai controllori SCSI, ai controllori Ethernet completi su singolo chip.

Il primo bus non proprietario potrebbe essere stato l'Unibus del PDP-11 nel 1970. La DEC incoraggiò altre compagnie a costruire dispositivi che potessero connettersi a tale bus e molte società lo fecero. Un esempio più recente è lo SCSI (Small Computer Systems Interface); tale bus, originariamente chiamato SASI, fu inventato da Shugart e fu successivamente standardizzato dalla IEEE. L'approccio aperto ai bus si contrappone ai bus proprietari che usano delle interfacce brevettate, adottate dalle aziende per combattere la concorrenza da parte dei venditori di prodotti compatibili. L'uso di bus proprietari fa inoltre crescere i costi e riduce la disponibilità di dispositivi di I/O che possono connettersi ai bus proprietari, dal momento che tali dispositivi devono avere un'interfaccia progettata esclusivamente per tale bus.

Gli sviluppi in corso nelle aree delle unità a nastro robotizzate (si veda la figura 8.17), video a casco, guanti per fornire sensazioni tattili e schermi di calcolatore scrivibili con apposite penne, lasciano intendere che gli incredibili sviluppi della tecnologia di I/O continueranno ancora nel futuro.

Letture consigliate

Bashe, C.J., L.R. Johnson, J.H. Palmer, e E.W. Pugh [1986]. *IBM's Early Computers*, MIT Press, Cambridge, MA.
Descrive l'architettura ed i dispositivi del sistema di I/O dei primi calcolatori IBM.

Borrill, P.L. [1986]. "32-bit busses: An objective comparison," *Proc. Buscon 1986 West*, San Jose, CA, 138-45.
Un confronto tra i vari standard per i bus a 32 bit.

Chen, P.M., E.K. Lee, J.A. Gibson, R.H. Katz, e D.A. Patterson [1994]. "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys*, 26:2 (Giugno) 145-88.
Un'introduzione sulle schiere di dischi ed i vantaggi di una tale organizzazione.

Grey, J., e A. Reuter [1993]. *Transactions Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco.
Una descrizione dell'elaborazione di transazioni che include una discussione dei problemi campione e della valutazione delle prestazioni.

Hennessy, J., e D. Patterson [1995]. *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, San Francisco, cap. 6-7.
Il Capitolo 6 riguarda i dispositivi di I/O e comprende una discussione approfondita della tecnologia RAID ed un modello più accurato delle prestazioni di I/O. Il Capitolo 7 riguarda le tecnologie di interconnessione e comprende i bus ed un'approfondita discussione sulle reti.

Kahn, R.E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE 60:11* (Novembre) 1397-1407.
Un articolo classico di descrizione di ARPANET.

Levy, J.V. [1978]. "Busses: The skeleton of computer structures," in *Computer Engineering: A DEC View of Hardware Systems Design*, C.G. Bell, J.C. Mudge, e J.E. McNamara, eds., Digital Press, Bedford, MA.
Una buona panoramica dei concetti chiave nel campo del progetto dei bus.

Metcalfe, R.M., e D.R. Boggs [1976]. "Ethernet: distributed packet switching for local computer networks," *Comm. ACM 19:7* (Luglio) 395-404.
Descrive la rete Ethernet.

Smotherman, M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machine," *Computer Architecture News*, 17:5 (Settembre) 5-15.
Descrive lo sviluppo di alcune importanti idee nel campo dell'I/O.

Thacker, C.P., E.M. McCreight, B.W. Lampson, R.F. Sproull, e D.R. Boggs [1982]. "Alto: A personal computer," in *Computer Structures: Principles and Examples*, D.P. Siewiorek, C.G. Bell e A. Newell, eds., McGraw-Hill, New York, 549-72.
Descrive il sistema Alto, il precursore delle workstation e di Macintosh di Apple.

8.11 Parole chiave

L'ampia varietà di caratteristiche presenti nei diversi dispositivi di I/O e le corrispondenti tecniche per adattare il sistema a questi dispositivi hanno determinato l'introduzione di una serie di termini nuovi, elencati di seguito.

accesso diretto in memoria (DMA)
arbitraggio centralizzato/parallelo
arbitraggio del bus
arbitraggio distribuito tramite autoselezione
arbitraggio distribuito tramite rilevamento delle collisioni
arbitraggio in daisy chain
bus asincrono
bus di backplane
bus processore-memoria
bus sincrono
elaborazione delle transazioni
Ethernet
fairness
I/O basato su interruzioni
I/O mappato in memoria
istruzioni di I/O
latenza o ritardo di rotazione
master del bus
polling
protocollo a transazione suddivisa
protocollo di handshaking
richiesta di bus
schiere ridondanti di dischi economici (RAID)
seek
settore
slave
small computer systems interface (SCSI)
tempo di trasferimento
traccia
transazione su bus

8.12 Esercizi

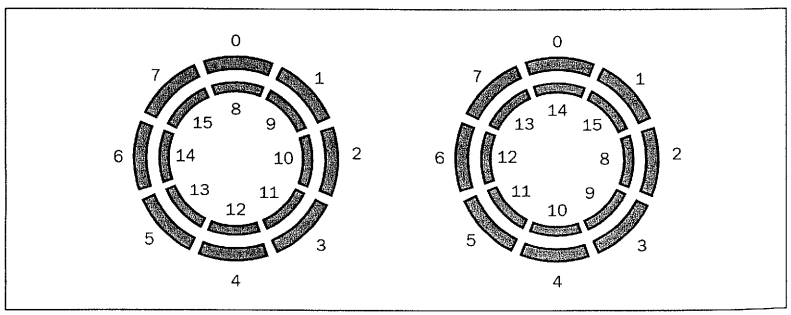
8.1 [10] <§§8.1-8.2> Siano dati due differenti sistemi di I/O per l'elaborazione di transazioni:

- Il sistema A può supportare 1000 operazioni di I/O al secondo
- Il sistema B può supportare 750 operazioni di I/O al secondo

I sistemi utilizzano lo stesso processore che esegue 50 milioni di istruzioni al secondo; si assuma che ogni transazione richieda 5 operazioni di I/O e che ciascuna di esse richieda 10.000 istruzioni. Ignorando il tempo di risposta ed assumendo che le transazioni possano essere sovrapposte in un modo qualsiasi, qual è la massima frequenza di transazioni per secondo che ciascun elaboratore può sostenere?

8.2 [15] <§§8.1-8.2> {Es. 8.1} La latenza di un'operazione di I/O per i due sistemi dell'Esercizio 8.1 è diversa: nel sistema A è pari a 20 ms, mentre nel sistema B è 18 ms per le prime 500 operazioni di I/O in un secondo, e 25 ms per ciascuna operazione di I/O tra la 500-esima e la 750-esima nello stesso secondo. All'interno del carico di lavoro vi è una transazione ogni dieci che dipende da quella immediatamente precedente e che deve attendere il suo completamento. Qual è la massima frequenza di transazioni che permette alle transazioni di essere completate in un secondo e che non eccede l'ampiezza di banda di I/O dell'elaboratore? Per semplicità si assuma che tutte le richieste di transazione arrivino all'inizio di un intervallo di un secondo.

8.3 [5] <§8.3> Il seguente diagramma semplificato mostra due possibili strade per numerare i settori su un disco che contengono i dati (sono mostrate due sole tracce e ciascuna traccia è composta da otto settori). Assumendo che le tipiche operazioni di lettura siano contigue (tutti e sedici i settori sono cioè letti nell'ordine) quale numerazione dei settori ha maggiori probabilità di produrre migliori prestazioni? Perché?



8.4 [5] <§8.3> In corrispondenza di quale dimensione dei messaggi la rete ATM avrà prestazioni migliori di Ethernet di un fattore pari a 2, assumendo valori di latenza e di ampiezza di banda equivalenti a quelli riportati nell'esempio del Paragrafo 8.3?

8.5 [5] <§8.3> La velocità della luce è approssimativamente pari a 3×10^8 metri al secondo e all'interno di un conduttore i segnali elettrici viaggiano ad una velocità pari a circa il 50% di questa. Quando si utilizza il termine *ad alta velocità* riferito ad una rete, ciò che è maggiore è l'ampiezza di banda, non necessariamente la velocità dei segnali elettrici. Quanto varrà il "tempo di volo" reale per i segnali elettrici? Si considerino due calcolatori posti a 100 metri l'uno dall'altro ed altri due posti a 5000 chilometri, e si confrontino i risultati ottenuti con i tempi di latenza riportati nell'esempio del Paragrafo 8.3.

8.6 [5] <§8.3> Il numero di byte che transitano in una rete è definito come il tempo di volo (descritto nell'Esercizio 8.5) moltiplicato per l'ampiezza di banda fornita. Si calcoli il numero di byte che transitano nelle due reti descritte nell'Esercizio 8.5 assumendo un'ampiezza di banda di 5 MB al secondo.

8.7 [5] <§8.3> Un'agenzia segreta di stato controlla contemporaneamente 100 conversazioni su telefoni cellulari e trasferisce i dati su una rete avente un'ampiezza di banda di 1 MB/s ed una latenza aggiuntiva di 350 μs per un messaggio di 1 KB. Si calcoli il tempo di trasmissione per ciascun messaggio e si determini se vi è un'ampiezza di banda sufficiente per supportare una simile applicazione. Si assuma che i dati della conversazione telefonica consistano di due byte campionati alla frequenza di 4 KHz.

8.8 [10] <§8.3> Un programma esegue ripetutamente un processo composto da tre passi: lettura da disco di un blocco di 4 KB di dati, elaborazione di tali dati e scrittura dei risultati su disco sotto forma di un nuovo blocco di 4 KB. Ciascun blocco occupa un'area contigua ed è posizionato in maniera casuale su una singola traccia del disco. L'unità disco ruota a 7200 RPM, ha un tempo medio di seek di 8 ms ed ha una velocità di trasferimento di 20 MB/s; il tempo aggiuntivo richiesto dal controllore è pari a 2 ms. Nessun altro programma utilizza il disco o il processore e non c'è sovrapposizione tra le operazioni su disco e le operazioni di elaborazione; il passo di elaborazione richiede 20 milioni di cicli di clock e la frequenza di clock è 400 MHz. Qual è la velocità complessiva del sistema in termini di numero di blocchi elaborati al secondo?

8.9 [10] <§8.3> Un sistema di elaborazione di transazioni utilizza una rete trasmettendo messaggi di due diverse dimensioni. La richiesta di transazione è relativamente piccola e consiste di un messaggio di 10 byte; il messaggio corrispondente alla risposta ha dimensioni maggiori e consiste di 150 byte. Si assuma che ciascuna transazione sia formata da una richiesta e da una risposta. Si determini quale delle due reti descritte nell'esempio del Paragrafo 8.3 sia più adatta ad un tale sistema.

8.10 [5] <§§8.3, 8.4> Si assuma che il bus ed il sistema di memoria descritti nell'esempio del Paragrafo 8.3 siano usati per gestire degli accessi a dischi come quello descritto nell'esempio del Paragrafo 8.3. Se è permesso che l'I/O consumi il 100% dell'ampiezza di banda del bus e della memoria, qual è il numero massimo di trasferimenti simultanei da disco che possono essere eseguiti con le due dimensioni di blocco?

8.11 [5] <§8.4> L'esempio del Paragrafo 8.3 presuppone che il sistema di memoria impieghi 200 ns a leggere le prime quattro parole e 20 ns a leggere ciascun gruppo successivo di quattro parole. Si riscriva l'esempio facendo l'assunzione che il sistema di memoria impieghi 150 ns a leggere le prime quattro parole e 30 ns a leggere ciascun successivo gruppo di quattro parole.

8.12 [5] <§8.4> L'esempio del Paragrafo 8.3 dimostra che l'uso di blocchi di dimensioni più ampie porta ad un incremento della massima larghezza di banda raggiungibile. In quali condizioni un progettista potrebbe preferire blocchi di dimensioni più piccole? In particolare, perché un progettista dovrebbe scegliere una dimensione di blocco pari a 4 anziché 16, assumendo che tutti gli altri parametri siano quelli dell'esempio?

8.13 [15] <§8.4> Questo esercizio analizza in modo più dettagliato come l'incremento della dimensione del blocco per le transazioni su bus riduca la latenza totale richiesta ed aumenti la massima ampiezza di banda. Nell'esempio del Paragrafo 8.3 sono state prese in considerazione due differenti dimensioni di blocco (4 parole e 16 parole). Si calcoli la latenza totale e la massima ampiezza di banda per tutte le possibili dimensioni dei blocchi comprese tra 4 e 16 e si visualizzino i risultati. Si riassuma che cosa può essere dedotto dal grafico risultante.

8.14 [15] <§8.4> Questo esercizio è simile all'Esercizio 8.13. Questa volta si fissi la dimensione dei blocchi a 4 e a 16 (come nell'esempio del Paragrafo 8.3) ma si calcolino le latenze e le ampiezze di banda per operazioni di lettura eseguite con dimensioni diverse. In particolare si considerino operazioni di dimensione compresa tra 4 a 256 parole e si usi il massimo numero possibile di punti per costruire un grafico significativo. Si usi il grafico per determinare in quale punto le dimensioni di blocco pari a 16 diano luogo ad una latenza ridotta se confrontate con dimensioni di blocco pari a 4.

8.15 [10] <§8.4> Questo esercizio esamina un progetto alternativo a quello dell'esempio del Paragrafo 8.3, in grado di migliorare le prestazioni delle operazioni di scrittura. Per tali operazioni si assuma che i parametri siano gli stessi dell'esempio, e che inoltre:

5. Le prime quattro parole sono scritte 400 ns dopo che è disponibile l'indirizzo, e ciascuna ulteriore operazione di scrittura richiede 20 ns. Si assuma che il trasferimento su bus del dato relativo alla più recente operazione di scrittura possa essere sovrapposto alla scrittura delle precedenti 4 parole.

L'analisi delle prestazioni riportata nell'esempio non viene quindi modificata per le operazioni di scrittura (in realtà possono esistere alcune modifiche marginali dovute alla necessità di calcolare i codici di correzione dell'errore ed altro, ma queste verranno ignorate). Uno schema di bus alternativo si basa su linee separate di indirizzo e di dato di 32 bit; questo schema permette ad un indirizzo e ad un dato di venire trasmessi nel corso dello stesso ciclo di bus. Quale sarà la latenza dell'intera operazione di trasferimento di 256 parole con questo schema alternativo? Quale sarà l'ampiezza di banda ottenibile? Si considerino dimensioni dei blocchi pari a 4 e 8 parole. Quando è pensabile che questo schema alternativo sia significativamente vantaggioso?

8.16 [20] <§8.4> Si consideri un bus asincrono usato per interfacciare un dispositivo di I/O con il sistema di memoria descritto nell'esempio del Paragrafo 8.3. Ogni richiesta di I/O è finalizzata all'ottenimento di 16 parole di dato provenienti dalla memoria; questa, come il dispositivo di I/O, è connessa ad un bus di ampiezza pari a 4 parole. Si assuma di adottare lo stesso tipo di protocollo di handshaking che compare nella figura 8.10, esteso però in maniera tale che la memoria possa continuare la transazione inviando blocchi aggiuntivi di dati finché la transazione è completa. Si modifichi la figura 8.10 (sia i passi che il diagramma) per indicare come tale trasferimento possa aver luogo. Assumendo che ogni passo di handshaking richieda 20 ns e che l'accesso alla memoria richieda 60 ns, quanto tempo sarà necessario per completare un trasferimento? Qual è la massima ampiezza di banda raggiungibile da un tale bus asincrono e come si può confrontare con quella del bus sincrono dell'esempio?

8.17 [15] <§§8.3-8.6> Si ripeta l'esempio del Paragrafo 8.6, ma si assuma che le operazioni di lettura corrispondano a letture casuali di blocchi di 4 KB. Si può assumere che le letture siano sempre eseguite su un disco inattivo, se disponibile.

8.18 [20] <§§8.3-8.6> Nel seguito viene presentato un insieme di blocchi elementari usati in un sistema di I/O basato su un bus sincrono processore-memoria che lavora a 200 MHz e su uno o più adattatori di I/O che interfacciano i bus di I/O con il bus processore-memoria.

- *Sistema di memoria*: il sistema di memoria possiede un'interfaccia di 32 bit e gestisce trasferimenti di 4 parole. Esso possiede linee separate di indirizzo e di dato e per le operazioni di scrittura in memoria accetta una parola ogni ciclo di clock per 4 cicli; in seguito richiede 4 ulteriori cicli per memorizzare le parole prima di poter accettare la transazione successiva.
- *Interfacce DMA*: gli adattatori di I/O usano il DMA per trasferire i dati tra i bus di I/O ed il bus processore-memoria. L'unità di DMA esegue l'arbitraggio relativo al bus processore-memoria ed invia/riceve blocchi di 4 parole dal/al sistema di memoria. Il controllore di DMA può gestire fino ad 8 dischi. L'attivazione di una nuova operazione di I/O (includendo le operazioni di seek e di accesso) richiede 1 ms, durante il quale lo stesso controllore non può iniziare nessun'altra operazione di I/O (possono però essere gestite operazioni urgenti di altro tipo).
- *Bus di I/O*: il bus di I/O è un bus sincrono con un'ampiezza di banda pari a 10 MB/s; ciascun trasferimento è lungo una parola.
- *Dischi*: i dischi hanno una latenza media di seek e rotazionale pari a 12 ms. Durante il trasferimento i dischi presentano un'ampiezza di banda di lettura/scrittura pari a 5 MB/s.

Si calcoli il tempo richiesto per leggere da disco a memoria un settore di 16 KB assumendo che questa sia l'unica attività sul bus.

8.19 [15] <§§8.3-8.5> {Es. 8.18} Per il sistema di I/O descritto nell'Esercizio 8.18, si trovi la massima ampiezza di banda istantanea alla quale i dati possano essere trasferiti dal disco alla memoria utilizzando un numero arbitrario di dischi; quanti dischi e bus di I/O sono necessari per raggiungere una tale ampiezza di banda (si calcoli il valore minimo per ciascuno)? Dal momento che si deve raggiungere questa ampiezza di banda solo per un istante, non è necessario considerare i tempi di latenza.

8.20 [20] <§§8.3-8.5> {Es. 8.18, 8.19} Si assuma che tutti gli accessi eseguiti nell'ambito del sistema di I/O descritto nell'Esercizio 8.18 corrispondano a letture di blocchi di 4 KB. Se vi sono un totale di 6 bus di I/O, sei controllori di DMA e 48 dischi, si trovi il massimo numero di operazioni di I/O che il sistema può tollerare a regime, assumendo che le operazioni di lettura siano uniformemente distribuite tra i dischi. Qual è l'ampiezza della banda di I/O raggiungibile dal sistema?

8.21 [15] <§§8.3-8.5> {Es. 8.18, 8.19, 8.20} Lavorando con il sistema descritto nell'Esercizio 8.20 è chiaramente possibile saturare i 6 bus di I/O, dal momento che questi hanno un'ampiezza di banda pari a 10 MB/s, mentre i 48 dischi hanno un'ampiezza di banda pari a 5 MB/s. Si calcoli la minima dimensione dei blocchi (rimanendo nell'ambito delle potenze di 2) in grado di saturare i bus di I/O. Utilizzando questa dimensione dei blocchi, quante operazioni di I/O al secondo possono essere eseguite dal sistema, e qual è la banda di I/O complessiva?

8.22 [15] <§§7.3, 7.5, 8.4, 8.5> Si consideri una cache che implementi il meccanismo di write-back e che venga utilizzata in combinazione con un processore collegato ad un bus e ad un sotto-sistema di memoria come quelli descritti nell'esempio del Paragrafo 8.3 (si assuma che le operazioni di scrittura richiedano lo stesso tempo richiesto da quelle di lettura). Si assuma inoltre che siano state eseguite le seguenti misure di prestazioni:

- La frequenza delle situazioni di miss della cache è pari a 0,05 miss per istruzione, supponendo che la dimensione dei blocchi sia pari a 8 parole.
- La frequenza delle situazioni di miss della cache è pari a 0,03 miss per istruzione, supponendo che la dimensione dei blocchi sia pari a 16 parole.
- Per entrambe le dimensioni dei blocchi, il 40% delle situazioni di miss richiede un'operazione di write-back, mentre il restante 60% richiede semplicemente una lettura.

Assumendo che il processore entri in stallo per il periodo richiesto da una situazione di miss (includendo il tempo per il write-back se questo è necessario), si calcoli il numero di cicli per istruzione spesi per la gestione delle situazioni di miss per ciascuna dimensione dei blocchi. Suggerimento: si calcoli innanzitutto la penalità conseguente ad una situazione di miss.

8.23 [10] <§8.6> Si scriva un breve commento che identifichi alcune delle assunzioni semplificative fatte nell'analisi riportata nell'esempio del Paragrafo 8.5.

8.24 [2 giorni-1 settimana] <§§8.5, Appendice A> Questo esercizio richiede l'uso di SPIM per la costruzione di un insieme limitato di procedure di I/O in grado di eseguire l'I/O verso un terminale utilizzando la tecnica del polling. Innanzitutto è necessario scrivere due procedure di I/O, le cui dichiarazioni e descrizioni in linguaggio C sono riportate di seguito:

```
void print (char *string);
```

La procedura `print` ha un solo parametro, corrispondente all'indirizzo di una stringa ASCII terminata dal carattere nullo. La procedura `print` deve visualizzare tutti i caratteri della stringa (ad eccezione del carattere di fine stringa). La visualizzazione deve avvenire inviando i caratteri uno alla volta ed aspettando che ciascun carattere sia stato visualizzato prima di inviare quello successivo. La procedura non deve terminare sino a che non sono stati visualizzati tutti i caratteri. La procedura deve lavorare con stringhe di lunghezza qualsiasi. La versione qui considerata di `print` non deve fare uso delle interruzioni, ma soltanto controllare continuamente il bit di pronto (ready bit) del registro di controllo del dispositivo trasmettitore sino a che il dispositivo non diventa libero.

```
char getchar ();
```

La procedura `getchar` non ha parametri e restituisce come risultato un carattere. La procedura attende sino a che non viene introdotto un carattere da terminale e restituisce poi il valore del carattere nel registro `$v0` (il registro risultato). Non si usino interruzioni: ci si limiti a controllare il bit di pronto sino a che non arriva un carattere.

Si scriva un programma principale che usi le due procedure per leggere una linea dal terminale, terminata da un carattere di "a capo"; si visualizzi poi l'intera linea sul terminale, includendo un "a capo". Il codice deve soddisfare le convenzioni dell'Appendice A per la chiamata di procedure, l'uso dello stack e dei registri.

8.25 [3 giorni-1 settimana] <§§8.5, Appendice A> L'esercizio consiste nella definizione di un meccanismo basato sulle interruzioni per eseguire operazioni di I/O con buffer da e verso un terminale. Questo esercizio in realtà gestisce il solo output, mentre l'Esercizio 8.26 gestisce l'input.

Per quanto riguarda l'output, il programma si compone di tre parti:

1. Un programma principale, che chiama ripetutamente la procedura `print` per visualizzare la stringa "`So cio' che sto facendo`".
2. La procedura `print`, che memorizza il carattere di output in un buffer condiviso con la procedura di interruzione.
3. La procedura di interruzione che copia i caratteri dal buffer di output al dispositivo trasmettitore.

La procedura `print` e la procedura di interruzione devono comunicare utilizzando un buffer circolare condiviso, capace di contenere 32 caratteri. La procedura `print` riceve come parametro una stringa ed aggiunge uno ad uno i caratteri della stringa al buffer di uscita, procedendo non appena vi è spazio nel buffer. Si ricordi che la procedura `print` non deve interagire direttamente con i registri del dispositivo periferico eccetto che per verificare che le interruzioni del dispositivo trasmettitore siano abilitate. Inoltre la procedura `print` deve prevedere del codice aggiuntivo per gestire la situazione in cui il buffer di uscita è pieno; il programma principale genera i caratteri molto più velocemente di quanto essi non possano essere visualizzati, così che il buffer verrà rapidamente riempito. In un sistema reale se il buffer di uscita si riempie, il sistema operativo arresta il processo utente correntemente in esecuzione ed attiva un altro processo. Il programma non deve necessariamente supportare più utenti, per cui la procedura `print` può seguire un approccio più semplice: essa si limita a verificare continuamente il buffer sino a che questo non è nuovamente pieno. Il buffer è pieno quando la posizione successiva nella quale la procedura `print` deve inserire un carattere non è ancora stata svuotata dalla procedura di interruzione.

Dopo aver scritto la procedura di visualizzazione, si scriva la procedura di interruzione chiamata dalla procedura `print`. Tra le operazioni che la procedura di interruzione deve eseguire vi sono le seguenti:

1. Se il dispositivo trasmettitore non è pronto, allora la procedura di interruzione non deve fare nulla. Non si dovrebbe ricevere un'interruzione se il dispositivo trasmettitore non è pronto, ma è comunque opportuno eseguire un controllo.
2. Se il buffer di output non è vuoto, si copi il carattere successivo dal buffer di output nel registro di dato del dispositivo trasmettitore e si aggiornino i puntatori del buffer.
3. Se il buffer di output è vuoto, si disabilitino le interruzioni nel registro di controllo del dispositivo trasmettitore (altrimenti continuerebbero ad arrivare interruzioni). Ogni volta che scrive un carattere nel buffer, la procedura `print` deve abilitare le interruzioni.

4. Non si dimentichi che è necessario salvare e ripristinare tutti i registri che vengono usati nella procedura di interruzione, compresi i registri temporanei come `$t0` e `$t1`. Questa operazione è necessaria dal momento che le interruzioni possono verificarsi in un qualunque momento e questi registri possono quindi essere in uso nel momento in cui si verifica l'interruzione; i registri devono essere salvati nello stack. Le uniche eccezioni a questa regola sono costituite dai registri `$k0` e `$k1`, che sono destinati esclusivamente alle procedure di interruzione, e che quindi non devono essere salvati e ripristinati. Uno di questi registri può essere usato per eseguire il ritorno dalla procedura di interruzione al codice che era stato interrotto.

Si controlli la correttezza del codice scrivendo una procedura principale che chiami la procedura `print` per visualizzare la stringa; questa deve visualizzare una dopo l'altra delle linee contenenti ciascuna la stringa `"So cio' che sto facendo"`.

8.26 [3 giorni-1 settimana] <§§8.5, Appendice A> {Es. 8.25} Si estenda il codice già scritto in modo che possa gestire un input tramite interruzioni. Il codice dovrebbe eseguire l'input nella stessa maniera in cui il precedente programma gestiva l'output: deve quindi utilizzare un buffer per la comunicazione tra la procedura *getchar* e la procedura di interruzione. Si noti che la procedura *getchar* estrae un carattere dal buffer e lo restituisce, aspettando in un ciclo se non vi sono caratteri disponibili. Analogamente la procedura di interruzione aggiunge caratteri al buffer mano a mano che questi sono inseriti, perdendo dei caratteri se il buffer è pieno quando questi giungono. Per questa ragione un buffer di 8 elementi dovrebbe funzionare adeguatamente.

Si usino le due procedure per leggere e visualizzare caratteri sul terminale. Si provi a digitare velocemente i caratteri per accertarsi che il programma possa gestire correttamente il riempimento del buffer di output e di quello di input. Se per esempio si digitano rapidamente due o tre caratteri, il buffer di output potrebbe riempirsi, ma non deve venir perso nessun carattere. La procedura di visualizzazione dovrà semplicemente darsi da fare per un po' e durante questo tempo gli eventuali caratteri di input verranno memorizzati nel buffer di input. Se si digitano molto rapidamente 8 o 10 caratteri, allora probabilmente il buffer di input verrà riempito. Quando si verifica questa situazione, la procedura di interruzione dovrà probabilmente scartare dei caratteri: il programma continuerà a funzionare, ma non vi sarà nessuna visualizzazione dei caratteri di input introdotti ma scartati. Una volta che l'output si è nuovamente messo in pari con l'input, il programma accetta nuovamente l'input come se il buffer di input non si fosse mai riempito.

8.27 [1 giorno-1 settimana] <§§8.2-8.4> Prendendo come riferimento il proprio calcolatore preferito, si scrivano dei programmi in grado di raggiungere i seguenti obiettivi:

1. Massima ampiezza di banda da e verso un singolo disco;
2. Massima ampiezza di banda da e verso dischi multipli;
3. Massimo numero di transazioni relative a blocchi di 512 byte da e verso un singolo disco;
4. Massimo numero di transazioni relative a blocchi di 512 byte da e verso dischi multipli.

Qual è la percentuale di ampiezza di banda che si può ottenere rispetto a quella dichiarata dal costruttore del dispositivo di I/O? Si memorizzi inoltre l'utilizzo del processore nel caso in cui ciascuno dei programmi lavori da solo. Successivamente si facciano lavorare i quattro programmi contemporaneamente, osservando quale sia la percentuale dei vari parametri che può essere raggiunta (rispetto ai valori massimi). Sulla base di queste misure è possibile determinare dove si trovi il collo di bottiglia del sistema?

In maggior dettaglio

Ethernet

Ethernet è essenzialmente un bus standard con più master (ciascun calcolatore può essere un master) ed uno schema di arbitraggio distribuito basato sul rilevamento delle collisioni. La maggior parte delle reti Ethernet utilizza come supporto un cavo coassiale. Quando un certo nodo desidera utilizzare il bus, verifica innanzitutto se qualche altro nodo lo sta già utilizzando; in caso negativo invia il proprio segnale sul bus ed inizia a trasmettere. Può nascere qualche complicazione dal fatto che il controllo è distribuito ed i dispositivi possono trovarsi in posizioni fisicamente distanti tra loro. Come risultato è possibile che due nodi verifichino insieme lo stato della rete, entrambi la trovino libera ed inizino insieme a trasmettere; una simile situazione viene definita *collisione*. Un nodo rileva le collisioni osservando la rete mentre sta trasmettendo: quando un nodo si accorge che ciò che osserva sulla rete è diverso da ciò che sta trasmettendo, allora si è verificata una collisione. In questo caso entrambi i nodi smettono di trasmettere ed attendono per un tempo di durata casuale prima di riprovare ad usare la rete, in maniera simile al modo in cui si comportano due persone educate quando entrambe iniziano a parlare contemporaneamente. Di conseguenza il numero dei nodi sulla rete deve essere limitato, in quanto le prestazioni peggiorano se vi sono troppe collisioni. Inoltre i vincoli imposti dall'esigenza che le collisioni siano rilevate da tutti i nodi limita la lunghezza di una rete Ethernet, nonché il numero di connessioni alla rete. Benché l'idea sembri non poter funzionare, in realtà lo fa sorprendentemente bene ed ha contribuito all'enorme crescita che si è verificata nel campo delle reti locali.

8.28 [3 giorni-1 settimana] <§§8.3-8.4> Si scriva un programma che simuli una rete Ethernet, assumendo che il sistema di interconnessione abbia le seguenti caratteristiche:

- Un'ampiezza di banda per la trasmissione pari a 10 Mbit/s.
- Una latenza per un segnale che debba attraversare l'intera lunghezza della rete e ritornare nel punto di origine pari a 15 μs. Lo stesso valore corrisponde anche al tempo richiesto per il rilevamento di una collisione.

Si facciano inoltre le seguenti assunzioni relativamente ai 100 nodi esistenti nella rete:

- La dimensione dei pacchetti è pari a 1000 byte.
- Ciascun nodo tenta di inviare un pacchetto dopo T secondi di elaborazione, dove T è una variabile casuale con distribuzione esponenziale e valor medio M. Si noti che il nodo inizia i suoi T secondi di elaborazione solo dopo aver trasmesso con successo un pacchetto.
- Se viene rilevata una collisione, il nodo attende per un periodo di tempo di durata casuale scelto utilizzando una distribuzione esponenziale con valor medio pari a 60 μs.

Si effettui una simulazione e si riporti graficamente l'ampiezza di banda raggiunta a regime dalla rete confrontandola con il valor medio della durata dell'intervallo tra due tentativi di trasmissione (M). Si riporti inoltre nel grafico il tempo medio di attesa tra un tentativo di inizio di una trasmissione e il momento in cui essa inizia veramente (confrontando tale valore con M).

Le reti Ethernet in realtà utilizzano un algoritmo di tipo esponenziale che incrementa il valor medio del tempo di attesa dopo ciascuna collisione. Si assuma che il valor medio della distribuzione sulla base della quale il nodo sceglie quanto deve valere il tempo di attesa venga raddoppiato dopo due collisioni consecutive. Quanto funziona bene un simile meccanismo? L'ampiezza di banda è più elevata rispetto al caso in cui si usa una sola distribuzione? È possibile che il valor medio iniziale abbia un valore minore?

In maggior dettaglio

Schiere di dischi

Come illustrato nel Paragrafo 8.9, uno dei modi in cui è possibile organizzare dei sistemi di dischi consiste nell'usare delle schiere di piccoli dischi in grado di fornire un'ampiezza di banda maggiore grazie all'accesso parallelo. Nella maggior parte delle schiere di dischi tutti gli assi sono sincronizzati (quindi il settore 0 di tutti i dischi si trova sotto la testina esattamente nello stesso istante) e i bracci che sostengono le varie testine sono sempre al di sopra della stessa traccia. Inoltre i dati sono suddivisi tra i dischi della schiera, così che settori consecutivi possono essere letti in parallelo. Ora si esploreranno le modalità di funzionamento di un simile sistema.

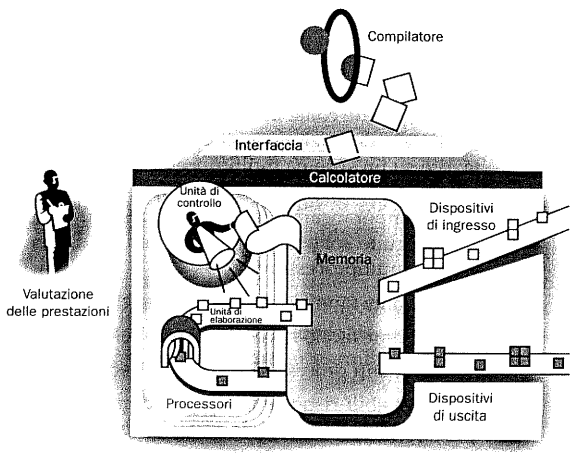
8.29 [20] <§§8.3-8.5> Si assuma di avere le seguenti configurazioni per due dischi magnetici: da un lato un disco singolo, dall'altro una schiera di quattro dischi. Ciascun disco ha 64 settori per traccia, ciascun settore contiene 1000 byte ed il disco ruota a 7200 RPM; si assuma che il tempo di seek sia pari a 6 ms. Il ritardo introdotto dal controllore del disco è pari ad 1 ms per ciascuna transazione, sia essa relativa ad un singolo disco o ad una schiera. Si assuma che le prestazioni del sistema di I/O siano limitate solo dai dischi e dal controllore. Si ricordi che settori consecutivi in un sistema a disco singolo verranno distribuiti ciascuno su un diverso disco della schiera. Si confrontino le prestazioni in termini di operazioni di I/O al secondo di ciascuna delle due organizzazioni di dischi, assumendo che le richieste corrispondano a letture casuali, metà delle quali di dimensione pari a 4 KB e metà pari a 16 KB di dati, sempre a partire da settori consecutivi. I settori possono essere letti in un qualsiasi ordine; per semplicità si assuma che la latenza rotazionale sia pari a metà del tempo per eseguire una singola rotazione nel caso di una lettura di 16 settori da un singolo disco e in quello di una lettura di 4 settori da una schiera di dischi. Problema: è possibile dedurre l'effettiva latenza rotazionale media nei due casi?

8.30 [10] <§§8.3-8.5> {Es. 8.29} Usando gli stessi sistemi di dischi dell'Esercizio 8.29, con le stesse configurazioni di accesso, si determinino le prestazioni in megabyte al secondo per ciascun sistema.

9

I sistemi multiprocessore

Le cinque componenti classiche del calcolatore



Nel mare ci sono pesci più belli di quanti ne siano mai stati pescati.

Proverbio irlandese

Si effettui una simulazione e si riporti graficamente l'ampiezza di banda raggiunta a regime dalla rete confrontandola con il valor medio della durata dell'intervallo tra due tentativi di trasmissione (M). Si riporti inoltre nel grafico il tempo medio di attesa tra un tentativo di inizio di una trasmissione e il momento in cui essa inizia veramente (confrontando tale valore con M).

Le reti Ethernet in realtà utilizzano un algoritmo di tipo esponenziale che incrementa il valor medio del tempo di attesa dopo ciascuna collisione. Si assuma che il valor medio della distribuzione sulla base della quale il nodo sceglie quanto deve valere il tempo di attesa venga raddoppiato dopo due collisioni consecutive. Quanto funziona bene un simile meccanismo? L'ampiezza di banda è più elevata rispetto al caso in cui si usa una sola distribuzione? È possibile che il valor medio iniziale abbia un valore minore?

In maggior dettaglio

Schiere di dischi

Come illustrato nel Paragrafo 8.9, uno dei modi in cui è possibile organizzare dei sistemi di dischi consiste nell'usare delle schiere di piccoli dischi in grado di fornire un'ampiezza di banda maggiore grazie all'accesso parallelo. Nella maggior parte delle schiere di dischi tutti gli assi sono sincronizzati (quindi il settore 0 di tutti i dischi si trova sotto la testina esattamente nello stesso istante) e i bracci che sostengono le varie testine sono sempre al di sopra della stessa traccia. Inoltre i dati sono suddivisi tra i dischi della schiera, così che settori consecutivi possono essere letti in parallelo. Ora si esploreranno le modalità di funzionamento di un simile sistema.

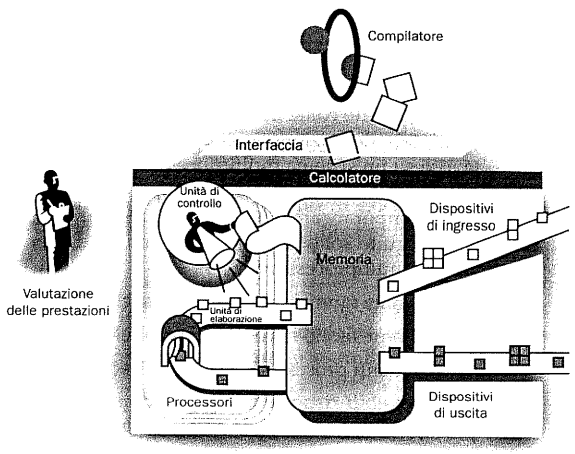
8.29 [20] <§§8.3-8.5> Si assuma di avere le seguenti configurazioni per due dischi magnetici: da un lato un disco singolo, dall'altro una schiera di quattro dischi. Ciascun disco ha 64 settori per traccia, ciascun settore contiene 1000 byte ed il disco ruota a 7200 RPM; si assuma che il tempo di seek sia pari a 6 ms. Il ritardo introdotto dal controllore del disco è pari ad 1 ms per ciascuna transazione, sia essa relativa ad un singolo disco o ad una schiera. Si assuma che le prestazioni del sistema di I/O siano limitate solo dai dischi e dal controllore. Si ricordi che settori consecutivi in un sistema a disco singolo verranno distribuiti ciascuno su un diverso disco della schiera. Si confrontino le prestazioni in termini di operazioni di I/O al secondo di ciascuna delle due organizzazioni di dischi, assumendo che le richieste corrispondano a letture casuali, metà delle quali di dimensione pari a 4 KB e metà pari a 16 KB di dati, sempre a partire da settori consecutivi. I settori possono essere letti in un qualsiasi ordine; per semplicità si assuma che la latenza rotazionale sia pari a metà del tempo per eseguire una singola rotazione nel caso di una lettura di 16 settori da un singolo disco e in quello di una lettura di 4 settori da una schiera di dischi. Problema: è possibile dedurre l'effettiva latenza rotazionale media nei due casi?

8.30 [10] <§§8.3-8.5> {Es. 8.29} Usando gli stessi sistemi di dischi dell'Esercizio 8.29, con le stesse configurazioni di accesso, si determinino le prestazioni in megabyte al secondo per ciascun sistema.

9

I sistemi multiprocessore

Le cinque componenti classiche del calcolatore



Nel mare ci sono pesci più belli di quanti ne siano mai stati pescati.

Proverbio irlandese

9.1 Introduzione

"Sulle Montagne
della Luna
giù nella Valle dell'Ombra
cavalca, cavalca impavido"
Replicò l'ombra, -
"Se cerchi l'Eldorado!"

Edgar Allan Poe, "Eldorado," stanza 4, 1849

I progettisti di calcolatori hanno sempre pensato che l'Eldorado della progettazione si potesse raggiungere creando calcolatori potenti tramite la semplice connessione di molti calcolatori più piccoli; questa visione ideale è alla base dei sistemi multiprocessore. Il cliente ordina tanti processori quanti sono compatibili con le proprie finanze, ed in cambio riceve prestazioni proporzionali all'investimento. I sistemi multiprocessore devono quindi essere scalabili: hardware e software sono progettati in modo da poter essere adattati ad un numero variabile di processori, che nel caso di alcuni calcolatori variano di un fattore maggiore di 50. Dato che il software è scalabile, alcuni sistemi multiprocessore possono funzionare anche in presenza di un malfunzionamento hardware: se solo uno degli n processori di un sistema multiprocessore si guasta, il sistema continua a funzionare con $n - 1$ processori. Infine i sistemi multiprocessore possono raggiungere le migliori prestazioni in assoluto, ed essere più veloci del più veloce sistema a processore singolo.

I sistemi multiprocessore costituiscono ormai un mercato ben assestato. Tenendo conto che questi al momento sono i sistemi con il miglior rapporto costo–prestazioni, in generale si concorda sul fatto che non è possibile gestire un carico di lavoro a suddivisione di tempo (in timesharing) su un processore a singolo chip: costruire un sistema multiprocessore composto da molti processori a singolo chip è più efficiente che costruire un processore singolo ad alte prestazioni con tecnologie più "stravaganti". Tutti i file server attuali sono realizzabili come sistemi multiprocessore, e l'industria delle basi dati si è rivolta a sistemi di questo tipo: di conseguenza i sistemi multiprocessore oggi rappresentano un mercato significativo.

Per i sistemi multiprocessore commerciali prestazioni elevate vengono solitamente definite come un elevato throughput per lo svolgimento di compiti indipendenti; tale definizione è all'estremo opposto rispetto all'esecuzione di un singolo lavoro su un sistema multiprocessore. Nel libro si userà il termine *programma in elaborazione parallela*, o *programma parallelo*, per indicare un singolo programma eseguito contemporaneamente su più processori.

Di seguito sono elencati i problemi fondamentali che influenzano il progetto dei sistemi multiprocessore:

- Come fanno i processori paralleli a condividere i dati?
- Come fanno i processori paralleli ad agire in modo coordinato?
- Quanti processori occorre prevedere?

Le possibili risposte alla prima domanda ricadono in due categorie principali. I processori con un *singolo spazio di indirizzamento*, talvolta chiamati *processori con memoria condivisa*, offrono al programmatore un singolo spazio di indirizzamento che tutti i processori condividono. I processori comunicano attraverso variabili condivise che si trovano in memoria, e tutti i processori sono in grado di accedere con le istruzioni di load e di store a qualsiasi locazione di memoria.

Dato che processori che operano in parallelo normalmente condividono i dati, hanno anche bisogno di coordinare le proprie azioni quando lavorano sui dati condivisi, altrimenti un processore potrebbe iniziare a utilizzare un dato prima che un altro abbia finito di operare su di esso. Questo tipo di coordinamento è detto *sincronizzazione*. Quando esiste un singolo spazio di indirizzamento, deve esistere un meccanismo di sincronizzazione separato; uno degli approcci possibili si basa sull'utilizzo di tecniche di *lock* (letteralmente serratura, blocco): solo un processore per volta può acquisire il "lock" e gli altri processori interessati ai dati condivisi devono aspettare che il processore che ha il controllo del lock rilasci la variabile bloccata; questo meccanismo è descritto nel Paragrafo 9.3.

I sistemi multiprocessore con singolo spazio di indirizzamento seguono due modelli diversi. Il primo di questi modelli prevede che venga impiegato lo stesso tempo per accedere alla memoria principale, indipendentemente da quale processore faccia la richiesta e da quale parola venga indirizzata; questi calcolatori sono detti sistemi multiprocessore *con accesso uniforme alla memoria* (*UMA*, Uniform Memory Access) o *sistemi multiprocessore simmetrici* (*SMP*, Symmetric Multiprocessors). Nel caso del secondo modello alcuni accessi alla memoria sono più veloci di altri, a seconda del processore e a seconda della parola richiesta; questi calcolatori sono detti sistemi multiprocessore *con accesso non uniforme alla memoria* (*NUMA*, NonUniform Memory Access). Come ci si può aspettare, programmare un sistema multiprocessore NUMA in modo da ottenere le migliori prestazioni possibili rappresenta una sfida più difficile rispetto al caso dei sistemi UMA, ma i primi sono scalabili fino a dimensioni maggiori, e quindi potenzialmente consentono prestazioni più elevate.

In alternativa alla memoria condivisa, un altro modello per consentire ai processori di comunicare fra loro è lo *scambio di messaggi*, richiesto nel caso di calcolatori dotati di *memorie private*. Come esempio limite, i processori all'interno di Personal Computer distinti comunicano fra loro scambiandosi messaggi su una rete locale. Dando per scontato che il sistema abbia delle procedure per *spedire* e *ricevere* messaggi, il coordinamento è raggiunto con lo scambio di messaggi: ogni processore sa quando un messaggio viene spedito, ed il processore che deve riceverlo sa quando arriva il suo messaggio. Il processore ricevente può restituire un messaggio al trasmettitore comunicandogli che il messaggio è giunto a destinazione, se quest'ultimo ha bisogno della conferma.

Un fenomeno recente è stato quello di cercare di portare all'estremo l'esempio precedente (calcolatori connessi in rete locale) per farli lavorare come se fossero un grande sistema multiprocessore. Questi *cluster* (gruppi) di calcolatori sfruttano le reti locali per ottenere un'elevata ampiezza della banda di comunicazione fra i calcolatori del cluster stesso.

Oltre ai due principali modelli di comunicazione, i sistemi multiprocessore possono seguire due modelli base di organizzazione: a bus singolo o connessi via rete; il numero di processori all'interno di un sistema multiprocessore è molto legato a questa scelta. I due stili di organizzazione saranno esaminati in dettaglio nei Paragrafi 9.3 e 9.4.

Categoria	Scelta		Numero di processori
Modello di comunicazione	Scambio di messaggi		8-256
	Indirizzamento condiviso	NUMA	8-256
		UMA	2-64
Connessione fisica	Rete		8-256
	Bus		2-32

Figura 9.1 Le scelte sullo stile di comunicazione e sul tipo di connessione fisica nei sistemi multiprocessore al variare del numero di processori. Si noti che lo spazio di indirizzamento condiviso è suddiviso in accesso alla memoria uniforme (UMA) e accesso alla memoria non uniforme (NUMA).

Quadro d'insieme

La figura 9.1 mostra la relazione fra il numero dei processori in un sistema multiprocessore, la scelta fra lo spazio di indirizzamento condiviso rispetto alla comunicazione con scambio di messaggi, e la scelta fra la connessione fisica via bus o via rete. Nel caso dello spazio di indirizzamento condiviso c'è un ulteriore suddivisione fra accesso alla memoria uniforme o non uniforme. Nonostante ci sia ampia possibilità di scelta per alcuni valori del numero di processori, in altre zone della tabella la scelta è praticamente obbligata.

Una delle principali difficoltà nello scrivere questo libro è stata quella di presentare lo stato dell'arte più recente in un settore in cui l'industria cambia rapidamente. In questo capitolo si è deciso di fornire dei puntatori ad esempi reali mediante World Wide Web: per trovare le informazioni più aggiornate sui calcolatori che esemplificano le idee esposte, si consiglia di consultare la parte relativa al Capitolo 9 all'indirizzo *www.mkp.com/cod2e.htm*.

Nel seguito si inizierà trattando i concetti generali della programmazione dei sistemi multiprocessore.

9.2 La programmazione dei sistemi multiprocessore

Sovente si sente parlare con criticità del caso in cui calcolatori molto veloci ... restano "disoccupati". Occorre considerare che ... [nel passato] la complessità dei problemi affrontati dipendeva dalla velocità delle macchine allora disponibili. ... Nel caso di macchine più veloci, lo stesso principio porterà ad affrontare problemi di grandi dimensioni.

John von Neumann, presentazione durante un seminario IBM
sul calcolo scientifico, novembre 1949

Quali applicazioni davvero importanti vengono eseguite più velocemente sui sistemi multiprocessore grazie all'elaborazione parallela? La risposta non è così ovvia. L'ostacolo non è il prezzo del singolo processore usato per costruire il sistema multiprocessore, o i difetti nella topologia di interconnessione della rete, e neppure la non disponibilità di linguaggi di programmazione adeguati; la difficoltà consiste

nel fatto che poche applicazioni importanti sono state riscritte per essere eseguite su un sistema multiprocessore. Inoltre poiché è difficile trovare applicazioni che traggano vantaggio dall'uso di un gran numero di processori, nel caso di sistemi multiprocessore di grandi dimensioni la sfida è ancora maggiore.

Per via della difficoltà di programmazione, dietro la maggior parte dei successi relativi all'uso dell'elaborazione parallela c'è il lavoro di maghi del software che sviluppano sistemi paralleli con un'interfaccia sequenziale. Gli esempi includono basi di dati, file server, pacchetti software per la progettazione assistita da calcolatore e sistemi operativi multiprocessore.

Ma perché avviene questo? Perché i programmi che sfruttano l'elaborazione parallela sono molto più difficili da sviluppare dei programmi sequenziali?

La prima ragione è che *bisogna* ottenere un buon livello di prestazioni e di efficienza dai programmi paralleli eseguiti su un sistema multiprocessore, altrimenti tanto varrebbe usare un solo processore, dato che in quel caso la programmazione è più semplice. Infatti le tecniche di progettazione dei processori singoli, come l'esecuzione superscalare e non sequenziale, traggono vantaggio dal parallelismo a livello delle istruzioni, solitamente senza alcun intervento da parte del programmatore; questa innovazione riduce la necessità di riscrivere i programmi per adattarli ai sistemi multiprocessore.

Perché è difficile scrivere programmi veloci per i sistemi multiprocessore, specialmente se il numero di processori cresce? Come analogia si pensi al lavoro aggiuntivo dovuto alla comunicazione nel caso di un compito svolto da una persona, e lo si confronti con il caso di un compito svolto da un gruppo di lavoro, specialmente quando cresce il numero dei componenti. Benché n persone abbiano le potenzialità di terminare un dato compito n volte più in fretta, il lavoro aggiuntivo dovuto alla comunicazione all'interno del gruppo può far sì che questo non sia vero; un aumento di velocità pari a n diventa particolarmente improbabile al crescere di n (si immagini il variare del carico di lavoro aggiuntivo dovuto alla comunicazione quanto il numero dei componenti di un gruppo di lavoro passa da 10 a 1000 a 1000000).

Un'altra ragione per la quale è difficile scrivere programmi che sfruttino l'elaborazione parallela è che il programmatore deve conoscere l'hardware davvero molto bene. Nel caso di un singolo processore, il programmatore ad alto livello può scrivere i suoi programmi ignorando in gran parte l'organizzazione della macchina sottostante, dato che quello è il compito del compilatore. Invece, almeno per il momento, per poter sviluppare applicazioni veloci ed in grado di adattarsi ad un numero variabile di processori, chi scrive programmi per l'elaborazione parallela deve necessariamente conoscere l'organizzazione della macchina. In più questi programmi paralleli, scritti appositamente per una macchina, non sono portabili su altri sistemi multiprocessore.

Benché questo secondo limite perda via via di importanza, la discussione svolta nel Capitolo 2 evidenzia una terza difficoltà insita nella legge di Amdahl; essa ricorda che anche le più piccole parti del programma devono essere ottimizzate per l'elaborazione parallela perché possa essere raggiunta pienamente la potenzialità di questo approccio. Avvicinarsi ad un aumento lineare della velocità di esecuzione implica quindi la scoperta di nuovi algoritmi intrinsecamente paralleli.

La sfida dell'aumento di velocità (speedup)

Esempio Si supponga di voler raggiungere un aumento lineare della velocità di esecuzione nel caso di 100 processori. Quanta parte dell'elaborazione originaria può rimanere sequenziale?

Soluzione La legge di Amdahl (Capitolo 2) dice che

$$\text{tempo di esecuzione dopo il miglioramento} = \frac{\begin{pmatrix}\text{tempo di esecuzione influenzato}\\ \text{dal miglioramento}\end{pmatrix}}{\text{ammontare del miglioramento}} + \text{tempo di esecuzione non influenzato}$$

Avendo come obiettivo un incremento di velocità lineare nel caso di 100 processori, il tempo di esecuzione deve essere ridotto di un fattore pari a 100:

$$\frac{\text{tempo di esecuzione dopo il miglioramento}}{100} = \frac{\begin{pmatrix}\text{tempo di esecuzione influenzato}\\ \text{dal miglioramento}\end{pmatrix}}{100} + \text{tempo di esecuzione non influenzato}$$

Risolvendo l'equazione per ottenere il tempo di esecuzione non influenzato si ha

$$\text{tempo di esecuzione non influenzato} = \frac{\begin{pmatrix}\text{tempo di esecuzione}\\ \text{dopo il miglioramento}\end{pmatrix}}{100} + - \frac{\text{tempo di esecuzione influenzato dal miglioramento}}{100} = 0$$

Di conseguenza, per raggiungere un incremento di velocità lineare nel caso di 100 processori *nessuna* parte dell'elaborazione originaria può essere sequenziale. In altre parole, un incremento di velocità pari a 99 con 100 processori significa che la percentuale del programma originale che può rimanere sequenziale deve essere inferiore allo 0,01%.

Nonostante ciò esistono applicazioni che sfruttano il parallelismo in modo sostanziale.

La sfida dell'aumento di velocità (speedup) in un caso più complesso

Esempio Si supponga di voler eseguire due somme: una somma di due variabili scalari ed una somma matriciale di una coppia di vettori bidimensionali, di dimensione 1000 per 1000. Quale incremento di velocità si può ottenere con 1000 processori?

Soluzione Assumendo che le prestazioni siano una funzione del tempo t impiegato per eseguire una somma, allora c'è una somma che non trae beneficio dall'elaborazione parallela, e 1000000 somme che ne traggono vantaggio. Se il tempo inizialmente era pari a 1000001 t

$$\text{tempo di esecuzione dopo il miglioramento} = \frac{\begin{pmatrix}\text{tempo di esecuzione influenzato}\\ \text{dal miglioramento}\end{pmatrix}}{\text{ammontare del miglioramento}} + \text{tempo di esecuzione non influenzato}$$

$$\text{tempo di esecuzione dopo il miglioramento} = \frac{1000000\,t}{1000} + 1t = 1001\,t$$

L'incremento di velocità è quindi pari a

$$\text{incremento di velocità} = \frac{1000001\,t}{1001\,t} = 999$$

Se anche la parte sequenziale fosse estesa a 100 somme di variabili scalari rispetto ad una somma di due vettori 1000 per 1000, l'incremento di velocità sarebbe pari a 909.

9.3 Sistemi multiprocessore a bus singolo

Negli anni '80 le alte prestazioni dei microprocessori ed il loro basso costo ispirarono un rinnovato interesse nei loro confronti. Poteva essere utile sistemare più microprocessori su un bus comune per diverse ragioni:

- Ciascun microprocessore è molto più piccolo di un processore complesso realizzato su più chip, per cui su un bus possono essere sistemati più processori.
- L'uso delle cache può diminuire la quantità di traffico sul bus.
- Furono inventati dei meccanismi per mantenere coerenti cache e memoria nel caso di sistemi multiprocessore, allo stesso modo in cui cache e memoria sono mantenuti coerenti per l'I/O, semplificando quindi la programmazione.

La figura 9.2 mostra lo schema di un generico sistema multiprocessore a bus singolo, e la figura 9.3 elenca le caratteristiche di alcuni calcolatori a bus singolo in commercio.

Il traffico per processore e l'ampiezza di banda del bus determinano il numero utile di processori in un sistema multiprocessore di questo tipo. Le cache replicano i dati al loro interno *sia* per ridurne la latenza *che* per ridurre il traffico di memoria sul bus.

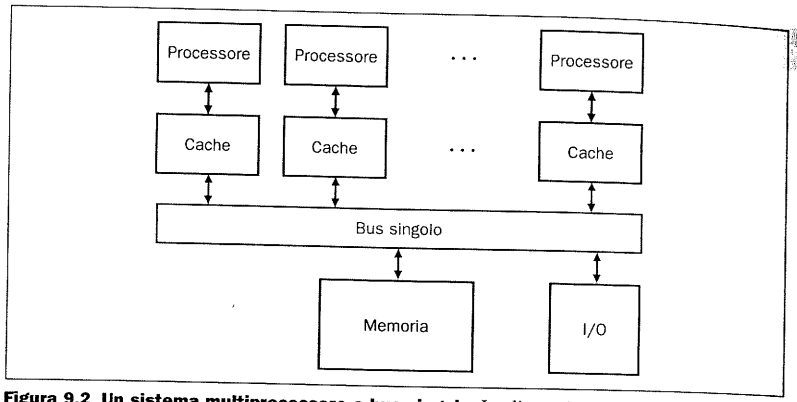


Figura 9.2 Un sistema multiprocessore a bus singolo. La dimensione tipica è compresa fra 2 e 32 processori.

Nome	Numero massimo di processori	Nome del processore	Frequenza di clock del processore	Massima dimensione della memoria/sistema	Ampiezza della banda di comunicazione/sistema
Compaq ProLiant 5000	4	Pentium Pro	200 MHz	2048 MB	540 MB/s
Digital AlphaServer 8400	12	Alpha 21164	440 MHz	28672 MB	2150 MB/s
HP 9000 K460	4	PA-8000	180 MHz	4096 MB	960 MB/s
IBM RS/6000 R40	8	PowerPC 604	112 MHz	2048 MB	1800 MB/s
SGI Power Challenge	36	MIPS R10000	195 MHz	16384 MB	1200 MB/s
Sun Enterprise 6000	30	UltraSPARC 1	167 MHz	30720 MB	2600 MB/s

Figura 9.3 Caratteristiche dei sistemi multiprocessore connessi da un solo bus di backplane, in commercio nel 1997. Lo stile di comunicazione di questi calcolatori è lo spazio di indirizzamento condiviso con tempo uniforme di accesso alla memoria. Queste macchine in generale sono progettate per essere usate con diverse generazioni di processori, sia per consentire ai clienti di aggiornare le macchine esistenti che per permettere alle aziende costruttrici di ammortizzare gli investimenti in ricerca e sviluppo. Ad esempio l'SGI Power Challenge fu inizialmente introdotto nel mercato nel 1993 con il MIPS R4400, e poi nuovamente nel 1995 con l'R8000; si noti che il bus ed il sistema di memoria non cambiarono nel frattempo. Si consiglia di consultare l'indirizzo *www.mkp.com/cod2e.htm* per i puntatori a questi ed altri sistemi multiprocessore più recenti a bus singolo.

Esempio

Programma parallelo (a bus singolo)

Si supponga di voler sommare 100000 numeri su un sistema multiprocessore a bus singolo; si assuma che i processori siano 10.

Soluzione

Il primo passo consiste nel suddividere l'insieme dei numeri in sottoinsiemi delle stesse dimensioni. I sottoinsiemi non sono allocati in memorie diverse, dal momento che in questo tipo di calcolatori esiste un'unica memoria; ciascun processore avrà semplicemente un proprio indirizzo di partenza. `Pn` rappresenta il numero del processore, compreso fra 0 ed 9. Tutti i processori iniziano il programma eseguendo un ciclo che somma il proprio sottoinsieme di numeri:


```
somma[Pn] = 0;
for (i = 10000*Pn; i < 10000*(Pn+1); i = i + 1)
   somma[Pn] = somma[Pn] + A[i]; /* somma le aree assegnate */
```

Questo ciclo usa delle istruzioni di load per trasferire l'opportuno sottoinsieme di numeri dalla memoria principale alla cache di ciascun processore.

Il passo successivo consiste nel sommare tutte le somme parziali, secondo il principio del "divide et impera"; metà dei processori sommano coppie di somme parziali, quindi un quarto dei processori sommano coppie delle nuove somme parziali, e così via fino all'unica somma finale. Ciascun processore deve avere la propria variabile `i` per il controllo del ciclo, e quindi `i` deve essere una variabile "privata".

In questo esempio due processori devono sincronizzarsi prima che il processore "consumatore" cerchi di leggere il risultato dalla locazione di memoria scritta dal processore "produttore", altrimenti il consumatore potrebbe leggere il valore vecchio del dato. Il codice è riportato nel seguito (`mezzo` è un'altra variabile privata):

```
mezzo = 10; /* 10 processori nel sistema multiprocessore a
               bus singolo */
repeat
          synch(); /* aspetta il completamento della somma
                      parziale */
          if (mezzo%2 != 0 && Pn == 0)
                 somma[0] = somma[0] + somma[mezzo-1];
         mezzo = mezzo/2; /* linea di divisione fra chi somma */
          if (Pn < mezzo) somma[Pn] = somma[Pn] +
            somma[Pn+mezzo];
until (mezzo == 1); /* esce con la somma finale in somma[0] */
```

Quella che è stata usata in questo esempio è una primitiva di *sincronizzazione a barriera*; i processori aspettano alla barriera finché tutti l'hanno raggiunta, e quindi ripartono. Il meccanismo di sincronizzazione a barriera fa sì che tutti i processori si sincronizzino rapidamente. Questa funzione può essere implementata sia via software, con la primitiva di sincronizzazione di lock (descritta più avanti), sia con un circuito hardware che combina il segnale di "pronto" di tutti i processori in un segnale globale che tutti i processori possono controllare.

Nel Capitolo 8 si è visto che l'I/O può essere soggetto a fenomeni di incoerenza fra la versione dei dati contenuti nella memoria e quella dei dati contenuti nella cache; questo problema di *coerenza della cache* può verificarsi anche nel caso dei sistemi multiprocessore. A differenza dell'I/O, che raramente utilizza copie multiple di dati (una situazione da evitare il più possibile), i processori multipli necessitano comunemente di copie degli stessi dati mantenute in cache diverse, come suggerisce la seconda metà dell'esempio. In alternativa l'accesso ai dati condivisi potrebbe essere forzato ad ignorare la cache ed accedere solo alla memoria centrale, ma sarebbe troppo lento ed userebbe troppa ampiezza di banda del bus; le prestazioni di un programma su un sistema multiprocessore dipendono dalle prestazioni nel caso di accesso ai dati condivisi.

I protocolli che mantengono la coerenza nel caso di processori multipli sono detti *protocolli di coerenza delle cache*; i prossimi paragrafi spiegano questi protocolli ed i metodi di sincronizzazione dei processori che li utilizzano.

Coerenza delle cache nei sistemi multiprocessore

Il protocollo più diffuso per mantenere la coerenza dei dati nelle cache è chiamato *snooping* (dall'inglese to snoop, spiare). La figura 9.4 mostra come le cache accedano alla memoria attraverso il bus comune; tutti i controllori delle cache "spiano" il bus per vedere se possiedono o meno una copia del dato condiviso.

Questo meccanismo si diffuse nei calcolatori negli anni '80, che avevano un unico bus per collegare le memorie principali; i processori singoli di questo tipo vennero estesi aggiungendo altri processori sullo stesso bus, per dare un semplice meccanismo di accesso alla memoria condivisa. Le cache vennero poi aggiunte per migliorare le prestazioni di ciascun processore, usando degli schemi per mantenere le cache aggiornate basati sul controllo delle informazioni che transitano sul bus condiviso.

Mantenere la coerenza presenta due aspetti, che coinvolgono le operazioni di lettura e quelle di scrittura. Copie multiple non sono un problema nel caso delle letture, ma un processore deve avere un accesso esclusivo in scrittura per poter scrivere una parola. I processori inoltre devono avere accesso alla copia più recente quando leggono un dato, per cui tutti processori devono ottenere il nuovo dato dopo una scrittura. I protocolli di snooping devono quindi localizzare tutte le cache che condividono un oggetto che deve essere scritto; la conseguenza di un'operazione di scrittura sui dati condivisi è l'invalidazione di tutte le altre copie oppure l'aggiornamento delle copie condivise con il nuovo valore che è stato scritto.

I bit di stato già presenti in un blocco di cache sono estesi nel caso dei protocolli di snooping, e queste informazioni sono usate per tenere sotto controllo le attività sul bus. Al verificarsi di una situazione di miss in lettura, tutte le cache controllano se possiedono una copia del blocco richiesto, e agiscono in modo appropriato, ad esempio fornendo il dato alla cache che ne era priva. In modo analogo nel caso delle operazioni di scrittura le cache controllano se hanno una copia del blocco, ed agiscono in modo opportuno, invalidando o aggiornando la copia del valore che è stato modificato.

Dato che in ogni transazione sul bus vengono controllati i campi tag delle cache, si potrebbe assumere un'interferenza con il processore; tale interferenza si verificherebbe se non fosse per la duplicazione della porzione di tag dell'indirizzo della

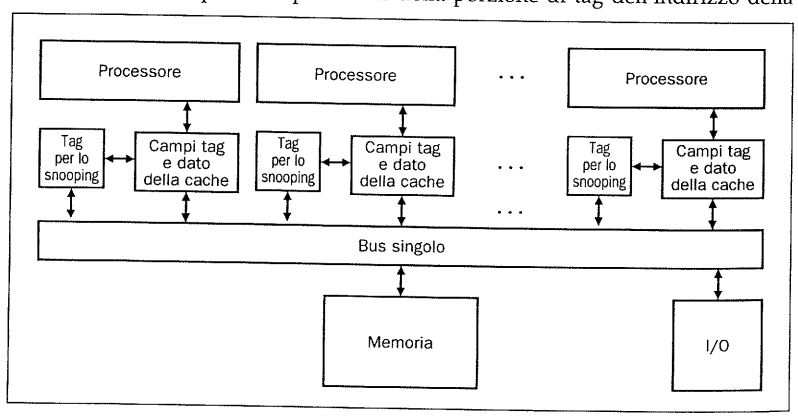


Figura 9.4 Un sistema multiprocessore a bus singolo che utilizza il protocollo di snooping per mantenere la coerenza delle cache. I campi tag aggiuntivi, mostrati in colore, servono per gestire le richieste di snoop; tali campi sono duplicati per ridurre il carico sulle cache dovuto allo snooping.

cache (non dell'intera cache), che fornisce un'ulteriore porta di lettura dedicata allo snooping (vedere figura 9.4). In questo modo lo snooping raramente interferisce con l'accesso alla cache da parte del processore; quando si verifica un'interferenza, il processore probabilmente verrà posto in stallo per la non disponibilità della cache.

I protocolli di snooping sono di due tipi, a seconda di ciò che avviene nel caso delle operazioni di scrittura:

- *Invalidazione in scrittura* (*write-invalidate*): il processore che esegue l'operazione di scrittura fa sì che tutte le copie contenute nelle altre cache siano invalidate prima di cambiare la propria copia locale; a questo punto è libero di aggiornare i dati *locali* fino a quando un altro processore ne fa richiesta. Il processore che scrive invia sul bus un segnale di invalidazione, e tutte le cache controllano se possiedono una copia della parola scritta; se lo posseggono invalidano il blocco che contiene la parola. Questo schema permette quindi operazioni di lettura multiple ed operazioni di scrittura singole.
- *Aggiornamento in scrittura* (*write-update*): invece di invalidare tutti i blocchi condivisi, il processore che esegue l'operazione di scrittura trasmette sul bus il nuovo dato, e tutte le copie sono aggiornate con il nuovo valore. Questo schema, chiamato anche broadcast in scrittura, trasmette in continuazione le operazioni di scrittura sui blocchi condivisi, mentre lo schema di invalidazione in scrittura cancella tutte le altre copie in modo che ci sia un'unica copia locale nelle operazioni di scrittura successive.

Il meccanismo di aggiornamento in scrittura è analogo al write-through, dato che tutte le operazioni di scrittura attraversano il bus per aggiornare le copie dei dati condivisi. Il meccanismo di invalidazione in scrittura invece usa il bus solo nel caso della *prima* operazione di scrittura per invalidare tutte le altre copie, e le scritture successive non generano attività sul bus. Di conseguenza l'invalidazione in scrittura ha gli stessi vantaggi del write-back in termini di riduzione della richiesta di banda del bus, mentre l'aggiornamento in scrittura ha il vantaggio di far caricare prima i nuovi valori nella cache, il che può ridurre la latenza.

I sistemi multiprocessore in commercio usano cache di tipo write-back per ridurre il traffico sul bus e permettere l'uso di un solo bus per connettere i processi; per risparmiare preziosa banda di comunicazione, tutti i calcolatori in commercio usano l'invalidazione in scrittura come protocollo standard.

Ad oggi le misurazioni effettuate indicano che i dati condivisi hanno una località spaziale e temporale inferiore agli altri tipi di dati; quindi le situazioni di miss che riguardano i dati condivisi dominano spesso il comportamento delle cache, anche se rappresentano solo dal 10% al 40% degli accessi ai dati.

Interfaccia Hardware Software

Si può intuire che le dimensioni dei blocchi giocano un ruolo importante nella coerenza delle cache. Si consideri ad esempio il caso di snooping su una cache con blocchi di otto parole, con una singola parola che viene alternativamente scritta e letta da due processori: un protocollo che trasmette una sola parola è più vantaggioso di un protocollo che trasferisce l'intero blocco.

Blocchi di grandi dimensioni possono anche causare quella che si chiama *falsa condivisione*: quando due variabili condivise non correlate fra loro si trovano nello stesso blocco della cache, l'intero blocco viene scambiato fra processori anche se questi ultimi accedono a variabili diverse (vedere gli Esercizi 9.5 e

9.6). Attività di ricerca in corso sui compilatori sono volte a ridurre la falsa condivisione allocando dati ad alta correlazione nello stesso blocco della cache, riducendo così il verificarsi di situazioni di miss della cache.

Approfondimento: Che cosa succede se in un sistema multiprocessore a bus singolo che garantisce la coerenza delle cache due processori cercano di scrivere nella stessa parola condivisa durante lo stesso ciclo di clock? L'arbitro del bus decide quale processore ha per primo il controllo del bus, e questo processore invaliderà o aggiornerà la copia dell'altro processore, a seconda del protocollo usato. A questo punto il secondo processore potrà eseguire la sua scrittura. L'arbitraggio del bus forza un comportamento sequenziale nel caso di scrittura da parte di più processori, il che spiega perché le operazioni di scrittura da parte di più processori su parole diverse appartenenti allo stesso blocco funzionino correttamente.

La politica che determina quando un processore si accorge di un'operazione di scrittura da parte di un altro processore è detta *modello di coerenza della memoria*. La più conservativa è chiamata *coerenza sequenziale*: il risultato di qualunque esecuzione è lo stesso che si verificherebbe se gli accessi di ciascun processore fossero sequenziali, e se gli accessi da parte dei diversi processori fossero alternati. Altri calcolatori usano modelli meno conservativi per ottenere migliori prestazioni dalla memoria.

Un esempio di protocollo per la coerenza delle cache

Allo scopo di illustrare le complicazioni di un protocollo per la coerenza delle cache, la figura 9.5 mostra il diagramma di transizione a stati finiti di un protocollo di invalidazione in scrittura basato sulla politica del write-back. Ciascun blocco della cache si trova in uno dei tre stati:

- *Solo lettura*: il blocco di cache è pulito (non scritto) e può essere condiviso.
- *Lettura/scrittura*: il blocco di cache è sporco (scritto) e *non* può essere condiviso.
- *Invalido*: il blocco di cache non contiene dati validi.

I tre stati del protocollo sono duplicati in figura per mostrare le transizioni dovute alle azioni del processore rispetto a quelle dovute alle operazioni sul bus. Tale duplicazione è fatta solo allo scopo di illustrare il protocollo: c'è un'unica macchina a stati finiti per ogni cache, con i segnali provenienti dal relativo processore o dal bus.

Le transizioni di stato nei blocchi di una cache si verificano in condizioni di miss in lettura e scrittura, e di hit in scrittura; gli hit in lettura non cambiano lo stato della cache. Per prima cosa si prenda in considerazione una situazione di miss in lettura; quando il processore si trova in questa condizione per quanto riguarda un blocco di cache, cambia lo stato di tale blocco in "Sola lettura", dopo di che acquisisce il controllo del bus e riscrive il vecchio blocco se questo si trovava nello stato di "Lettura/scrittura" (sporco). Tutte le cache degli altri processori osservano la situazione di miss in lettura per vedere se contengono il blocco in questione; se una di esse ne ha una copia che si trova nello stato di "Lettura/scrittura", cambia lo stato del blocco in "Invalido" (alcuni protocolli lo cambierebbero invece in "Sola lettura"). La situazione di miss in lettura è quindi risolta eseguendo una lettura dalla memoria.

Si veda ora il caso di miss in scrittura; per scrivere un blocco, il processore acquisisce il controllo del bus, invia un segnale di invalidazione, scrive nel blocco e lo pone nello stato di "Lettura/scrittura". Osservando il bus, tutte le altre cache controllano se hanno una copia del blocco, e se sì la invalidano.

Come si può immaginare, esistono molte varianti di questo protocollo per la coerenza delle cache, molto più complesse di questo semplice modello. Quella

usata sia dal Pentium Pro che dal PowerPC si chiama *MESI*, un protocollo di invalidazione in scrittura il cui nome è l'acronimo dei quattro possibili stati: Modificato, Esclusivo, Condiviso, Invalido (Modified, Exclusive, Shared, Invalid). Lo stato "Modificato" è analogo a quello di "Lettura/scrittura" riportato in figura 9.5, ed anche lo stato "Invalido" è lo stesso; lo stato di "Sola lettura" di figura 9.5 è suddiviso in due stati, a seconda che esistano copie multiple (stato "Condiviso") o una copia unica (stato "Esclusivo") del blocco; in entrambi i casi la memoria ha una versione aggiornata dei dati. Lo stato aggiuntivo sta ad indicare che un'operazione di scrittura su dati che si trovano nello stato "Esclusivo" non necessita della procedura di invalidazione, dato che esiste un'unica copia del blocco; al contrario un'operazione di scrittura su dati che si trovano nello stato di "Sola lettura" di figura 9.5 richiederebbe la procedura di invalidazione, dato che potrebbero esistere altre copie.

Altre varianti del protocollo per la coerenza delle cache comprendono la possibilità che una cache fornisca il dato richiesto ad un'altra cache, se ne ha una copia, la possibilità di invalidare o meno un blocco in caso di miss in lettura, e la possibilità di invalidare oppure aggiornare i dati condivisi.

Meccanismi di sincronizzazione che usano la coerenza dei dati

Uno dei requisiti principali di un sistema multiprocessore a bus singolo è la capacità di coordinare processi che lavorano ad un obiettivo comune. Solitamente il programmatore utilizza delle *variabili di lock*, comunemente dette *semafori*, per coordinare o sincronizzare i processi; la sfida per chi progetta i sistemi multiprocessore è fornire un meccanismo per decidere quale processore abbia il controllo e predisporre l'operazione che crea un lock su una variabile. Nel caso dei sistemi multiprocessore a bus singolo l'arbitraggio è facile, dato che il bus è l'unica via per raggiungere la memoria: il processore che ha il controllo del bus non permette a nessun altro processore di accedere alla memoria. Se il processore e il bus forniscono un'*operazione atomica di scambio* (*swap*), i programmatori possono creare delle situazioni di lock con una semantica opportuna; l'aggettivo *atomico* in questo caso vuol dire indivisibile, per cui uno scambio atomico significa che un processore può leggere una locazione *e* fare il lock del suo valore durante la stessa operazione del bus, facendo in modo che nessun altro processore o dispositivo di I/O possa accedere alla memoria in lettura o scrittura finché l'operazione di scambio non è stata completata.

La figura 9.6 mostra una procedura tipica per fare il lock di una variabile con un'istruzione atomica di scambio. Si assuma che 0 significhi che la variabile non ha il lock ("via libera") e che 1 significhi che ha il lock ("stop"); per prima cosa il processore legge la variabile per controllarne lo stato, e continua a leggerla e controllarla fino a quando il valore indica che il lock non è attivo.

Quindi il processore compete con tutti gli altri processori in *attesa* per stabilire chi per primo può fare il lock della variabile. Tutti i processori usano un'istruzione atomica di scambio che legge il vecchio valore e memorizza un 1 ("stop") per fare il lock della variabile; l'unico vincitore vedrà lo 0 ("via libera"), mentre i perdenti vedranno l'1 che è stato memorizzato dal vincitore. Il processore che ha vinto esegue il codice ed aggiorna il dato condiviso, e prima di terminare inserisce uno 0 ("via libera") per togliere il lock alla variabile; a questo punto la gara ricomincia.

Si esamini ora come lo schema di lock di figura 9.6 mantenga la coerenza delle cache nel caso di bus singolo. Uno dei vantaggi di questo algoritmo è che consente

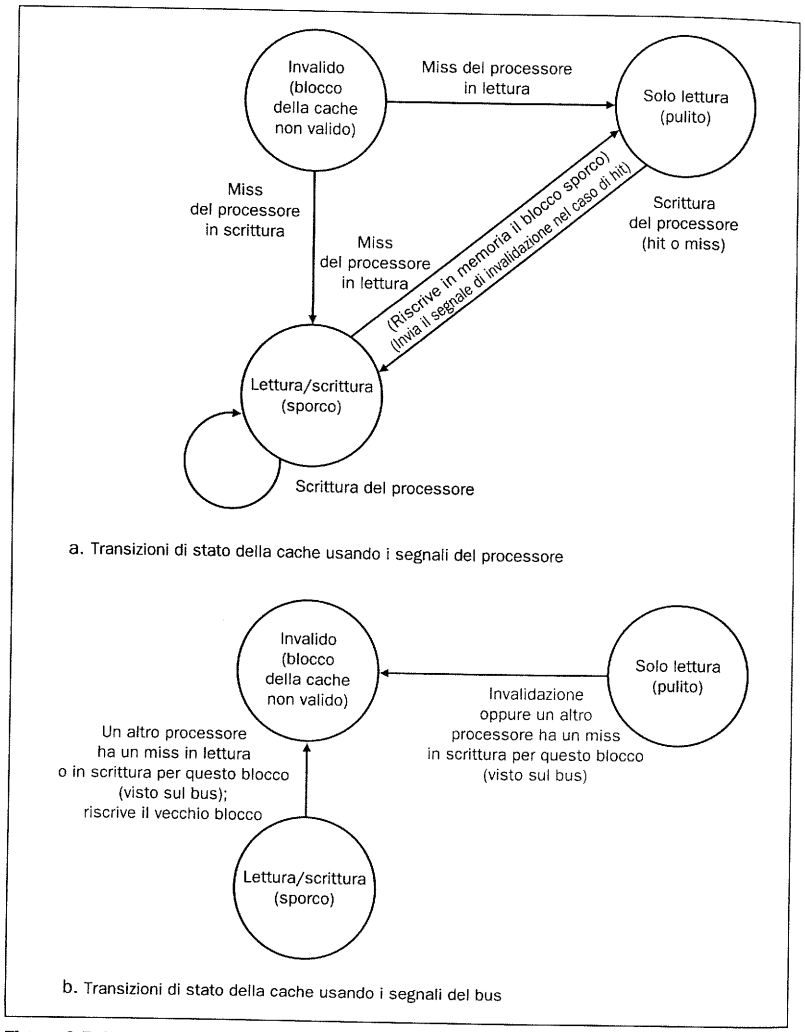


Figura 9.5 Un protocollo per la coerenza delle cache a invalidazione in scrittura. La parte superiore del diagramma mostra le transizioni di stato dovute ad azioni del processore associato, mentre la parte inferiore mostra le transizioni dovute alle azioni di altri processori viste come operazioni sul bus. In realtà c'è una sola macchina a stati finiti per ogni cache, benché qui ne siano rappresentate due per chiarire il verificarsi delle transizioni. Le freccie nere e le azioni scritte in nero sarebbero le stesse nel caso di cache che non garantiscono la coerenza; le frecce e le azioni colorate sono quelle aggiunte per ottenere la coerenza delle cache. A differenza di quanto mostrato in questa figura, alcuni protocolli considerano le operazioni di scrittura su dati puliti come situazioni di miss della cache, per cui non esiste un segnale di invalidazione separato.

ai processori di aspettare il proprio turno basandosi su una copia locale della variabile di lock contenuta nella propria cache; ciò riduce la quantità di traffico sul bus.

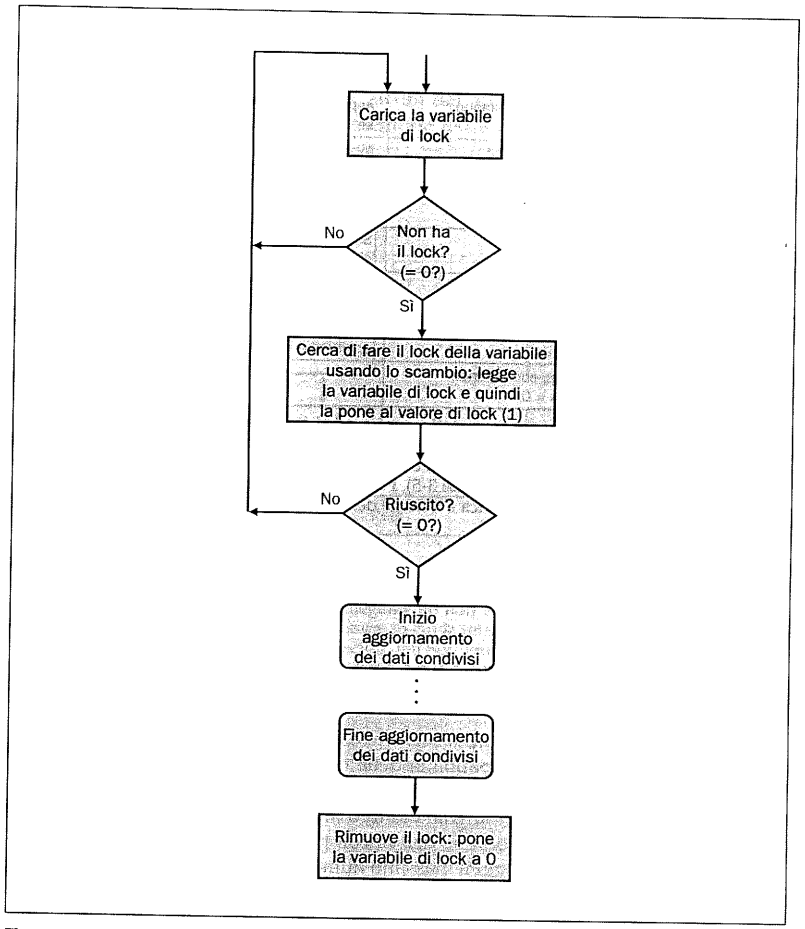


Figura 9.6 I passi necessari ad acquisire un lock, o semaforo, per sincronizzare i processi, e per rilasciarlo in uscita dalla porzione chiave del codice.

La figura 9.7 mostra le operazioni del bus e delle cache nel caso in cui più processori vogliano fare il lock di una variabile. Una volta che il processore che ha il controllo memorizza uno 0 nella variabile di lock, tutti gli altri se ne accorgono e invalidano la copia della variabile di lock all'interno della propria cache, e in seguito cercano di ottenerne il nuovo valore (nel caso della coerenza della cache ottenuta con l'aggiornamento in scrittura, le cache aggiornano la propria copia invece di invalidarla prima e caricare dalla memoria il nuovo valore poi). Il nuovo valore fa sì che inizi la competizione per stabilire chi può fare il lock; il vincitore ottiene il controllo del bus e memorizza 1 nella variabile di lock, mentre le altre cache sostituiscono la propria copia della variabile di lock che contiene 0 con il valore 1; a questo punto si rendono conto che la variabile non è disponibile, e continuano ad aspettare.

Passo	Processore P0	Processore P1	Processore P2	Attività sul bus
1	Ha il lock	Aspetta, controllando se lock = 0	Aspetta, controllando se lock = 0	Nessuna
2	Mette il lock a 0 e invia 0 sul bus	Aspetta, controllando se lock = 0	Aspetta, controllando se lock = 0	Invalidazione in scrittura della variabile di lock da parte di P0
3		Miss della cache	Miss della cache	Il bus decide di servire il miss della cache di P2
4		(Aspetta mentre il bus è occupato)	Lock = 0	Il miss della cache di P2 è soddisfatto
5		Lock = 0	Scambio: legge il lock e lo mette a 1	Il miss della cache di P1 è soddisfatto
6		Scambio: legge il lock e lo mette a 1	Valore di scambio = 0 e invia 1 sul bus	Invalidazione in scrittura della variabile di lock da parte di P2
7		Valore di scambio = 1 e invia 1 sul bus	Ottiene il lock, per cui può aggiornare i dati condivisi	Invalidazione in scrittura della variabile di lock da parte di P1
8		Aspetta, controllando se lock = 0		Nessuna

Figura 9.7 I passi riguardanti la coerenza delle cache ed il traffico sul bus per tre processori P0, P1 e P2. Questa figura suppone un meccanismo per la garanzia della coerenza di tipo invalidazione in scrittura. P0 parte con il controllo del lock (passo 1). P0 termina e rilascia il lock (passo 2). P1 e P2 competono per stabilire chi legge il valore rilasciato durante lo scambio (passi 3-5). P2 vince ed entra nella *sezione critica* (passi 6 e 7), mentre P1 aspetta il suo turno (passi 7 e 8). Sezione critica è il nome della porzione di codice fra l'attivazione ed il rilascio del lock.

Questo schema è difficile da estendere al caso di molti processori per via del traffico generato quando viene rilasciato il lock.

9.4 Sistemi multiprocessore connessi via rete

I sistemi multiprocessore a bus singolo oltre a notevoli vantaggi presentano alcuni limiti dovuti all'incompatibilità delle tre caratteristiche principali dei bus: ampiezza di banda elevata, bassa latenza e lunghezza. Esiste anche un limite dovuto all'ampiezza di banda di un singolo modulo di memoria connesso al bus. Quindi l'organizzazione a bus singolo pone dei vincoli pratici al numero di processori che possono essere connessi. Ad oggi il massimo numero di processori connessi ad un singolo bus nei calcolatori in commercio è 36, e questo numero sembra diminuire con il tempo.

Se l'obiettivo è connettere un numero molto maggiore di processori, allora chi progetta il calcolatore deve necessariamente usare più di un bus; la figura 9.8 mostra quale può essere l'organizzazione. Si noti che nella figura 9.2 il mezzo attraverso cui avviene il collegamento (il bus) si trova fra il processore e la memoria, mentre in figura 9.8 la memoria è connessa a ciascun processore, ed il mezzo di collegamento (la rete) è fra i diversi nodi. Nei sistemi a bus singolo il bus è usato per tutti gli accessi in memoria, mentre nel secondo caso il sistema di interconnessione è coinvolto solo nella comunicazione fra processori diversi. La figura 9.9 elenca diversi sistemi multiprocessore connessi via rete.

Tutto ciò riporta ad un vecchio dibattito sull'organizzazione della memoria nei processori ad elaborazione parallela su larga scala; sfortunatamente tale dibattito è spesso centrato su una falsa dicotomia: *memoria condivisa* verso *memoria distribuita*. Memoria condivisa significa in realtà singolo spazio di indirizzamento, indicando in modo implicito che la comunicazione avviene tramite operazioni di load e di store. All'opposto del singolo spazio di indirizzamento si trovano in realtà le *memorie private multiple*, in cui la comunicazione avviene attraverso operazioni di trasmissione e ricezione.

La memoria distribuita si riferisce alla localizzazione fisica della memoria. Se la memoria fisica è suddivisa in moduli, alcuni dei quali vicini a ciascun processore, come mostrato in figura 9.8, allora la memoria fisica è distribuita. All'opposto della memoria distribuita c'è in realtà la *memoria centralizzata*, in cui il tempo di accesso

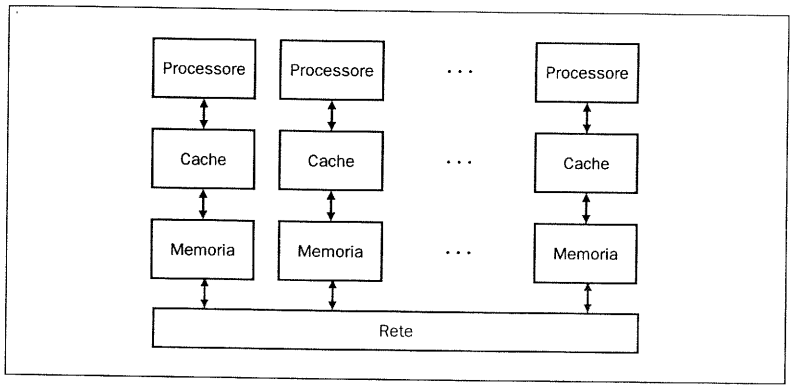


Figura 9.8 L'organizzazione di un sistema multiprocessore connesso via rete. Si noti che, a differenza della figura 9.2, il mezzo di collegamento non è più fra i processori e la memoria. Sono anche stati costruiti sistemi multiprocessore in cui la rete è al di sopra della memoria; i Cray XMP e YMP sono forse gli esempi più conosciuti, e il Sun Enterprise 10000 l'esempio più recente (vedere figura 9.9).

Nome	Numero massimo di processori	Nome del processore	Frequenza di clock del processore	Massima dimensione della memoria/sistema	Ampiezza della banda di comunicazione/collegamento	Nodo	Topologia
Cray Research T3E	2048	Alpha 21164	450 MHz	524288 MB	1200 MB/s	SMP a 4 vie	Toroidale 3D
HP/Convex Exemplar X-class	64	PA-8000	180 MHz	65536 MB	980 MB/s	SMP a 2 vie	Crossbar a 8 vie+anello
Sequent NUMA-Q	32	Pentium Pro	200 MHz	131072 MB	1024 MB/s	SMP a 4 vie	Anello
SGI Origin2000	128	MIPS R10000	195 MHz	131072 MB	800 MB/s	SMP a 2 vie	6-cubo
Sun Enterprise 10000	64	UltraSPARC 1	250 MHz	65536 MB	1600 MB/s	SMP a 4 vie	Crossbar a 16 vie

Figura 9.9 Caratteristiche dei sistemi multiprocessore connessi via rete in commercio nel 1997. Tutti questi calcolatori hanno un singolo spazio di indirizzamento con tempo di accesso alla memoria non uniforme, a parte il Sun Enterprise 10000, che ha uno spazio di indirizzamento condiviso con tempo di accesso alla memoria uniforme. Tutti questi calcolatori, a parte il Cray Research T3E, garantiscono la coerenza delle cache; HP, Sequent e SGI lo fanno con l'uso delle directory. I calcolatori Sun utilizzano i bus per gli indirizzi, e gli switch per i dati, per cui la coerenza è realizzata tramite uno snooping convenzionale sul bus degli indirizzi. L'ampiezza della banda di comunicazione riportata si riferisce al picco per ciascun collegamento, tenendo conto di tutti i byte inviati, compresi quelli di intestazione della rete. La banda di bisezione solitamente dipende dal numero di processori. Si consiglia di consultare l'indirizzo *www.mkp.com/cod2e.htm* per i puntatori a questi ed altri sistemi multiprocessore più recenti connessi via rete.

ad una locazione della memoria fisica è lo stesso per tutti i processori dato che qualunque accesso passa sull'interconnessione, come in figura 9.2. Questo stile di organizzazione è spesso chiamato *sala da ballo*: i processori sono tutti da una parte e le memorie dall'altra, evocando l'immagine di una scuola di ballo con i ragazzi da una parte della sala e le ragazze dall'altra (il Tera Computer ne è un esempio, vedere l'indirizzo *www.mkp.com/cod2e.htm*).

Come detto nel Paragrafo 9.1, un singolo spazio di indirizzamento rispetto ad uno spazio di indirizzamento multiplo, così come la memoria distribuita rispetto alla memoria centralizzata sono aspetti ortogonali: i sistemi multiprocessore possono avere spazio di indirizzamento singolo e memoria fisica distribuita. Per essere appropriato il dibattito deve riguardare i pro e i contro dello spazio di indirizzamento singolo, della comunicazione esplicita e della memoria fisica distribuita.

Nei calcolatori privi di uno spazio di indirizzamento globale la comunicazione è necessariamente esplicita: il programmatore o il compilatore devono inviare messaggi per trasferire dei dati ad un altro nodo, e ricevere messaggi per accettare i dati.

Esempio

Programma parallelo (scambio di messaggi)

Si esamini di nuovo l'esempio della somma nel caso di un sistema multiprocessore connesso via rete, con 100 processori che usano memorie private.

Soluzione

Dato che questo calcolatore ha uno spazio di indirizzamento multiplo, il primo passo consiste nel distribuire i 100 sottoinsiemi in ciascuna delle memorie locali; il processore che contiene i 100000 numeri invia i sottoinsiemi a ciascuno dei 100 nodi processore-memoria.

Il passo successivo consiste nell'ottenere la somma di ciascun sottoinsieme: tutte le unità di elaborazione eseguono un semplice ciclo, che legge una parola dalla memoria locale e la somma al contenuto di una variabile locale:

```
somma = 0;
for (i = 0; i < 1000; i = i + 1) /* esegue un ciclo su
                                    ciascun vettore */
 somma = somma + A1[i]; /* somma i vettori locali */
 limite = 100;
```

L'ultimo passo consiste nel sommare le 100 somme parziali. Il difficile sta nel fatto che ciascuna di queste somme parziali è localizzata in una diversa unità di esecuzione, e quindi è necessario usare la rete di interconnessione per far sì che le somme parziali si accumulino nella somma finale. Invece di inviare tutte le somme parziali ad un solo processore, che dovrebbe sommarle sequenzialmente, conviene nuovamente suddividere il compito. Per prima cosa metà delle unità di esecuzione inviano la propria somma parziale all'altra metà, dove vengono sommate due somme parziali. Successivamente un quarto delle unità di esecuzione (la metà della metà) invia la nuova somma parziale all'altro quarto (la restante metà della metà) per un nuovo turno di somme. Questo meccanismo di suddivisione in metà, di invio e di ricezione continua finché si ottiene un'unica somma di tutti i numeri. Se `Pn` rappresenta il numero delle unità di esecuzione, `invia(x,y)` è una procedura che invia attraverso la rete di interconnessione il valore `y` all'unità di esecuzione `x`, e `ricevi()` è una funzione che accetta un valore proveniente dalla rete, il seguente codice rappresenta il meccanismo sopra esposto:

```
mezzo = 100; /* 100 processori */
repeat
 mezzo = (mezzo+1)/2; /* linea di divisione fra chi invia e
                         chi riceve */
 if (Pn >= mezzo && Pn < limite) invia(Pn - mezzo, somma);
 if (Pn < (limite/2 - 1)) somma = somma + ricevi();
 limite = mezzo; /* limite superiore di chi invia */
until (mezzo == 1); /* esce con la somma finale */
```

Questo codice suddivide i processori in trasmittenti e riceventi, e ciascun processore ricevente riceve un solo messaggio, per cui si può presumere che sia in stallo fino a quando riceve un messaggio. Quindi trasmissioni e ricezioni possono essere usate come primitive di sincronizzazione oltre che di comunicazione, dal momento che i processori sanno quando avviene la trasmissione dei dati.

L'indirizzamento nei processori paralleli a larga scala

Poiché è molto difficile e molto costoso costruire un calcolatore che possa aumentare progressivamente il numero di processori ed il numero di moduli di memoria, la maggior parte dei processori paralleli a larga scala usa la memoria distribuita.

Negli elaboratori con memoria distribuita si deve molto spesso affrontare il problema della comunicazione. La soluzione più semplice per il progettista dell'hardware è quella di consentire solo i meccanismi di invio e ricezione, evitando la comunicazione implicita possibile con le operazioni di load e di store. L'invio e la ricezione di messaggi ha anche il vantaggio di rendere più facile al programmatore l'ottimizzazione della comunicazione: sovrapporre l'elaborazione alla comunicazione è più semplice nel caso di invio e ricezione di messaggi rispetto al caso delle operazioni di load e di store.

Dall'altro lato le operazioni di load e store solitamente implicano un carico di lavoro aggiuntivo meno pesante dell'invio e della ricezione di messaggi. Alcune applicazioni inoltre fanno riferimento ad informazioni remote a cui viene fatto accesso solo in modo occasionale e imprevedibile, per cui è molto più efficiente usare l'indirizzo del dato remoto quando *richiesto*, invece di richiamarlo nel caso *dovesse essere* utile. Macchine di questo tipo hanno la *memoria condivisa distribuita* (*DSM, distributed shared memory*).

Le cache sono importanti per avere buone prestazioni, indipendentemente da come viene realizzata la comunicazione, per cui è necessario che i dati condivisi compaiano sia nella cache del processore che possiede i dati sia in quella del processore che li richiede. Ne consegue che fornire un singolo indirizzamento globale nel caso di sistemi multiprocessore connessi via rete fa sorgere nuovamente il problema della coerenza delle cache, dal momento che esistono più copie degli stessi dati con lo stesso indirizzo in processori diversi. Chiaramente il protocollo di snooping sul bus descritto nel Paragrafo 9.3 non funziona in questo caso, dato che non esiste un unico bus tramite il quale i riferimenti alla memoria sono inviati a tutti. Dato che i progettisti del Cray T3E (vedere figura 9.9) non avevano previsto nessun bus per supportare la coerenza delle cache, il T3E ha un singolo spazio di indirizzamento ma non garantisce la coerenza delle cache.

Interfaccia Hardware Software

Aggiungere uno strato software per consentire un singolo spazio di indirizzamento nascondendo invio e ricezione di messaggi, in modo che la comunicazione sia parte delle operazioni di load e di store è difficile, anche se è paragonabile al sistema di memoria virtuale che si trova già nella maggior parte dei processori (vedere Capitolo 7). Nel caso della memoria virtuale, un singolo processore usa delle tabelle di pagine per stabilire se un certo indirizzo si riferisce ad un dato in memoria o su disco; questo sistema di traduzione potrebbe essere modificato per decidere se un certo indirizzo si riferisce a dati locali, a dati contenuti nella memoria di un altro processore o a dati su disco. La *memoria virtuale condivisa*, come viene chiamata, crea l'illusione di avere una memoria condivisa (allo stesso modo in cui la memoria virtuale crea l'illusione di una memoria molto grande). Tuttavia, poiché invoca il sistema operativo, le prestazioni sono solitamente così ridotte che la comunicazione attraverso la memoria condivisa deve essere poco frequente. In caso contrario la maggior parte del tempo è spesa per trasferire pagine.

Un'alternativa allo snooping del bus per garantire la coerenza delle cache consiste nelle directory (elenchi). Nei protocolli basati sulle directory dal punto di vista logico esiste una sola directory che mantiene lo stato di ciascun blocco della memoria principale; le informazioni nella directory possono tenere traccia di quali cache hanno copie del blocco, se è sporco, e così via. Fortunatamente gli elementi delle directory possono essere distribuiti, in modo che richieste diverse vadano a memorie diverse, riducendo così i conflitti e consentendo la scalabilità del progetto. Le directory mantengono la caratteristica che lo stato di condivisione di un blocco è sempre in una sola locazione nota, rendendo plausibili i processori paralleli su larga scala.

Con lo snooping i progettisti delle cache e delle directory devono affrontare problemi simili; l'unica differenza sta nel meccanismo per scoprire quando si verifica un'operazione di scrittura su un dato condiviso: invece di osservare il bus per controllare se ci sono richieste per cui è necessario aggiornare o invalidare la cache locale, il controllore della directory invia comandi espliciti a tutti i processori che mantengono una copia dei dati. Questi messaggi possono poi essere inviati sulla rete; la figura 9.9 mostra le caratteristiche di diversi sistemi multiprocessore basati su directory, con accesso alla memoria non uniforme.

Interfaccia Hardware Software

Si noti che nel caso di spazio di indirizzamento unico, i dati potrebbero essere messi arbitrariamente nelle memorie di processori diversi. Questo avrebbe però due effetti negativi sulle prestazioni: per prima cosa la penalità dovuta alle situazioni di miss durerebbe di più, dato che la richiesta deve attraversare la rete. Secondariamente verrebbe consumata della banda di rete per spostare i dati al processore opportuno. Per i programmi in cui le situazioni di miss si verificano raramente questo può non essere significativo, mentre al contrario quelli ad elevata frequenza di miss avrebbero prestazioni peggiori se i dati fossero assegnati in modo casuale.

Se il programmatore o il compilatore allocano i dati nel processore che probabilmente li userà, allora questo calo di prestazioni non si verifica. A differenza dell'organizzazione con memorie private, questo tipo di allocazione deve solo essere buona, dato che i dati mancanti possono comunque essere prelevati; questo rilassamento dei vicoli semplifica il problema dell'allocazione dei dati.

Un'altra possibile soluzione consiste nell'aggiungere un secondo livello di coerenza alla memoria principale per ciascun processore; questa directory permetterebbe ai blocchi della memoria principale di migrare, liberando il programma o il compilatore dal compito di allocare i dati. Fino a che i blocchi della memoria principale non sono spostati qua e là ripetutamente, questo schema può raggiungere le prestazioni di un'allocazione intelligente della memoria al costo di una complessità hardware considerevolmente maggiore. Tale schema è chiamato *memoria solo cache* (*cache-only memory*); la migrazione può verificarsi a livello di pagina con l'ausilio del sistema operativo, oppure si può immaginare di realizzarla via hardware.

La figura 9.10 riassume le scelte per la coerenza nel caso di singolo spazio di indirizzamento.

Dato che il numero di piedini per chip è limitato, non tutti i processori possono essere connessi direttamente alla rete; questa restrizione ha ispirato una grande quantità e varietà di topologie di progetto delle reti. Nel Paragrafo 9.6 saranno esaminate le caratteristiche di alcune alternative chiave nel progetto delle reti, ma prima verrà descritto un modo alternativo di connettere i calcolatori via rete.

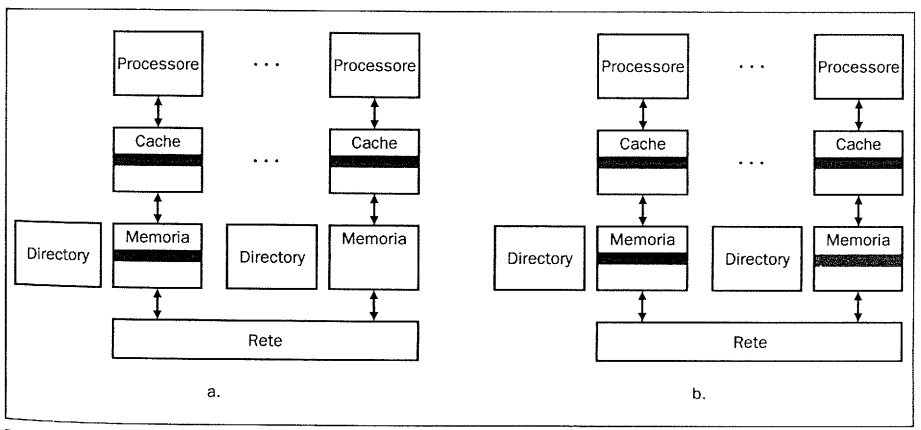


Figura 9.10 Scelte possibili per i processori paralleli a larga scala nel caso di singolo spazio di indirizzamento. I rettangoli ombreggiati rappresentano i dati replicati. *(a)* Coerenza a livello di *cache* usando le directory in un sistema multiprocessore connesso via rete; il dato originale è in memoria, e le copie sono replicate solo nelle cache. *(b)* Coerenza a livello di *memoria* usando le directory in un sistema multiprocessore connesso via rete; le copie sono replicate nella memoria remota (rappresentata in colore contrastante) *e* nelle cache. Lo schema rappresentato in *b* è simile a quello usato nel Kendall Square Research KSR-1. Fin quando la memoria è coerente, i dati possono essere inseriti nelle cache senza problemi; se il dato in memoria è invalidato, allora i blocchi corrispondenti contenuti nelle cache devono essere ugualmente invalidati.

Quadro d'insieme

La figura 9.11 confronta costi e prestazioni di sistemi multiprocessore UMA connessi tramite bus e sistemi multiprocessore NUMA connessi via rete. La rete ha un costo iniziale più basso, e poi sale un po' più rapidamente di quanto facciano i calcolatori connessi tramite bus. In entrambi i casi le prestazioni crescono linearmente finché non viene raggiunto il limite del bus, e poi la curva resta piatta indipendentemente dal numero dei processori. Quando i due effetti sono combinati, si vede che i sistemi NUMA connessi via rete hanno prestazioni costanti per unità di costo, mentre i sistemi UMA connessi via rete hanno una zona di massimo. Queste curve suggeriscono che i clienti devono essere molto più selettivi nel caso di sistemi multiprocessore connessi via bus, e che i progettisti di questi sistemi devono stare molto attenti a far sì che la zona col miglior rapporto costo-prestazioni coincida con le necessità della maggioranza dei clienti.

9.5 I cluster

Garry [Kasparov] vincerà, perché il computer non può migliorare così tanto in un solo anno.

Mike Valvo, esperto commentatore di scacchi, prima della partita fra Kasparov e Deep Blue, *Techwire*, 6 maggio 1997

Molte applicazioni per elaboratori centrali, come basi di dati, file server, Web server, simulatori e processi batch o in multiprogrammazione, potrebbero essere eseguite su sistemi connessi meno strettamente delle macchine NUMA che garantiscono la coerenza delle cache descritte nel paragrafo precedente. Spesso queste applicazioni devono essere sempre disponibili, rendendo necessaria qualche forma di tolleranza ai guasti e di riparabilità. Applicazioni di questo tipo, tenendo anche conto della somiglianza dei nodi dei sistemi multiprocessore con i calcolatori da tavolo e dello sviluppo di reti locali ad alta banda basate su switch, suggeriscono che l'elaborazione su larga scala del futuro userà *cluster* di calcolatori veri e propri. L'esempio più storico di questo è la vittoria sul campione di scacchi Kasparov nel 1997 dell'IBM SP2, un cluster di 32 nodi molto simili alle stazioni RS/6000 dotato di acceleratori hardware per l'analisi della scacchiera.

Nel 1997 alla UC Berkeley, ad esempio, un cluster di 100 calcolatori da tavolo UltraSPARC, connessi da switch Myrinet da 160 MB/s per collegamento, consentì di ottenere il record mondiale di ordinamento di basi di dati, ordinando 8,6 GB di dati che originariamente si trovavano su disco in 1 minuto, e nel decodificare un messaggio cifrato, impiegando solo 3,5 ore per decifrare una chiave DES a 40 bit.

Il cluster di Berkeley fu costruito dagli utenti finali usando componenti disponibili sul mercato, come spesso avviene in questi casi. La figura 9.12 mostra cluster venduti dalle aziende, come nel caso dell'IBM SP2; questi cluster solitamente sono venduti per fornire sistemi scalabili ed ampiamente disponibili, cioè sistemi il cui obiettivo è *sia* contenere un numero di processori, di memorie e di dischi largamente variabile, *sia* essere disponibili 24 ore al giorno per 365 giorni all'anno.

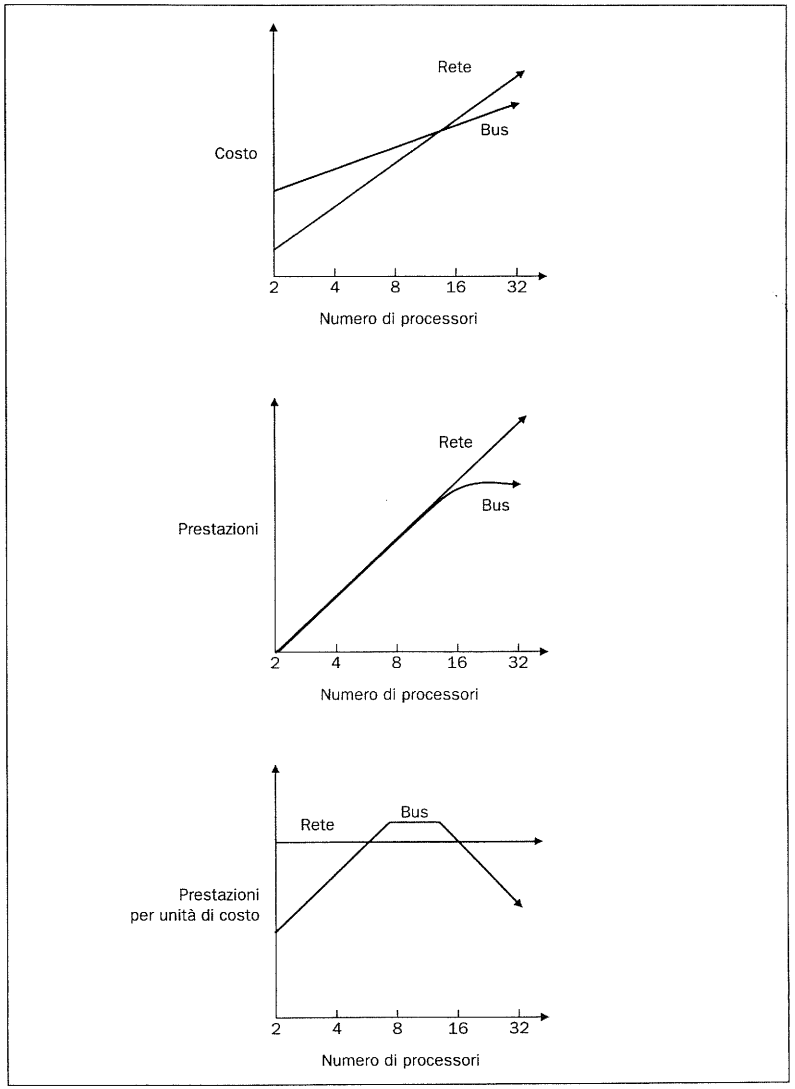


Figura 9.11 Costo, prestazioni, e rapporto costo-prestazioni di sistemi multiprocessore, connessi via bus o via rete, con spazio di indirizzamento condiviso. La combinazione di costi e prestazioni suggerisce l'esistenza di una zona di massimo per i sistemi multiprocessore connessi via bus che hanno da 8 a 16 processori (nel 1997), mostrata come una zona piatta nel grafico costo-prestazioni. I sistemi multiprocessore connessi via rete hanno un miglior rapporto costo-prestazioni a sinistra di quest'area perché sono meno costosi, e alla destra di quest'area perché hanno prestazioni più elevate. Il progettista del bus in effetti stabilisce dove si troverà la zona di massimo decidendo l'ampiezza e la velocità del bus, che determinano sia l'estremità sinistra della zona di massimo (costo) che la sua estremità destra (scalabilità).

Nome	Numero massimo di processori	Nome del processore	Frequenza di clock del processore	Massima dimensione della memoria/ sistema	Ampiezza della banda di comuni-cazione/ collega-mento	Nodo	Numero massimo di nodi
HP 9000 EPS21	64	PA-8000	180 MHz	65536 MB	532 MB/s	SMP a 4 vie	16
IBM RS/6000 HACMP R40	16	PowerPC 604	112 MHz	4096 MB	12 MB/s	SMP a 8 vie	2
IBM RS/6000 SP2	512	Power2 SC	135 MHz	1048576 MB	150 MB/s	nodo a 16 vie	32
Sun Enterprise Cluster 6000 HA	60	UltraSPARC	167 MHz	61440 MB	100 MB/s	SMP a 30 vie	2
Tandem NonSrop Himalaya S70000	4.096	MIPS R10000	195 MHz	1048576 MB	40 MB/s	SMP a 16 vie	256

Figura 9.12 Caratteristiche di cluster disponibili in commercio nel 1997. Tutti i cluster a parte l'IBM SP2 sono commercializzati per applicazioni a larga disponibilità. L'SP2 è utilizzato per applicazioni scientifiche che usano grandi quantità di numeri, e per la ricerca e la correlazione di dati (data mining). Si consiglia di consultare l'indirizzo *www.mkp.com/cod2e.htm* per i puntatori a questi ed altri cluster più recenti.

Un inconveniente dei cluster è che il costo per l'amministrazione di un cluster di N macchine è più o meno lo stesso che per N macchine indipendenti, mentre il costo per l'amministrazione di un sistema multiprocessore con spazio di indirizzamento condiviso contenente N processori è più o meno quello di una sola macchina.

Un altro inconveniente è che i cluster solitamente sono collegati usando il bus di I/O del calcolatore, mentre i sistemi multiprocessore sono connessi tramite il bus della memoria; il bus della memoria ha ampiezza di banda più elevata, permettendo ai sistemi multiprocessore di usare le connessioni di rete a velocità maggiore, e di avere meno conflitti con il traffico di I/O nel caso di applicazioni che ne fanno uso intensivo.

Un'ultima debolezza è la suddivisione della memoria: un cluster di N macchine ha N memorie indipendenti ed N copie del sistema operativo, mentre un sistema multiprocessore con la memoria condivisa consente ad un singolo programma di usare quasi tutta la memoria del sistema. Un programma sequenziale in un cluster ha quindi a disposizione una quantità di memoria pari a $1/N$ rispetto a quella a disposizione di un programma sequenziale su un sistema multiprocessore.

La debolezza delle memorie separate risulta essere una forza per la disponibilità e l'espandibilità del sistema: dato che un cluster è composto da calcolatori indipendenti connessi da una rete locale, è molto più facile sostituire una macchina senza interrompere il funzionamento del sistema di quanto non avvenga per i sistemi multiprocessore. Fondamentalmente l'esistenza della memoria condivisa significa che è difficile isolare un processore e sostituirlo senza un eroico lavoro da parte del sistema operativo. Dato che il software del cluster costituisce uno strato al di sopra del sistema operativo di ciascun calcolatore, è molto più semplice disconnettere e sostituire una macchina non funzionante.

Dal momento che i cluster sono costruiti a partire da calcolatori indipendenti e da reti indipendenti e scalabili, questo isolamento rende anche più facile espandere il sistema senza interrompere l'applicazione che lavora sul cluster. Alta disponibilità ed espandibilità rapida e incrementale rendono i cluster attraenti ai fornitori di servizi di tipo World Wide Web.

Un'altra differenza fra i due tipi di sistemi è il prezzo per ottenere una potenza di calcolo equivalente su macchine a larga scala. Dato che i sistemi multiprocessore a larga scala hanno un volume ridotto di vendite, i costi aggiuntivi di sviluppo di macchine di grandi dimensioni devono essere ammortizzati su pochi sistemi, risultando in costi più elevati per il cliente. Poiché gli stessi dispositivi, venduti in grandi quantità per piccoli sistemi, possono essere composti per costruire reti di grandi dimensioni per grandi cluster, gli switch per reti locali hanno lo stesso vantaggio di economia di scala dei piccoli calcolatori.

Come spesso avviene quando due soluzioni sono in competizione, ciascuna delle parti cerca di prendere idee dall'altra per diventare più attraente.

Da un lato, per combattere la debolezza dei sistemi multiprocessore per quanto riguarda la disponibilità, i progettisti hardware e gli sviluppatori dei sistemi operativi cercano di offrire la possibilità di usare più sistemi operativi su porzioni diverse della stessa macchina, in modo tale che un nodo possa guastarsi o essere aggiornato senza interrompere il funzionamento dell'intero sistema.

Dall'altro lato, dato che sia l'amministrazione del sistema che i limiti della dimensione della memoria sono legati in modo approssimativamente lineare al numero delle macchine indipendenti, c'è la tendenza a ridurre i problemi dei cluster costruendo cluster di SMP di piccole dimensioni. Ad esempio un cluster di 32 processori potrebbe essere costruito a partire da otto SMP a quattro vie, o da quattro SMP a otto vie. Questi cluster "ibridi", talvolta chiamati *cluster con memoria condivisa*, si stanno diffondendo per applicazioni critiche per quanto riguarda il rapporto costo-prestazioni, disponibilità ed espandibilità. Infatti tutti i cluster in commercio elencati in figura 9.12, meno uno, sono basati su SMP.

Il paragrafo successivo descrive le reti usate sia dai cluster che dai sistemi multiprocessore.

9.6 Topologie di rete

Il modo più diretto di connettere i nodi processore-memoria è predisporre un collegamento dedicato fra ciascun nodo. Fra l'alto rapporto costo-prestazioni di questa rete *completamente connessa* ed il basso rapporto costo-prestazioni di un bus esiste un insieme di soluzioni di rete che costituiscono un'ampia varietà di compromessi costo-prestazioni. Il costo della rete include il numero di switch, il numero di collegamenti su uno switch per le connessioni di rete, l'ampiezza (numero di bit) del collegamento, e la lunghezza dei collegamenti quando la rete è racchiusa in una macchina fisica. Ad esempio su una macchina scalabile da decine a centinaia di processori alcuni collegamenti possono essere realizzati tramite una piastra metallica lunga pochi millimetri all'interno di un chip, ed altri tramite cavi lunghi parecchi metri tra un armadio e l'altro. Anche le prestazioni delle reti sono sfaccettate: includono la latenza di invio e ricezione dei messaggi nel caso di rete scarica, il throughput in termini di massimo numero di messaggi che possono essere trasmessi in un dato periodo di tempo, i ritardi causati dalla contesa di una porzione di rete, e le variazioni di prestazioni che dipendono dal modello della comunicazione. Un altro obbligo a cui devono sottostare le reti è essere tolleranti ai guasti, dato che i sistemi di grandi dimensioni devono poter operare anche in presenza di componenti non funzionanti.

Le reti normalmente sono disegnate come dei grafi, in cui ciascun arco del grafo rappresenta un collegamento della rete di comunicazione; il nodo processore-memoria è rappresentato da un quadratino nero, e lo switch da un cerchietto colorato. In questo paragrafo tutti i collegamenti sono *bidirezionali*, cioè l'informazione può fluire in entrambe le direzioni; tutte le reti consistono di *switch* i cui collegamenti vanno dai nodi processore-memoria ad altri switch. Il primo miglioramento rispetto al bus è una rete che connette una sequenza di nodi:

Questa topologia è chiamata ad *anello*; dato che non tutti i nodi sono connessi in modo diretto, alcuni messaggi devono attraversare dei nodi intermedi fino ad arrivare alla destinazione finale.

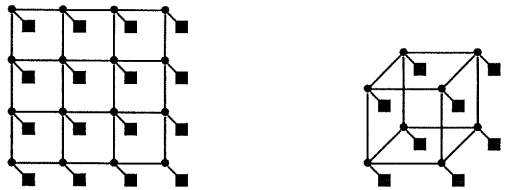
A differenza del bus, l'anello può effettuare più trasferimenti in modo simultaneo. Essendo possibile scegliere fra numerose topologie di rete, è necessario definire delle metriche per misurare le prestazioni di progetti diversi. Di queste, due sono particolarmente diffuse: la prima è la *banda totale della rete*, cioè la banda di ciascun collegamento moltiplicata per il numero di collegamenti. Questa metrica rappresenta il caso migliore: per la rete ad anello con P processori disegnata poco sopra, la banda totale della rete sarebbe P volte la banda di un collegamento; la banda totale della rete nel caso di un bus è semplicemente la banda del bus stesso, o in altre parole una volta la banda del collegamento.

Per controbilanciare il caso migliore, viene inclusa una seconda metrica vicina al caso peggiore: la *banda di bisezione*, calcolata suddividendo la macchina in due parti, ciascuna comprendente metà dei nodi, e sommando la banda dei collegamenti che attraversano la linea immaginaria di suddivisione. La banda di bisezione di una rete ad anello è due volte la banda del collegamento, e quella del bus è una volta tale banda; se un singolo collegamento è veloce quanto un bus, l'anello è due volte più veloce del bus nel caso peggiore, ma è P volte più veloce nel caso migliore.

Dato che alcune topologie di rete non sono simmetriche, possono esserci dei problemi per stabilire dove far passare la linea di suddivisione immaginaria per misurare la banda di bisezione. Essendo questa metrica legata al caso peggiore, la risposta al problema è di scegliere la suddivisione che conduce alle prestazioni di rete peggiori; detto in altre parole, consiste nel calcolare tutte le possibili bande di bisezione e nello scegliere la più bassa. Una visione così pessimistica è giustificata in quanto i programmi per l'elaborazione parallela sono spesso limitati dall'anello più debole della catena della comunicazione.

All'estremo opposto dell'anello si trova la *rete completamente connessa*, in cui ciascun processore ha un collegamento bidirezionale con tutti gli altri processori. Nel caso di reti completamente connesse, la banda totale della rete è pari a $P \times (P-1)/2$ volte quella di un singolo collegamento, e la banda di bisezione è pari a $(P/2)^2$ volte la banda di un collegamento.

L'enorme miglioramento di prestazioni di una rete completamente connessa è controbilanciato da un aumento enorme dei costi: questo ha indotto i progettisti ad inventare nuove topologie di rete che hanno caratteristiche intermedie fra il costo delle reti ad anello e le prestazioni delle reti completamente connesse. La valutazione del successo dipende in gran parte dalla natura della comunicazione all'interno del carico di lavoro dei programmi paralleli in esecuzione sulla macchina.



a. Griglia 2-D o maglia di 16 nodi b. n-cubo di 8 nodi ($8 = 2^3$, per cui $n = 3$)

Figura 9.13 Topologie di rete comparse nei processori paralleli in commercio. I cerchietti colorati rappresentano gli switch, mentre i quadratini neri i nodi processore-memoria; anche se uno switch ha più collegamenti, in generale uno solo va al processore. La topologia ad n-cubo booleano è un'interconnessione n-dimensionale con 2^n nodi, che richiede n collegamenti per switch (più uno per il processore) e che quindi ha n nodi vicini. Queste topologie di base sono frequentemente modificate aggiungendo degli archi per migliorare prestazioni ed affidabilità; ad esempio gli switch nelle colonne di sinistra e di destra della griglia 2-D potrebbero essere connessi tramite la porta inutilizzata di ciascuno switch, ottenendo quattro anelli orizzontali.

È difficile tenere conto di tutte le tipologie di rete discusse nelle pubblicazioni del settore, ma il numero di quelle usate nei processori paralleli commerciali si conta sulle dita di una mano. La figura 9.13 illustra due delle tipologie di rete più diffuse; le macchine reali aggiungono frequentemente dei collegamenti a queste semplici topologie per migliorarne le prestazioni e l'affidabilità.

Anziché sistemare un processore in ciascun nodo della rete si può lasciare soltanto uno switch in alcuni di questi nodi; gli switch sono più piccoli dei nodi processore-memoria-switch, per cui possono essere impaccati in modo più denso, diminuendo le distanze e migliorando le prestazioni. Queste reti sono spesso chiamate *reti multistadio*, per riflettere il fatto che un messaggio deve attraversare più stadi. Le tipologie di rete multistadio sono numerose quanto quelle a stadio singolo: la figura 9.14 mostra due delle organizzazioni multistadio più diffuse. Una *rete completamente connessa* o *rete crossbar* consente a ciascun nodo di comunicare con qualunque altro nodo con un solo passaggio attraverso la rete. Una *rete Omega* usa meno hardware di una rete crossbar ($2n \log_2 n$ invece di n^2 switch), ma possono verificarsi situazioni di conflitto fra messaggi, a seconda della struttura della comunicazione. Ad esempio la rete Omega di figura 9.14 non può inviare un messaggio da P0 a P6 contemporaneamente ad un messaggio da P1 a P7.

Implementare le topologie di rete

La semplice analisi delle reti fatta in questo paragrafo ignora importanti considerazioni pratiche su come vengono costruite le reti. La distanza di ciascun collegamento influisce sul costo della comunicazione alle alte frequenze di clock; in generale più la distanza è elevata, più è costoso utilizzare il collegamento a frequenze di clock elevate. Distanze più brevi inoltre rendono più semplice assegnare più conduttori al collegamento, dato che in un chip la capacità richiesta per pilotare più conduttori è minore se i conduttori sono corti; i conduttori più corti infine sono meno costosi dei conduttori lunghi. Un'ultima limitazione pratica è che le rappresentazioni tridimensionali devono essere implementate su chip e piastre essenzialmente bidimensionali. Topologie che disegnate su una lavagna appaiono eleganti possono diventare complicate e poco eleganti quando realizzate praticamente con chip, cavi, piastre e scatole.

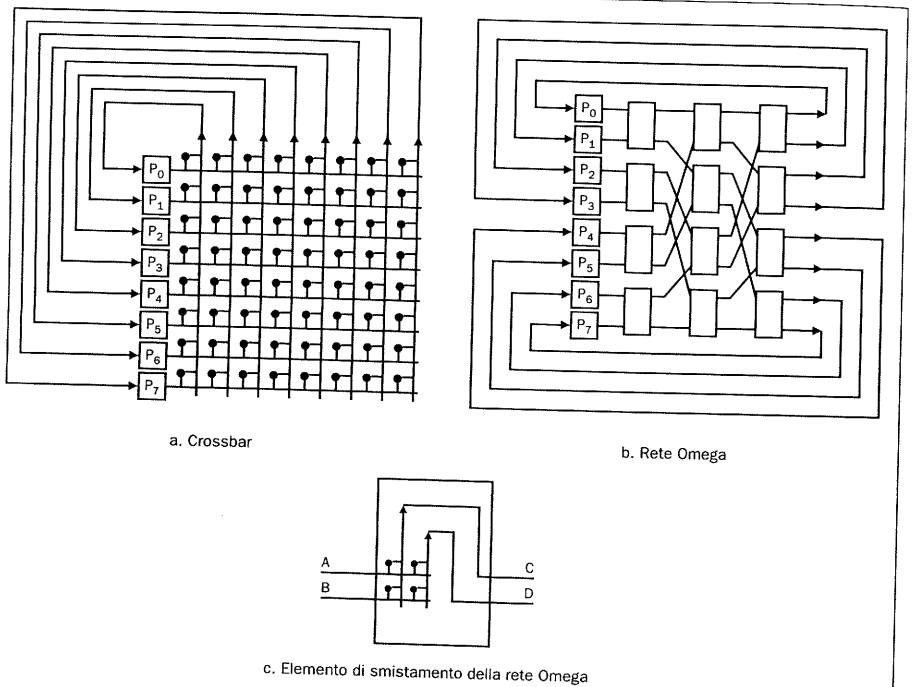


Figura 9.14 Topologie di rete multistadio usate frequentemente nel caso di otto nodi. In queste rappresentazioni gli switch sono più semplici rispetto ai disegni precedenti, perché i collegamenti sono unidirezionali; i dati entrano dal basso ed escono dal collegamento sulla destra. L'elemento di smistamento (switch box) rappresentato in *c* può collegare A a C e B a D oppure B a C e A a D. La rete crossbar usa n^2 switch, dove n è il numero dei processori, mentre la rete Omega utilizza $n/2 \log_2 n$ scatole di smistamento, ciascuna delle quali è costituita da quattro degli switch più piccoli. In questo caso la rete crossbar usa 64 switch rispetto alle 12 scatole di smistamento o ai 48 switch della rete Omega. La rete crossbar, però, può supportare qualunque combinazione di messaggi tra i processori, al contrario della rete Omega.

9.7 Un caso reale: le direzioni di sviluppo dei sistemi multiprocessore

Per oltre un decennio i soliti profeti hanno continuato a sostenere che il modello architetturale basato su un singolo calcolatore aveva raggiunto i propri limiti e che qualunque ulteriore progresso poteva essere realizzato solo attraverso l'interconnessione di un insieme di calcolatori che lavorassero in maniera coordinata per risolvere il problema … È stato dimostrato che l'approccio basato su un solo processore continua ad essere valido …

Gene Amdahl, "Validity of the single processor approach to achieve large scale computing capabilities," Spring Joint Computer Conference, 1967

Le prestazioni dei sistemi ad un solo processore stanno crescendo con una velocità senza precedenti, ed i microprocessori sono alla testa di questo processo. La figura 1.20 mostra che i microprocessori più veloci hanno migliorato le proprie prestazioni di oltre il 50% ogni anno a partire dal 1987. Questa elevata velocità di innovazione non è senza costi: il costo stimato per lo sviluppo del Pentium Pro è di 400 milioni di dollari (circa 650 miliardi di lire).

La capacità delle memorie è cresciuta con un tasso elevato per un periodo considerevolmente più lungo rispetto a quello dei processori: la figura 1.14 mostra che le DRAM hanno accresciuto la loro capacità di quattro volte circa ogni tre anni. Ancora una volta le enormi risorse destinate allo sviluppo, combinate ai bassi prezzi di vendita delle DRAM, hanno indotto quasi tutti i produttori di calcolatori, comprese le aziende produttrici di calcolatori paralleli, a costruire i loro sistemi di memoria utilizzando DRAM.

Un'ulteriore tecnologia è quella delle reti di interconnessione. La larghezza di banda della rete di interconnessione è migliorata grazie ai miglioramenti nella velocità dei circuiti logici, a quelli nel packaging dei processori paralleli, ed ai progressi nella velocità delle connessioni fisiche. Ad esempio l'ampiezza di banda di picco per ciascun canale in un decennio è cresciuta da 0,5 MB/s (Intel iPSC, 1986) a 800 MB/s (SGI Origin, 1996).

In definitiva i progettisti di processori paralleli hanno a disposizione la tecnologia dei microprocessori veloci, quella delle DRAM ad alta capacità, e quella delle reti con ampiezza di banda crescente; è interessante notare che tutte e tre le tecnologie stanno crescendo con velocità confrontabili.

Come già detto in precedenza, i rapidi mutamenti nel mercato dei sistemi multiprocessore hanno indotto gli autori a raccogliere e distribuire tramite World Wide Web le informazioni sui sistemi più esemplificativi, in maniera da evitare che tali esempi diventino superati. Si veda la parte relativa al Capitolo 9 all'indirizzo *www.mkp.com/cod2e.htm* per le informazioni più aggiornate.

Storie vere di processori paralleli su larga scala

La prospettiva esaltante di costruire il calcolatore più veloce del mondo è vincolata ad alcune esperienze vissute dai progettisti di processori paralleli. La prima esperienza consiste nello scoprire che i nodi di un calcolatore parallelo sono molto simili al nucleo di un calcolatore da tavolo, e quindi il costo di un nodo di un calcolatore parallelo di grandi dimensioni è confrontabile con quello di un calcolatore da tavolo.

La maggior parte dei supercomputer costano, per quanto riguarda processori e memoria, meno di 25000000 dollari; poiché il prezzo dei calcolatori da tavolo è stabile tra i 2500 ed i 10000 dollari, anche se i nodi potessero avere lo stesso prezzo dei calcolatori da tavolo, i più grandi sistemi paralleli non potrebbero avere più di 2500-10000 nodi. Una simile stima non tiene conto del costo della rete di interconnessione, e questo riduce ulteriormente il numero dei nodi possibili. Inoltre molti calcolatori destinati ad applicazioni di tipo scientifico vengono acquistati a prezzi molto inferiori; questi elaboratori hanno quindi molti meno nodi del massimo previsto. Di conseguenza, mentre il massimo numero possibile di processori varia tra 1000 e 10000, vi sono molti utenti e molte applicazioni per i quali 100 processori sono più che sufficienti.

Nel 1997 la maggior parte dei sistemi multiprocessore ricadeva nella categoria di quelli composti da 8-19 processori, ed il numero medio di processori sta crescendo

solo lentamente. Se il 95% degli elaboratori venduti contiene meno di 100 processori, quel 5% di elaboratori aventi dimensioni maggiori dovrà sopportare il costo di una ricerca e di uno sviluppo più sofisticati e quindi presentare un costo per nodo più elevato.

La topologia della rete di interconnessione è fondamentale nel progetto di un sistema in grado di essere configurato in maniera variabile tra 100 e 10000 nodi, ed è possibile che la topologia migliore per un sistema di 100-500 nodi non sia quella ideale per un sistema da 1000-10000 nodi. La topologia può quindi variare a seconda del numero massimo di nodi e delle scelte di configurazione per il sistema. Sul fronte delle buone notizie vi sono molte alternative valide per le reti di interconnessione; sul fronte delle cattive notizie invece vi è una situazione in cui, a fronte di tutte queste possibilità, e vista l'importanza della topologia dell'interconnessione nella definizione del costo del sistema, è improbabile che vi sia una sola topologia in grado di soddisfare le esigenze di tutte le aziende produttrici di sistemi paralleli.

La seconda ragione che riduce l'importanza della topologia di interconnessione è il fatto che la tolleranza ai guasti, per essere economicamente competitiva, è incompatibile con algoritmi dipendenti dalla topologia, dal momento che una connessione o un nodo guasti significano che necessariamente vi saranno dei messaggi che seguiranno strade diverse da quelle previste dal programmatore in base alla topologia. La tolleranza ai guasti è un aspetto critico in quanto un sistema con 10000 nodi, ciascuno simile ad un calcolatore da tavolo, è caratterizzato da un tempo medio tra due guasti 10000 volte inferiore a quello di un calcolatore da tavolo. Di conseguenza quando si tratta di sistemi paralleli di grandi dimensioni il punto non è *se* vi sia un componente guasto, ma piuttosto *quanti* componenti siano guasti. I processori paralleli devono poter lavorare in presenza di guasti nelle connessioni di rete e nei nodi, e quindi i sistemi paralleli di grandi dimensioni non possono utilizzare algoritmi specifici di una topologia anche se questi riducono il costo della comunicazione.

Interfaccia Hardware Software

La mancanza di una topologia standard non rappresenta un ostacolo per la scrittura di programmi portabili sui sistemi paralleli, come a prima vista si potrebbe pensare. Una spiegazione di questo fatto è che il ritardo introdotto dal software nelle operazioni di invio di un messaggio è così elevato da mascherare gli effetti della topologia: in altre parole questi ritardi sono così elevati che il tempo per inviare un messaggio al nodo più vicino è confrontabile con quello per inviarlo al nodo più lontano. In alcuni casi il ritardo è elevato perché il protocollo è progettato per l'invio di messaggi di grandi dimensioni e quindi la tecnica usata è quella di combinarli, in maniera da dover tollerare una volta sola il tempo di latenza. Altre ragioni sono legate all'elevato costo delle chiamate al sistema operativo per l'invio o la ricezione di un messaggio ed alla lentezza dell'interfaccia tra il processore e la rete.

Se combinati, i seguenti elementi riducono l'importanza degli algoritmi basati su una specifica topologia:

- La mancanza di una topologia standard e l'importanza di programmi portabili per processori paralleli

- L'elevato costo della comunicazione, che fa sì che il tempo di latenza sia virtualmente lo stesso per tutti i messaggi, indipendentemente dalla distanza tra i nodi connessi
- La necessità di operare anche in presenza di guasti nelle connessioni e nei nodi.

Quadro d'insieme

Una caratteristica fondamentale dei programmi paralleli è la frequenza delle operazioni di sincronizzazione e di comunicazione. I sistemi paralleli di grandi dimensioni possiedono una memoria fisica distribuita; la maggiore ampiezza di banda e il minor ritardo nell'accesso alla memoria locale rispetto a quella non locale fanno sì che siano fortemente privilegiati i programmatori che sfruttano la località.

Il parallelismo massiccio

Il termine *parallelismo massiccio* è stato ampiamente utilizzato ma raramente definito, anche se nessuno definirebbe massicciamente parallelo un calcolatore con meno di 100 processori. Anche adottando una simile definizione conservativa, l'elaborazione parallela basata su sistemi con più di 100 processori non è più così rilevante nella situazione attuale. Dal punto di vista teorico si dovrebbe avere un modello semplice in grado di permettere ai programmatori di creare con facilità programmi paralleli portabili e capaci di ottenere buone prestazioni su sistemi multiprocessore reali, e contemporaneamente di permettere ai ricercatori di inventare nuovi algoritmi che funzionino bene su molti sistemi multiprocessore. Sfortunatamente vi sono ben pochi modelli teorici relativi all'elaborazione parallela che possano predire con accuratezza le prestazioni dei sistemi multiprocessore oggi disponibili in commercio.

Un evento di grande interesse per il settore dei sistemi massicciamente paralleli (Massively Parallel Processors, MPP) è costituito dal programma denominato Accelerated Strategic Computing Initiative (ASCI), finalizzato ad accelerare lo sviluppo di un supercalcolatore massicciamente parallelo attraverso l'acquisto di un piccolo numero di elaboratori con costo variabile tra i 50 ed i 100 milioni di dollari. Entro l'inizio del prossimo secolo verrà costruita una serie di supercalcolatori ASCI che culmineranno con un supercalcolatore della potenza di 100 teraFLOPS. La figura 9.15 mostra le prime tre macchine di questa serie. Nel dicembre 1996 l'elaboratore a memoria distribuita tramite scambio di messaggi prodotto da Intel (ASCI Red) è stato il primo calcolatore a raggiungere la velocità di 1 teraFLOPS (pari ad un milione di MFLOPS).

Nome	Numero di processori	Processore	Dimensione della memoria/sistema	Ampiezza di banda/collegamento	Nodo	Topologia	Prestazioni di picco (teraFLOPS)	Anno
ASCI Red (Intel)	9216	Pentium Pro 200 MHz	580608 MB	800 MB/s	SMP a due vie	2 griglie 2D	1,8	1996
ASCI Blue Pacific (IBM RS/6000SP)	–	successore del del Power2 SC	–	–	–	–	3,0	1998
ASCI Blue Mountain (SGI Origin2000)	3072	successore del MIPS R10000	–	–	–	–	3,0	1998

Figura 9.15 Le caratteristiche dei calcolatori ASCI. Sono tutti elaboratori a memoria distribuita, ma solo l'elaboratore SGI comunica attraverso un indirizzo condiviso e garantisce la coerenza delle cache. Il costo dei tre elaboratori è pari a 46 milioni di dollari (74 miliardi di lire), 93 milioni di dollari (150 miliardi di lire) e 110 milioni di dollari (176 miliardi di lire), rispettivamente. Si veda l'indirizzo *www.mkp.com/cod2e.htm* per avere dei puntatori a questi elaboratori e ad altri più recenti.

9.8 Errori e trabocchetti

Numero 9: si definiscano le prestazioni in termini di uso del processore, accelerazione dovuta alla presenza di processori multipli o MFLOPS per dollaro.

David H. Bailey, "Dodici modi per ingannare le masse nella diffusione di dati sulle prestazioni dei supercalcolatori paralleli," *Supercomputing Review*, 1991

L'estremo interesse destato dall'elaborazione parallela ha fatto nascere molti errori e trabocchetti: eccone tre.

Trabocchetto: misurare le prestazioni dei sistemi paralleli usando l'accelerazione lineare in funzione del tempo di esecuzione.

Grafici a parabola, che tracciano l'andamento delle prestazioni in funzione del numero di processori, e che mostrano un'accelerazione lineare, un tratto orizzontale e poi una caduta sono stati utilizzati a lungo per valutare il successo dei sistemi paralleli. Anche se la scalabilità è solo uno degli aspetti di un programma parallelo, essa costituisce una misura indiretta delle prestazioni. La domanda fondamentale da porsi riguarda la potenza dei processori di cui si valuta la scalabilità: un programma che migliora linearmente le prestazioni fino ad uguagliare 100 processori Intel 8086 può essere più lento della versione sequenziale dello stesso programma che lavora su un calcolatore da tavolo Pentium Pro. Si deve fare particolare attenzione con i programmi che usano pesantemente le operazioni in virgola mobile, dal momento che gli elementi di elaborazione privi di supporto hardware per i calcoli in virgola mobile possono essere scalabili in maniera fantastica ma avere cattive prestazioni nel loro complesso.

La misura dei risultati utilizzando l'accelerazione lineare rispetto al tempo di esecuzione può trarre in inganno tanto il programmatore quanto coloro che ascoltano dal programmatore le dichiarazioni relative alle prestazioni del programma.

Molti programmi con un ridotto aumento di velocità sono più veloci di altri che ne mostrano uno eccezionale quando il numero di processori cresce.

Il confronto tra i tempi di esecuzione è corretto solo se si stanno confrontando gli algoritmi migliori disponibili su ciascun elaboratore. Ovviamente non è possibile sottrarre il tempo in cui i processori restano fermi allorché si valuta un sistema parallelo, cosicché il tempo di CPU non è una metrica appropriata per i sistemi paralleli. Il confronto tra i tempi di esecuzione dello stesso codice su due sistemi diversi può sembrare una scelta più corretta, ma non lo è: il programma parallelo può essere più lento su un sistema multiprocessore che nella versione sequenziale. A volte lo sviluppo di un programma parallelo conduce a dei miglioramenti algoritmici, cosicché il confronto tra il miglior programma sequenziale noto in precedenza e il codice parallelo (confronto apparentemente corretto) in realtà confronta gli algoritmi in maniera inappropriata. Per tener conto di questi aspetti si utilizzano a volte i termini *accelerazione relativa* (allo stesso programma) o *relative speedup* e *accelerazione vera* (relativa ai programmi migliori) o *true speedup*.

Errore: la legge di Amdahl non si applica ai calcolatori paralleli.

Nel 1987 il responsabile di un'organizzazione di ricerca sostenne che la legge di Amdahl era stata contraddetta da un sistema multiprocessore. Per cercare di capire su quale base i mezzi di comunicazione riportarono la notizia, ecco la formulazione della legge di Amdhal [1967, pag. 483]:

> *una conclusione abbastanza ovvia che può essere tratta a questo punto è che lo sforzo speso per raggiungere elevate velocità di elaborazione parallela è sprecato, a meno che non sia accompagnato da miglioramenti di valore assolutamente uguale nella velocità di elaborazione sequenziale.*

Questa affermazione deve comunque essere vera: la parte del programma non interessata dall'architettura parallela limita le prestazioni. Un'interpretazione di questa legge conduce al seguente lemma: alcune parti di ciascun programma devono essere sequenziali cosicché deve esserci un limite superiore di tipo economico al numero di processori (ad esempio 100). Se si osserva un'accelerazione lineare con 1.000 processori si dimostra che questo lemma è falso e quindi che è falsa anche l'ipotesi che la legge di Amdhal non sia più valida.

L'approccio seguito dai ricercatori era basato sul modificare i valori in ingresso all'applicazione campione: anziché cercare di andare 1000 volte più veloce essi cercarono di eseguire una quantità di lavoro 1000 volte superiore in un tempo confrontabile. Per il loro algoritmo la parte sequenziale del programma era di entità costante e indipendente dalla dimensione degli ingressi mentre la parte restante era completamente parallela: di conseguenza su questa parte era possibile ottenere un'accelerazione lineare utilizzando 1000 processori. È ingenuo pensare che sia scalabile la sola dimensione delle applicazioni, senza che lo siano anche la precisione delle operazioni in virgola mobile, il numero di iterazioni, le esigenze di I/O e il modo in cui le applicazioni gestiscono gli errori. Molte applicazioni non sono in grado di calcolare il risultato corretto se la dimensione del problema cresce senza limiti.

Non si vedono ragioni per cui la legge di Amdhal non debba poter essere applicata ai processori paralleli: ciò che viene evidenziato da queste ricerche è l'importanza di disporre di applicazioni campione che possano crescere a sufficienza, tanto da permettere la valutazione delle prestazioni di sistemi paralleli su larga scala.

Trabocchetto: le prestazioni di picco ricalcano il comportamento delle prestazioni osservate.

Sistema	Valutazione in MFLOPS di picco	Media armonica in MFLOPS delle applicazioni Perfect Club	Percentuale rispetto ai MFLOPS di picco
Cray X-MP/416	940	14,8	1%
IBM 3090-600S	800	8,3	1%
NEC SX/2	1300	16,6	1%

Figura 9.16 Prestazioni di picco e media armonica delle prestazioni reali per le 12 applicazioni campione Perfect Club. I risultati sono stati ottenuti facendo eseguire i programmi senza modifiche. Se li si ottimizza a mano le prestazioni dei tre sistemi crescono a 24,4, 11,3 e 18,3 MFLOPS rispettivamente. Queste misure sono ancora inferiori al 2% delle prestazioni di picco.

	Cray YMP (8 processori)		IBM SP2 (64 processori)	
	MFLOPS	% Picco	MFLOPS	% Picco
Picco	2666	100%	14636	100%
3D FFT PDE	1795	67%	1093	7%

Figura 9.17 Prestazioni di picco e prestazioni osservate per Cray YMP e IBM RS/6000 SP2.

Una possibile definizione per le prestazioni di picco è la seguente: prestazioni che un elaboratore non può in nessuna maniera superare. L'industria dei supercalcolatori usa questa metrica nel marketing e la sua falsità è enfatizzata nel caso dei sistemi paralleli. Coloro che operano nel settore commerciale non solo usano il concetto di prestazioni di picco (quasi raggiungibili) per un nodo a singolo processore (si veda la figura 9.16), ma essi eseguono anche la moltiplicazione dello stesso per il numero totale di processori, dando per scontata un'accelerazione perfetta. La legge di Amdhal illustra quanto sia difficile raggiungere l'uno e l'altro picco; il moltiplicare l'uno per l'altro moltiplica anche l'errore. La figura 9.17 confronta le prestazioni di picco con quelle a regime su un'applicazione campione: il sistema a 64 processori IBM SP2 raggiunge solamente il 7% delle prestazioni di picco. È chiaro quindi che le prestazioni di picco non sempre seguono il comportamento delle prestazioni osservate.

Tali prestazioni, nella forma in cui vengono diffuse, possono confondere sia il produttore dell'elaboratore che il suo utente. Il pericolo consiste nel fatto che il produttore sviluppi delle librerie di software il cui successo è valutato come percentuale delle prestazioni di picco misurate in MFLOPS anziché come tempo richiesto, oppure che aggiunga dell'hardware in grado di aumentare le prestazioni di picco del nodo, ma ben difficile da utilizzare.

9.9 Note conclusive: evoluzione e rivoluzione nell'architettura dei calcolatori

Il modo incerto con cui necessariamente si muove anche il più abile degli scienziati nell'attraversare le foreste di osservazioni errate, di generalizzazioni fuorvianti, di formulazioni inadeguate e di pregiudizi inconsci è raramente compreso da coloro che derivano dai libri la loro cultura scientifica.

James B. Conant, *Science and Common Sense*, 1951

La lettura degli articoli comparsi negli ultimi trent'anni sugli atti delle conferenze e sulle riviste può essere scoraggiante: a fronte di tanti sforzi i risultati sono stati minimi. A voler essere ottimisti, tutti quegli articoli fungono da massicciata e, presi nel loro complesso, costituiscono le fondamenta della prossima generazione di calcolatori; a voler invece essere più pessimisti si potrebbe dire che se il 90% di quelle idee scomparisse nessuno se ne accorgerebbe.

Una delle ragioni di questa situazione è quella che potrebbe essere chiamata "sindrome di Von Neumann": sperando di inventare un nuovo modello di calcolo che rivoluzioni l'informatica, i ricercatori si sforzano di diventare i Von Neumann del XXI secolo. Un'altra ragione è il fatto che i ricercatori spesso scelgono problemi che non interessano a nessun altro. Anche se vengono scelti dei problemi importanti vi è spesso una mancanza di supporto sperimentale nel dimostrare in maniera convincente il valore di una soluzione; inoltre quando si scelgono dei problemi importanti e si dimostra la validità delle soluzioni proposte, è possibile che queste abbiano un costo eccessivo rispetto ai benefici che portano. A volte questo costo viene valutato direttamente attraverso il rapporto costo-prestazioni, ed il miglioramento delle prestazioni non giustifica il costo aggiuntivo. Più spesso il costo dell'innovazione deriva dall'impatto sugli utenti di calcolatori.

La figura 9.18 mostra ciò che si intende per *spettro evoluzione-rivoluzione* dell'innovazione nell'architettura dei calcolatori. A sinistra vi sono le idee invisibili per gli utenti (ad eccezione presumibilmente di un minor costo, migliori prestazioni o entrambi); questo è l'estremo evolutivo dello spettro. All'altro capo vi sono le idee

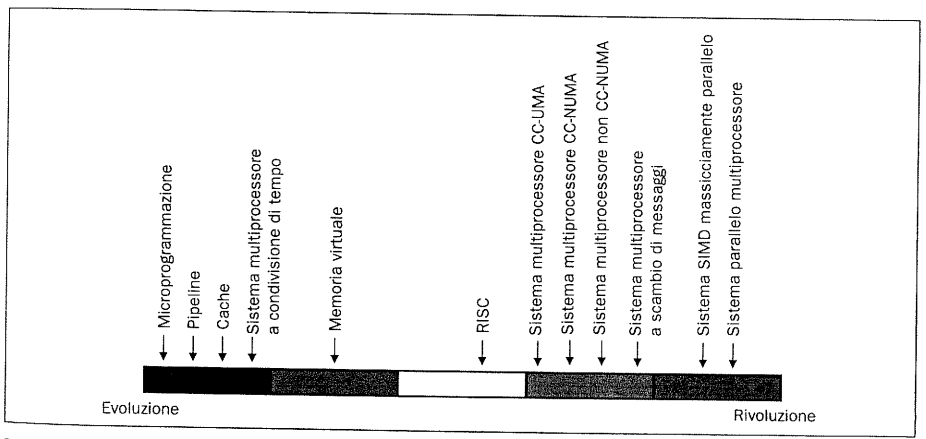


Figura 9.18 Lo spettro evoluzione-rivoluzione relativo all'architettura dei calcolatori. Le prime quattro colonne si distinguono dall'ultima in quanto le applicazioni ed i sistemi operativi possono essere portati da un calcolatore all'altro anziché essere riscritti da capo. Per esempio RISC compare nella parte centrale dello spettro poiché la sua compatibilità è relativa al solo livello dei linguaggi di alto livello, mentre la microprogrammazione permette la compatibilità del codice binario ed i sistemi paralleli multiprocessore richiedono delle modifiche algoritmiche e delle estensioni dei linguaggi di alto livello. Nella figura compaiono diverse categorie di sistemi multiprocessore. "Sistema multiprocessore a condivisione di tempo" indica un sistema multiprocessore nel quale diversi programmi tra loro indipendenti lavorano contemporaneamente. "CC-UMA" e "CC-NUMA" indicano dei sistemi multiprocessore con coerenza delle cache di tipo UMA e NUMA che eseguono sottosistemi paralleli quali basi di dati o file server. "nonCC-NUMA" indica un sistema di tipo NUMA ad indirizzi condivisi ma senza coerenza delle cache per applicazioni dello stesso tipo. Le stesse applicazioni sono quelle presupposte per i sistemi "a scambio di messaggi". "Sistema parallelo multiprocessore" indica un sistema multiprocessore dello stesso tipo di quelli venduti per accelerare specifici programmi sviluppati da utenti. Si veda il Paragrafo 9.10 per la definizione di SIMD.

architetturali di tipo rivoluzionario: tali idee richiedono che i programmatori di nuove applicazioni debbano imparare nuovi linguaggi di programmazione e nuovi modelli di elaborazione, e debbano proporre nuove strutture dati e nuovi algoritmi.

Le idee di tipo rivoluzionario sono più facili da pubblicare di quelle evolutive, ma per essere adottate devono presentare vantaggi molto maggiori. Le cache sono un esempio di miglioramento di tipo evolutivo: nel giro di cinque anni dalla prima pubblicazione sulle cache, praticamente tutte le aziende di calcolatori progettarono un elaboratore dotato di cache. L'idea di un calcolatore con un set ridotto di istruzioni (RISC) si trova vicino alla metà dello spettro e ci vollero quasi dieci anni perché la maggior parte delle aziende disponesse di un prodotto RISC. Diverse aziende di calcolatori paralleli sorte alla fine degli anni '80 forniscono esempi di architetture di calcolatori di tipo rivoluzionario: qualunque programma eseguito efficientemente su tali elaboratori era una versione profondamente modificata di un programma originale, oppure era stato scritto espressamente per quell'architettura, ed i programmatori dovevano imparare un nuovo stile di programmazione. Gli ostacoli furono troppo grandi perché gran parte di queste aziende potesse sopravvivere.

Progetti ignorati dall'industria informatica potrebbero essere importanti se servissero a far ricordare in futuro ciò che si è imparato. La colpa non consiste nel produrre una nuova architettura commercialmente fallimentare bensì nel trascurare la valutazione quantitativa dei punti di forza e di debolezza delle nuove idee. Il fallimento di progetti di ricerca passati non significa che le relative idee siano morte per sempre: le mutazioni nella tecnologia possono far resuscitare un'idea che in precedenza non era conveniente oppure che era troppo avanzata rispetto alla tecnologia.

Quando si guarda al futuro e si inventa il proprio contributo in un settore, si deve avere presente lo spettro evoluzione-rivoluzione. Quanto bene saranno accolte le idee relative all'hardware dipende dall'accoglienza riservata loro dai programmatori: di conseguenza i progettisti di hardware devono saperne di più a proposito del software. Se i programmatori vogliono dei buoni elaboratori devono saperne di più a proposito dell'hardware, per essere in grado di comunicare e quindi di influenzare i progettisti. Inoltre si tengano presenti i principi base dell'organizzazione dei calcolatori presentati in questo libro: essi saranno sicuramente di guida per i calcolatori del futuro, esattamente come lo sono stati per i calcolatori del passato.

9.10 Inquadramento storico e approfondimenti

Poiché si può parlare di parallelismo a diversi livelli, è utile disporre di una classificazione delle varie possibilità esistenti. Nel 1966 Flynn propose un semplice modello di classificazione per i calcolatori che è ancor oggi utilizzato. Osservando le componenti elementari di un elaboratore, egli propose di contare il numero di flussi paralleli di istruzioni e di dati e di caratterizzare i calcolatori sulla base di questo valore:

1. *Un flusso di istruzioni, un flusso di dati* (Single instruction stream, Single data stream, o SISD)
2. *Un flusso di istruzioni, più flussi di dato* (Single instruction stream, Multiple data streams, o SIMD)

3. *Più flussi di istruzioni, un flusso di dati* (Multiple instruction streams, Single data stream, o MISD)
4. *Più flussi di istruzioni, più flussi di dati* (Multiple instruction streams, Multiple data streams, o MIMD).

Alcuni sistemi di elaborazione rappresentano degli ibridi rispetto a queste categorie, come è ovvio, ma questo modello di classificazione è sopravvissuto grazie alla sua semplicità e facilità di comprensione, e rappresenta un buon punto di partenza. Forse proprio grazie alla sua comprensibilità inoltre è lo schema di classificazione più utilizzato.

La prima domanda che ci si può porre a proposito di questo modello è "Un flusso o più flussi rispetto a che cosa?" Si potrebbe pensare che un elaboratore che somma un numero di 32 bit in un solo colpo di clock abbia più flussi di dati se confrontato con un elaboratore seriale che richiede 32 colpi di clock per eseguire la somma. Flynn usò come modello dei sistemi SISD i calcolatori più diffusi in quel momento, ossia gli IBM 704 e gli IBM 7090, mentre le implementazioni dell'architettura MIPS introdotte nei Capitoli 5 e 6 possono oggi costituire dei buoni punti di riferimento.

I calcolatori ad un flusso di istruzioni e più flussi di dati

I calcolatori SIMD lavorano su vettori di dati. Quando ad esempio una singola istruzione SIMD somma 64 numeri, la circuiteria SIMD invia 64 flussi di dati a 64 ALU in modo da calcolare 64 somme in un solo colpo di clock.

L'aspetto interessante dei sistemi SIMD sta nel fatto che tutte le unità di elaborazione parallela sono sincronizzate ed eseguono una stessa istruzione che viene indirizzata da uno stesso program counter. Dal punto di vista del programmatore, questo modello è simile a quello ben noto dei sistemi SISD: anche se ciascuna unità esegue la stessa istruzione, ciascuna unità di esecuzione possiede i propri registri di indirizzo, così che ciascuna unità può utilizzare indirizzi di dato diversi.

La motivazione originaria che giustificava i sistemi SIMD era la possibilità di ammortizzare il costo dell'unità di controllo su decine di unità di esecuzione. Un altro vantaggio risiede nella ridotta dimensione della memoria di programma, dal momento che un sistema SIMD richiede una sola copia del codice in corso di esecuzione, mentre i sistemi MIMD possono richiedere una copia in ciascun processore. Il meccanismo della memoria virtuale e le maggiori capacità dei chip DRAM hanno ridotto l'importanza di questo vantaggio.

I calcolatori SIMD in realtà possiedono un misto di istruzioni SISD e SIMD. Normalmente vi è un calcolatore host di tipo SISD che esegue le operazioni sequenziali quali i salti o il calcolo degli indirizzi; le istruzioni SIMD sono invece inviate a tutte le unità di esecuzione, ciascuna dotata del proprio insieme di registri e della propria memoria. Le unità di esecuzione sfruttano una rete di interconnessione per scambiarsi i dati.

I sistemi SIMD lavorano al meglio quando hanno a che fare con vettori all'interno di cicli *for*. Di conseguenza perché il parallelismo massiccio sia efficace nei sistemi SIMD bisogna che vi siano grandi quantità di dati da elaborare (*parallelismo nei dati*). I sistemi SIMD si trovano invece in difficoltà in corrispondenza di costrutti *case* o *switch*, nei quali ciascuna unità di esecuzione deve eseguire una diversa operazione sui propri dati locali, a seconda del valore di tali dati. Le unità di esecuzione con i dati sbagliati sono disabilitate, mentre quelle con i dati corretti possono

Produttore	Nome	Massimo numero di processori	Bit/processore	Frequenza di clock del processore (MHz)	Numero di FPU	Massima dimensione di memoria per sistema (MB)	Ampiezza di banda di comunicazione per sistema (MB/s)	Anno
Università dell'Illinois	Illiac IV	64	64	5	64	0,125	2560	1972
ICL	DAP	4096	1	5	0	2	2560	1980
Goodyear	MPP	16384	1	10	0	2	20480	1982
Thinking Machines	CM-2	65536	1	7	2048 (opzionali)	512	16384	1987
Maspar	MP-1216	16384	4	25	0	256 o 1024	23000	1989

Figura 9.19 Caratteristiche di cinque calcolatori SIMD. Il numero di FPU indica il numero di unità per l'aritmetica in virgola mobile.

continuare l'elaborazione. Tali situazioni sono caratterizzate da un calo di prestazioni pari a n volte, dove n è il numero di clausole del costrutto.

Un parametro cruciale negli elaboratori SIMD è dato dalle prestazioni del processore rispetto al numero di processori. Il sistema Connection Machine 2 (CM-2) ad esempio si basa su 65536 processori con ampiezza pari ad un bit, mentre il sistema Illiac IV ha 64 processori con parallelismo pari a 64 bit. La figura 9.19 elenca le caratteristiche di alcuni sistemi SIMD.

Il sistema SIMD più conosciuto è sicuramente lo Illiac IV (si veda la figura 9.20), che rappresenta forse il progetto più disastroso nel campo dei supercalcolatori. Anche se rappresentò un successo in quanto stimolo di numerose tecnologie utilizzate poi in progetti successivi, lo Illiac IV fu un fallimento come calcolatore. I costi passarono dagli 8 milioni di dollari stimati nel 1966 ai 31 milioni di dollari del 1972,

Figura 9.20 L'unità di controllo dello Illiac IV e sullo sfondo i suoi 64 elementi di elaborazione. Fu probabilmente il più disastroso tra i supercalcolatori. Il progetto partì nel 1965 ed eseguì la prima applicazione reale nel 1976. I 64 processori utilizzavano un clock a 13 MHz e la dimensione complessiva della memoria principale era pari ad 1 MB (64 × 16 KB). Lo Illiac IV fu il primo sistema a dimostrare che il software per i sistemi paralleli è più importante dell'hardware. Fotografia gentilmente concessa dallo Ames Research Center della NASA.

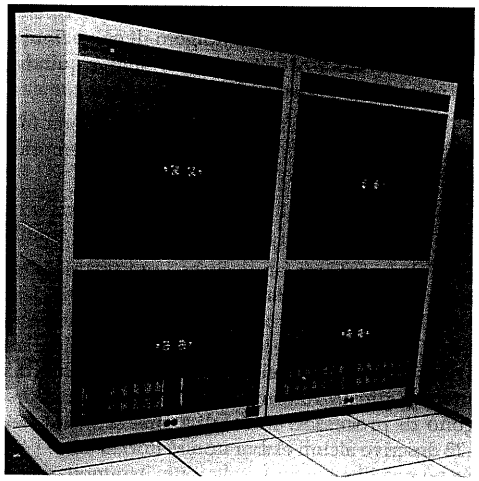


Figura 9.21 Lo MPP di Goodyear (16.384 processori). Fu consegnato il 2 maggio 1983 al Goddard Space Center della NASA e fu operativo il giorno seguente. Fu dismesso il 1 marzo 1991.

malgrado nel frattempo fosse stato costruito solo un quarto del sistema. Le prestazioni effettive erano al più pari a 15 MFLOPS in confronto ai 1000 MFLOPS delle previsioni iniziali per il sistema completo (si veda Falk [1976]). Consegnato allo Ames Research Center della NASA nel 1972 il sistema richiese altri tre anni di ingegnerizzazione prima di divenire operativo. Malgrado tutto questo i progettisti di calcolatori non si scoraggiarono: i successori di tipo SIMD dell'Illiac IV furono tra l'altro il DAP di ICL, lo MPP di Goodyear (si veda la figura 9.21), le CM-1 e CM-2 di Thinking Machines e gli MP-1 ed MP-2 di Maspar.

I calcolatori vettoriali

Un modello connesso al SIMD è quello dell'*elaborazione vettoriale*. Si tratta di un'architettura e di un modello di compilazione oggi ben assestati che si diffuse grazie ai supercalcolatori, e che è considerevolmente più utilizzato rispetto al modello SIMD. I processori vettoriali dispongono di operazioni di alto livello in grado di lavorare su schiere lineari di numeri, o vettori. Un esempio di operazione su vettori è

$$A = B \times C$$

dove A, B e C sono vettori di 64 elementi corrispondenti ciascuno ad un numero in virgola mobile di 64 bit. I sistemi SIMD hanno delle istruzioni analoghe; la differenza risiede nel fatto che i processori vettoriali si basano su unità funzionali con pipeline che operano tipicamente su pochi elementi per ciascun colpo di clock, mentre un sistema SIMD lavora generalmente su tutti gli elementi in una volta sola.

I vantaggi dei calcolatori vettoriali rispetto ai processori SISD tradizionali sono tra l'altro:

1. Ciascun risultato è indipendente dai risultati precedenti, e questo consente di utilizzare pipeline più profonde e frequenze di clock più elevate.

2. Ciascuna istruzione vettoriale esegue una considerevole mole di lavoro, e questo comporta un minor numero di operazioni di prelievo delle istruzioni, un numero minore di istruzioni di salto, e quindi un numero minore di salti con previsione errata.
3. Le istruzioni vettoriali accedono alla memoria in blocchi, e questo permette di ammortizzare il tempo di accesso alla memoria su un numero di elementi elevato (ad esempio 64).
4. Le istruzioni vettoriali accedono alla memoria secondo schemi noti, e questo permette a più banchi di memoria di fornire simultaneamente gli operandi.

Gli ultimi due vantaggi fanno sì che i processori vettoriali non abbiano la necessità di disporre di cache di dato con frequenze di hit elevate per raggiungere buone prestazioni. Essi tendono a basarsi su memorie principali a bassa latenza, spesso costituite di SRAM, e ad avere un numero elevato di banchi di memoria (ad esempio 1024) per raggiungere un'elevata ampiezza di banda verso la memoria.

Per ottenere prestazioni ancora più elevate tutti i supercalcolatori vettoriali dispongono di processori multipli, come si vedrà nel prossimo paragrafo. La figura 9.22 descrive alcuni elaboratori vettoriali.

Approfondimento: Anche se la classificazione di Flynn prevede i sistemi MISD, questi sono difficili da immaginare. Un flusso di istruzioni singolo è più semplice di un flusso di istruzioni multiplo, ma flussi di istruzioni multipli insieme con flussi di dati multipli (MIMD) sono più semplici da immaginare rispetto ad istruzioni multiple con un singolo flusso di dati (MISD).

Anche se si può pensare di scrivere 100 programmi diversi per 100 processori diversi appartenenti ad un sistema MIMD, in pratica questo non è possibile. Oggi i programmatori di sistemi MIMD scrivono un solo programma sorgente e ragionano come se lo stesso programma lavorasse su tutti i processori. Un tale approccio è a volte definito come *singolo programma con dati multipli* (single program multiple data, o SPMD).

I calcolatori con più flussi di istruzioni e più flussi di dati

È difficile identificare il primo sistema MIMD: le argomentazioni a favore dell'elaborazione parallela possono essere fatte risalire fino al XIX secolo [Menabrea 1842]! Già il primo calcolatore, prodotto dalla Eckert-Mauchly Corporation, possedeva unità duplicate, in quel caso per aumentarne l'affidabilità.

Nome	Registri vettoriali	Elementi per registro vettoriale	Elementi elaborati per ciclo di clock	Numero di unità funzionali	Frequenza di clock del processore	Massimo numero di processori	Massima dimensione di memoria/sistema
Cray J90	8	64	1	4	100 MHz	32	8192 MB
Cray T90	8	128	2	8	455 MHz	32	8192 MB
Fujitsu VPP300	8-256	64-2048	8	4	140 MHz	16	32768 MB
NEC SX-4 (singolo nodo)	8 + 8192 (scratchpad)	256 + numero variabile sino a 8K	8	16	125 MHz	32	8192 MB

Figura 9.22 Caratteristiche di quattro calcolatori vettoriali in vendita nel 1997. Tutti i calcolatori vettoriali ~~descritti~~ in questa tabella (ad eccezione del Cray T90) utilizzano la tecnologia CMOS per il processore. Per ridurre ~~la latenza~~ della memoria principale (al costo di aumentare il prezzo del sistema) il T90 utilizza delle SRAM anziché ~~le~~ DRAM.

descritti la latenza delle

Due dei più noti progetti per la costruzione di un sistema multiprocessore furono lanciati negli anni '70 alla Carnegie-Mellon University. Il primo di questi fu C.mmp, che consisteva in 16 PDP-11 connessi a 16 unità di memoria attraverso un crossbar switch. Il sistema era tra i primi ad essere dotato di più di qualche processore, ed era basato su un modello di programmazione a memoria condivisa. Molta attenzione nel progetto C.mmp era rivolta al software, in particolare nell'area dei sistemi operativi. Un elaboratore successivo, il Cm*, era un sistema multiprocessore basato su cluster di processori con una memoria distribuita ed un tempo di accesso non uniforme, e questo rendeva la sua programmazione ancora più difficile. L'assenza di cache ed un elevato tempo di latenza per gli accessi remoti rendeva critica la collocazione dei dati.

Benché negli anni '70 si fossero costruiti grandi mainframe con più processori, i sistemi multiprocessore non ebbero successo sino agli anni '80. Bell [1985] suggerisce che la chiave per il loro successo sia stata la minor dimensione dei microprocessori, che permise ai bus di memoria di sostituire la circuiteria delle reti di interconnessione, mentre la disponibilità di sistemi operativi portabili fece sì che il progetto di un sistema parallelo non richiedesse più l'invenzione di un nuovo sistema operativo. Bell distingue i sistemi paralleli tra quelli dotati di spazi di indirizzamento multipli (che vengono denominati *multicalcolatori*) e quelli con un singolo spazio di indirizzamento (per i quali riserva il termine *multiprocessore*).

Il primo sistema multiprocessore con connessione a bus e snooping della cache fu il Synapse N+1 nel 1984. La metà degli anni '80 vide un'esplosione nello sviluppo di nuovi protocolli di coerenza, e Archibald e Baer [1986] presentano una buona analisi panoramica al riguardo, completata da riferimenti agli articoli originali. Gli ultimi anni '80 hanno visto l'introduzione di molte architetture commerciali con connessione a bus e snooping della cache, tra cui il sistema 4D/240 di Silicon Graphics, il Multimax di Encore e il Symmetry di Sequent.

Nel tentativo di costruire sistemi multiprocessore di grandi dimensioni sono state esplorate due strade differenti: i multicalcolatori a scambio di messaggi ed i multiprocessori scalabili a memoria condivisa. Anche se vi sono stati molti tentativi di costruire sistemi multiprocessore con connessioni a griglia o ad ipercubo, uno dei primi elaboratori a combinare con successo le varie componenti fu il Cosmic Cube, costruito alla Caltech [Seitz 1985]: esso segnò l'introduzione di significativi miglioramenti nei metodi di instradamento e nella tecnologia di interconnessione, riducendone in maniera significativa il costo, e contribuendo quindi a rendere fattibili i sistemi multicalcolatore. Tra i sistemi commerciali (e relative architetture) vi sono l'iPSC 860 ed il Paragon di Intel e la CM-5 di Thinking Machines. Il mercato per tali sistemi si rivelò molto più piccolo di quanto sperato, così che Intel si ritirò dal mercato (il sistema ASCI Red è l'ultimo prodotto in quest'area) e la Thinking Machines non esiste più. Oggi il settore è occupato principalmente dalle architetture a cluster, come l'RS/6000 SP2 di IBM (figura 9.23).

L'estensione del modello a memoria condivisa con meccanismi scalabili per la coerenza delle cache venne realizzata combinando diverse idee. Le tecniche basate su directory per la coerenza delle cache erano note già prima di quelle basate su meccanismi di snooping. Il primo protocollo per la coerenza delle cache usava infatti le directory e fu implementato nel sistema 3081 di IBM nel 1976. L'idea di distribuire anche le directory oltre alle memorie per ottenere un'implementazione scalabile dei meccanismi di coerenza delle cache era alla base del sistema multiprocessore DASH costruito a Stanford; tale meccanismo viene ora denominato memoria distribuita condivisa (o Distributed Shared Memory, DSM), ed è considerato il progenitore dei

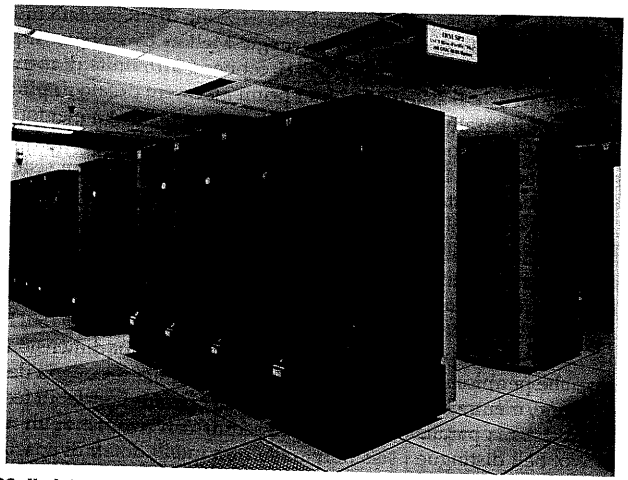


Figura 9.23 Il sistema RS/6000 SP2 di IBM (256 processori). Questo elaboratore a memoria distribuita è composto di schede derivate da calcolatori da tavolo in buona parte senza modifiche, oltre ad un elemento di commutazione proprietario per le interconnessioni. Al contrario dell'SP2, molti sistemi a cluster usano reti locali commutate di tipo commerciale. Fotografia gentilmente concessa dal Lawrence Livermore National Laboratory.

calcolatori di tipo NUMA (figura 9.9). Il sistema KSR-1 di Kendall Square Research (riposi in pace) fu la prima implementazione commerciale di una memoria condivisa scalabile e coerente. Tale sistema estese l'approccio DSM elementare per implementare un concetto denominato COMA (Cache Only Memory Architecture), nel quale la memoria principale diventa una cache (si veda la figura 9.10).

Sui sistemi multiprocessore vi è una gran mole di informazioni: conferenze, articoli su riviste, e persino libri sembrano apparire con velocità maggiore di quella con cui una qualunque persona possa assimilare le idee. Una buona sorgente è la Supercomputing Conference, che si tiene ogni anno a partire dal 1988. Due riviste, il *Journal of Parallel and Distributed Computing* e le *IEEE Transactions on Parallel and Distributed Systems* contengono articoli sui diversi aspetti del calcolo parallelo. Libri di testo sul calcolo parallelo sono stati scritti da Almasi e Gottlieb [1989], Andrews [1991], Culler, Singh e Gupta [1998] e Hwang [1993]. Il libro di Pfister [1995] è uno dei pochi sui sistemi a cluster.

Letture consigliate

Almasi, G.S., e A. Gottlieb [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA.
Un libro di testo sul tema dei calcolatori paralleli.

Amdahl, G.M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.*, Atlantic City, NJ, (Aprile) 483-85
Scritto in risposta alle pretese dell'Illiac IV, questo articolo di 3 pagine descrive la legge di Amdahl e dà le risposte classiche alle argomentazioni tendenti ad abbandonare l'attuale forma di elaborazione.

Andrews, G.R. [1991]. *Concurrent Programming Principles and Practice*, Benjamin/Cummings, Redwood City, CA.
Un testo che fornisce i principi della programmazione parallela.

Archibald, J., e J.-L. Baer [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems* 4:4 (Novembre) 273-98.
Un articolo classico che dà una panoramica sui protocolli di coerenza delle cache nei sistemi a bus condiviso.

Arpaci-Dusseau, A., R. Arpaci-Dusseau, D. Culler, J. Hellerstein e D. Patterson [1997]. "High-performance sorting on networks of workstations," *Proc. ACM SIGMOD/PODS Conference on Management of Data*, Tucson, AZ, Maggio 12-15.
Contiene la descrizione di come venne conseguito il record mondiale di ordinamento su un sistema a cluster, compresa la descrizione dell'architettura della workstation e dell'interfaccia di rete. Il primo aprile 1997 gli stessi autori stabilirono il nuovo record ordinando 8.6 GB in un minuto e impiegando 2,2 secondi per ordinare 100 MB.

Bell, C.G. [1985]. "Multis: A new class of multiprocessor computers," *Science* 228 (Aprile 26) 462-67
Introduce la distinzione tra sistemi multiprocessore basati su microprocessore con spazio di indirizzamento condiviso e non.

Culler, D.E. e J.P. Singh, con A. Gupta [1998]. *Parallel Computer Architecture*, Morgan Kaufmann, San Francisco.
Un nuovo libro di testo sui calcolatori paralleli.

Falk, H. [1976]. "Reaching for the Gigaflop," *IEEE* Spectrum 13:10 (Ottobre) 65-70
Ricostruisce la triste storia dello Illiac IV: costo quattro volte maggiore e prestazioni dieci volte minori rispetto agli obiettivi originari.

Flynn, M.J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (Dicembre) 1901-09.
Articolo classico che introduce la classificazione dei sistemi SISD/SIMD/MISD/MIMD.

Hord, R.M. [1982]. *The Illiac-IV, the First Supercomputer*, Computer Science Press, Rockville, MD
Un'esposizione storica del progetto Illiac IV.

Hwang, K. [1993]. *Advanced Computer Architecture with Parallel Programming*, McGraw-Hill New York.
Un altro libro di testo sui calcolatori paralleli.

9.11 Parole chiave

I termini elencati nel seguito riflettono le idee chiave discusse in questo capitolo. Coloro che non sono sicuri del significato dei termini sono invitati a fare riferimento al Glossario che si trova al fondo del testo.

Accesso in memoria non uniforme (NUMA)
Accesso in memoria uniforme (UMA)
Ampiezza di banda della rete
Broadcasting in scrittura
Cluster
Coerenza della cache tramite snooping
Coerenza della memoria principale
Coerenza delle cache
Directory
Falsa condivisione
Flusso di istruzioni multiplo, flusso di dati multiplo (MIMD)
Flusso di istruzioni singolo, flusso di dati multiplo (SIMD)
Flusso di istruzioni singolo, flusso di dati singolo (SISD)
Invalidazione della scrittura
Lock
Massicciamente parallelo
Memoria condivisa
Memoria distribuita
Memoria distribuita condivisa (Distributed Shared Memory o DSM)
Multicalcolatore
Operazione atomica
Parallelismo dei dati
Procedura di invio dei messaggi
Procedura di ricezione dei messaggi
Processore vettoriale
Programma per l'elaborazione parallela
Protocollo MESI di coerenza delle cache
Rete completamente connessa
Rete crossbar
Rete multistadio
Scambio di messaggi
Sincronizzazione
Sincronizzazione a barriera
Sistema multiprocessore
Sistema multiprocessore simmetrico (SMP)

9.12 Esercizi

9.1 [15] <§9.1> Si scriva un articolo di una pagina in cui si esamini la propria vita alla ricerca di situazioni in cui sia presente la concorrenza e sia realizzata la mutua esclusione. Si considerino ad esempio casi quali un'autostrada che passa da due ad una corsia, le attese in coda nei vari uffici, le modalità per fare delle domande al proprio insegnante, e così via. Si cerchi di scoprire i diversi significati ed i meccanismi utilizzati per realizzare la comunicazione e la sincronizzazione. Vi sono situazioni in cui sarebbe auspicabile l'uso di un algoritmo alternativo in grado di migliorare la latenza o l'ampiezza di banda, o magari di realizzare un sistema più "corretto"?

9.2 [10] <§9.1> Si considerino i seguenti frammenti appartenenti a due diversi programmi che lavorano contemporaneamente su due processori in un sistema SMP. Si assuma che prima dell'avvio del programma sia x che y valgano 0.

Processore 1: `...; x:=x+1; y:=x+y; ...`

Processore 2: `...; y:=x+1; ...`

Quali sono i possibili valori finali di x e y, assumendo che il codice sia implementato utilizzando un'architettura load-store? Per ogni possibile risultato, si spieghi come x e y possono aver raggiunto tale valore. Suggerimento: si esaminino tutte le possibili sequenze di esecuzione delle istruzioni in linguaggio assembler.

9.3 [10] <§§9.1-9.3> Si immagini che tutti i dipendenti di una grande azienda abbiano dimenticato chi è il padrone dell'azienda e ricordino soltanto chi sia il loro capo diretto. I responsabili stanno prendendo in considerazione la possibilità di dare una delle seguenti direttive:

- "Oggi ciascun dipendente chieda al proprio capo chi è il suo capo; domani chieda a quella persona chi è il suo capo, e così via, sino a che tutti sappiano chi è il padrone dell'azienda."
- "Ciascuno scriva su un pezzo di carta il nome del proprio capo. Si guardi quale nome ha scritto sul suo pezzo di carta la persona il cui nome compare sul proprio pezzo, e domani (prima di cominciare il lavoro) si scriva tale nome sul proprio pezzo di carta. Si ripeta il procedimento sino a che ciascuno non ha scoperto chi è il padrone dell'azienda."

Si scriva un paragrafo in cui si descrivono le differenze tra le due direttive e le relative conseguenze. Si illustri la relazione tra le due alternative descritte e ciò che se ne deduce in termini di concorrenza, sincronizzazione e comunicazione tra processi.

9.4 [5] <§§9.1-9.3> {Es. 9.3} Si analizzino le prestazioni dei due algoritmi proposti sopra. Si considerino aziende di 100, 1000 e 10000 dipendenti. È possibile individuare un modo per migliorare l'uno o l'altro dei due algoritmi o per ottenere lo stesso risultato in maniera più veloce?

9.5 [5] <§9.3> Si conti il numero di transazioni sul bus nel corso della seguente sequenza di attività che coinvolge dei dati condivisi. Si assuma che ambedue i processori utilizzino cache di tipo write-back, un meccanismo di coerenza della cache di tipo write-update e una dimensione del blocco pari ad una parola. Si assuma che tutte le parole nelle due cache siano pulite.

Passo	Processore	Attività sulla memoria	Indirizzo in memoria
1	processore 1	scrittura	100
2	processore 2	scrittura	104
3	processore 1	lettura	100
4	processore 2	lettura	104

9.6 [10] <§9.3> La falsa condivisione può portare a inutile traffico sul bus e a ritardi. Si seguano le indicazioni dell'Esercizio 9.5, ma si cambi la dimensione del blocco portandola a 4 parole.

9.7 [15] <§9.4> La griglia tridimensionale è un'altra possibile topologia di rete. Si disegni la topologia per una rete di 64 nodi sulla falsa riga della figura 9.13. Qual è l'ampiezza di banda di bisezione per questa topologia?

9.8 [1 settimana] <§§9.2-9.6> I processori paralleli vengono spesso commercializzati riportando dati relativi a programmi le cui prestazioni crescono linearmente con il numero di processori. Si portino i programmi scritti per un processore parallelo su di un altro, e si misurino le prestazioni assolute ed il modo in cui queste cambiano variando il numero di processori. Quali modifiche si devono introdurre per migliorare su ciascun elaboratore le prestazioni dei programmi considerati? Quali sono le prestazioni deducibili osservando ciascun programma?

9.9 [1 settimana] <§§9.2-9.6> Anziché andare alla ricerca di programmi campione corretti, si inventino dei programmi che facciano sembrare terribile un certo processore parallelo rispetto agli altri, nonché dei programmi che facciano apparire un processore sempre migliore degli altri. Quali sono le caratteristiche chiave per ciascun programma e ciascun elaboratore in termini di prestazioni?

9.10 [1 settimana] <§§9.2-9.6> I processori paralleli sono normalmente caratterizzati da prestazioni crescenti al crescere del numero dei processori, fino ad un valore ideale pari ad un aumento di velocità di n volte con n processori. L'obiettivo di questo esercizio è quello di creare un programma campione particolare, in grado di produrre delle prestazioni peggiori quando si aggiungono dei processori. Per esempio, un singolo processore esegue il programma nel modo più veloce, due processori sono più lenti, quattro processori sono più lenti di due, e così via. Quali sono le caratteristiche chiave in termini di prestazioni che per ciascuna organizzazione portano ad un aumento di velocità lineare inverso?

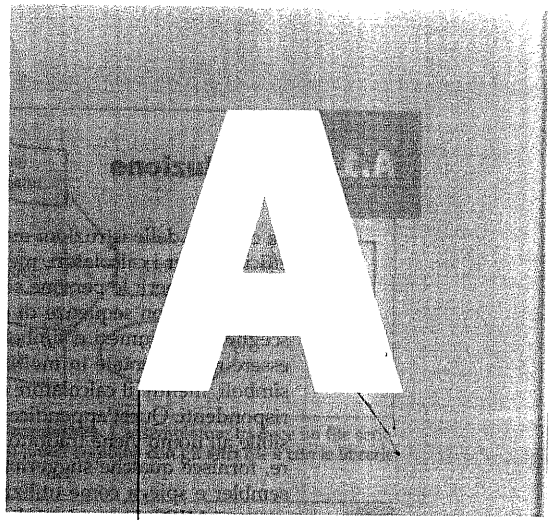
9.11 [1 settimana] <§§9.2-9.6> Un gruppo di workstation connesse in rete può essere considerato un processore parallelo, anche se è caratterizzato da una bassa velocità di comunicazione rispetto a quella di elaborazione. Si trasportino i programmi campione sviluppati per i processori paralleli sulla rete di workstation utilizzando per la comunicazione le chiamate a procedure remote. Quanto sono efficienti i programmi prescelti nell'aumentare le loro prestazioni al crescere del numero di processori sulla rete, rispetto al comportamento sul processore parallelo? Quali sono le differenze pratiche tra delle workstation connesse in rete ed un processore parallelo commerciale?

9.12 [1 settimana] <§§9.2-9.6> I miglioramenti nelle prestazioni si dicono *superlineari* quando un programma che lavora su n processori è più di n volte veloce

rispetto all'equivalente programma su un sistema uniprocessore. Una spiegazione per l'accelerazione superlineare consiste nell'osservare che il tempo speso per servire le interruzioni o il cambio di contesto si riduce quando si hanno molti processori, dal momento che solo uno di essi deve servire la richiesta, mentre gli altri possono essere condivisi dagli utenti. Si misuri il tempo speso durante il normale funzionamento per gestire le interruzioni o il cambio di contesto su un sistema uniprocessore e su un processore parallelo. Il carico di lavoro può essere costituito da una combinazione di attività indipendenti per un ambiente multiprogramma oppure da un unico lavoro di grandi dimensioni. La spiegazione data è accettabile?

9.13 [15] <§9.9> Si costruisca uno scenario in cui un'architettura veramente rivoluzionaria (si scelga quella preferita) giochi un ruolo significativo. Il termine *significativo* indica una quota del 10% dei calcolatori venduti, del 10% degli utenti, del 10% del denaro speso per i calcolatori, o del 10% di qualche altro parametro di merito.

9.14 [20] <§§9.1-9.11> Questo capitolo ha introdotto molti nuovi termini legati all'argomento dei sistemi multiprocessore. Alcuni degli esercizi di questo capitolo (ad esempio gli Esercizi 9.1 e 9.3) sono basati sull'analogia tra persone e processori e tra gruppi di persone e sistemi multiprocessore. Si scriva un articolo di una pagina che esplori in maggior dettaglio questa analogia: ad esempio si indaghi sull'esistenza di gruppi di persone che utilizzano tecniche quali lo scambio di messaggi o le memorie condivise. È possibile individuare delle analogie con i protocolli di coerenza delle cache, le topologie di rete, o i cluster? Si cerchi di includere almeno un vocabolo preso da ciascuna parte del testo.



A P P E N D I C E

Assemblatori, linker ed il simulatore SPIM

James R. Larius
Computer Science Department
University of Wisconsin-Madison

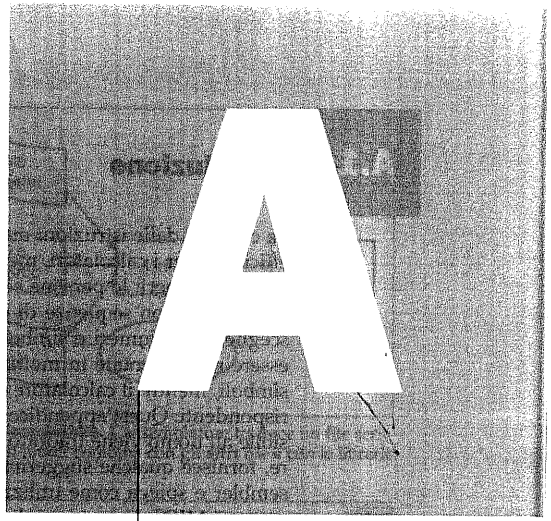
Il timore di gravi danni non può da solo giustificare la soppressione della libertà di parola e di riunione

Louis Brandeis
Whitney v. California, 1927

rispetto all'equivalente programma su un sistema uniprocessore. Una spiegazione per l'accelerazione superlineare consiste nell'osservare che il tempo speso per servire le interruzioni o il cambio di contesto si riduce quando si hanno molti processori, dal momento che solo uno di essi deve servire la richiesta, mentre gli altri possono essere condivisi dagli utenti. Si misuri il tempo speso durante il normale funzionamento per gestire le interruzioni o il cambio di contesto su un sistema uniprocessore e su un processore parallelo. Il carico di lavoro può essere costituito da una combinazione di attività indipendenti per un ambiente multiprogramma oppure da un unico lavoro di grandi dimensioni. La spiegazione data è accettabile?

9.13 [15] <§9.9> Si costruisca uno scenario in cui un'architettura veramente rivoluzionaria (si scelga quella preferita) giochi un ruolo significativo. Il termine *significativo* indica una quota del 10% dei calcolatori venduti, del 10% degli utenti, del 10% del denaro speso per i calcolatori, o del 10% di qualche altro parametro di merito.

9.14 [20] <§§9.1-9.11> Questo capitolo ha introdotto molti nuovi termini legati all'argomento dei sistemi multiprocessore. Alcuni degli esercizi di questo capitolo (ad esempio gli Esercizi 9.1 e 9.3) sono basati sull'analogia tra persone e processori e tra gruppi di persone e sistemi multiprocessore. Si scriva un articolo di una pagina che esplori in maggior dettaglio questa analogia: ad esempio si indaghi sull'esistenza di gruppi di persone che utilizzano tecniche quali lo scambio di messaggi o le memorie condivise. È possibile individuare delle analogie con i protocolli di coerenza delle cache, le topologie di rete, o i cluster? Si cerchi di includere almeno un vocabolo preso da ciascuna parte del testo.



A P P E N D I C E

Assemblatori, linker ed il simulatore SPIM

James R. Larius
Computer Science Department
University of Wisconsin-Madison

Il timore di gravi danni non può da solo giustificare la soppressione della libertà di parola e di riunione

Louis Brandeis
Whitney v. California, 1927

A.1 Introduzione

La codifica delle istruzioni mediante numeri binari è un procedimento naturale ed efficiente per i calcolatori, mentre per gli uomini è molto difficile capire e manipolare tali numeri: le persone sono in grado di manipolare simboli (le parole) molto meglio che non sequenze di cifre. Il Capitolo 3 ha mostrato che non è necessario scegliere fra numeri e simboli, in quanto le istruzioni di un calcolatore possono essere rappresentate in molti modi diversi: gli uomini possono scrivere e leggere simboli, mentre il calcolatore esegue il programma a partire dai numeri binari corrispondenti. Quest'appendice descrive il processo con cui un programma comprensibile all'uomo viene tradotto in un formato che il calcolatore è in grado di eseguire, fornisce qualche suggerimento su come scrivere programmi in linguaggio assembler e spiega come utilizzare SPIM, un simulatore in grado di eseguire i programmi MIPS. Le versioni di SPIM per DOS, Windows e Unix sono disponibili all'indirizzo *www.mkp.com/cod2e.htm*.

Il *linguaggio assembler* è la rappresentazione simbolica della codifica binaria usata dal calcolatore, detta *linguaggio macchina*; l'assembler è più leggibile del linguaggio macchina perché utilizza simboli invece di bit. I simboli del linguaggio assembler permettono di associare dei nomi alle configurazioni di bit più frequenti, come i codici operativi o i riferimenti ai registri, in modo tale che le persone abbiano più facilità a leggerle e ricordarle. In più l'assembler permette ai programmatori di utilizzare *etichette* (label) per identificare particolari parole di memoria che contengono dati o istruzioni.

Uno strumento chiamato *assemblatore* traduce il linguaggio assembler in istruzioni binarie; gli assemblatori forniscono una rappresentazione più vicina all'utente rispetto a zeri ed uni e semplificano la scrittura e la lettura dei programmi. I nomi simbolici di operazioni e locazioni di memoria sono un aspetto di questa rappresentazione; un altro aspetto è costituito dalle funzioni che il programmatore ha a disposizione per rendere più chiaro il programma. Le *macro* ad esempio, discusse nel Paragrafo A.2, consentono al programmatore di estendere il linguaggio assembler definendo nuove operazioni.

Un assemblatore legge un singolo *file sorgente* in linguaggio assembler e produce un *file oggetto* contenente le istruzioni macchina e le informazioni di gestione che gli consentono di combinare più file oggetto in un unico programma. La figura A.1 mostra il processo di costruzione di un programma: la maggior parte dei programmi consistono di diversi file, detti anche *moduli*, che sono scritti, compilati ed assemblati in modo indipendente; un programma può anche utilizzare delle procedure già scritte facenti parte di una *libreria di programmi*. Un modulo generalmente contiene *riferimenti* a dati e procedure definiti all'interno di una libreria o di altri file oggetto. Il codice di un modulo non può essere eseguito se contiene dei *riferimenti non risolti* ad etichette presenti nelle librerie o in altri file oggetto; un altro strumento, chiamato *linker*, combina l'insieme dei file oggetto e delle librerie in un *file eseguibile*, cioè in un file che il calcolatore è in grado di eseguire.

Per capire i vantaggi del linguaggio assembler si consideri la serie di figure riportata nel seguito, ciascuna delle quali contiene una breve procedura che calcola e stampa la somma dei quadrati degli interi compresi fra 0 e 100. La figura A.2 mostra il linguaggio macchina che un calcolatore MIPS è in grado di eseguire; con

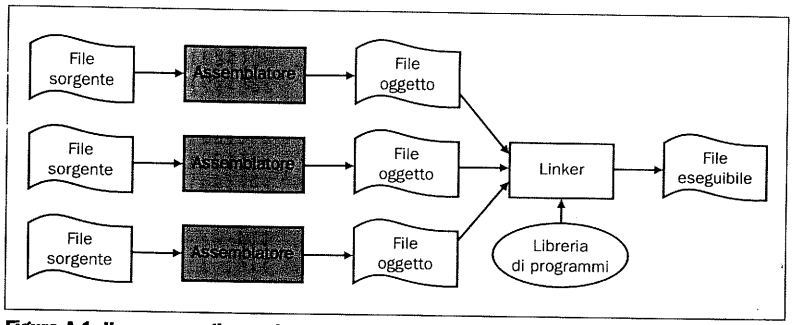


Figura A.1 Il processo di creazione di un file eseguibile. L'assemblatore traduce un file scritto in linguaggio assembler in un file oggetto, che il linker combina con gli altri file e con le librerie per generare il file eseguibile.

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

Figura A.2 Il codice di una procedura che calcola e stampa la somma dei quadrati degli interi compresi fra 0 e 100, scritto in linguaggio macchina MIPS.

uno sforzo considerevole è possibile usare le tabelle di formato dei codici operativi e delle istruzioni riportate nel Capitoli 3 e 4 per tradurre le istruzioni in una rappresentazione simbolica del programma simile a quella mostrata in figura A.3. In questo formato la procedura è molto più leggibile, dal momento che le operazioni e gli operandi sono rappresentati in forma simbolica invece che come sequenze di bit. Questo linguaggio assembler è però ancora difficile da seguire perché le locazioni di memoria sono identificate dal loro indirizzo e non da un'etichetta simbolica.

```
addiu   $29,  $29, -32
sw      $31,  20($29)
sw      $4,   32($29)
sw      $5,   36($29)
sw      $0,   24($29)
sw      $0,   28($29)
lw      $14,  28($29)
lw      $24,  24($29)
multu   $14,  $14
addiu   $8,   $14, 1
slti    $1,   $8, 101
sw      $8,   28($29)
mflo    $15
addu    $25,  $24, 15
bne     $1,   $0, -9
sw      $25,  24($29)
lui     $4,   4096
lw      $5,   24($29)
jal     1048  812
addiu   $4,   $4, 1072
lw      $31,  20($29)
addiu   $29,  $29, 32
jr      $31
move    $2,   $0
```

Figura A.3 La stessa procedura scritta in linguaggio assembler. Questo codice non utilizza né etichette per i registri e per le locazioni di memoria, né commenti.

La figura A.4 mostra il linguaggio assembler che utilizza le etichette mnemoniche per gli indirizzi di memoria; la maggior parte dei programmatori preferisce leggere e scrivere programmi in questa forma. I nomi che iniziano con un punto, come `.data` o `.globl`, sono *direttive dell'assemblatore;* esse specificano all'assemblatore come tradurre un programma ma non generano istruzioni macchina. I nomi seguiti dai due punti, come `str` o `main`, sono etichette che associano un nome alla locazione di memoria successiva. Questo programma ha lo stesso grado di leggibilità di gran parte dei programmi assembler (a parte l'evidente mancanza di commenti), ma è ancora difficile da seguire perché usa molte piccole operazioni per eseguire compiti semplici, e perché l'assenza in assembler dei costrutti per il controllo del flusso fornisce poche indicazioni sulle operazioni eseguite dal programma.

Al contrario la procedura C di figura A.5 è più breve e più chiara, dato che le variabili hanno nomi mnemonici ed il ciclo è esplicito invece di essere realizzato tramite salti condizionati. Il lettore che non abbia familiarità con il linguaggio C è invitato a consultare l'estensione Web II all'indirizzo *www.mkp.com/cod2e.htm*). In realtà l'unica procedura che gli autori del libro hanno scritto è quest'ultima, e tutti gli altri formati sono stati prodotti dal compilatore C e dall'assemblatore.

In generale il linguaggio assembler svolge due ruoli (vedere figura A.6). Per prima cosa è il linguaggio di uscita dei compilatori: un *compilatore* traduce un programma scritto in un *linguaggio ad alto livello* (come il C o il Pascal) nel programma


```
        .text
        .align    2
        .globl    main
main:
        subu      $sp,    $sp, 32
        sw        $ra,    20($sp)
        sd        $a0,    32($sp)
        sw        $0,     24($sp)
        sw        $0,     28($sp)
ciclo:
        lw        $t6,    28($sp)
        mul       $t7,    $t6, $t6
        lw        $t8,    24($sp)
        addu      $t9,    $t8, $t7
        sw        $t9,    24($sp)
        addu      $t0,    $t6, 1
        sw        $t0,    28($sp)
        ble       $t0,    100, ciclo
        la        $a0,    str
        lw        $a1,    24($sp)
        jal       printf
        move      $v0,    $0
        lw        $ra,    20($sp)
        addu      $sp,    $sp, 32
        j         $ra

        .data
        .align 0
str:
        .asciiz   "La somma da 0 a 100 vale %d\n"
```

Figura A.4 La stessa procedura scritta in linguaggio assembler con le etichette, ma senza commenti. I comandi che iniziano con il punto sono direttive dell'assemblatore (vedere il Paragrafo A.10). `.text` indica che le linee seguenti contengono istruzioni, mentre `.data` indica che contengono dei dati. `.align n` indica che gli elementi riportati nelle linee successive devono essere allineati ogni 2^n byte. Quindi `.align 2` indica che l'elemento successivo è allineato ad una parola di memoria. `.globl main` dichiara che `main` è un simbolo globale che deve essere visibile anche al codice memorizzato in altri file. Infine `.asciiz` copia in memoria una stringa terminata dal carattere Null.

```
# include <stdio.h>
int main (int argc, char *argv[])
{
        int i;
        int somma = 0;

        for (i = 0; i <= 100; i = i + 1) somma = somma + i * i;
        printf ("La somma da 0 a 100 vale %d\n", somma);
}
```

Figura A.5 La procedura scritta nel linguaggio di programmazione C.

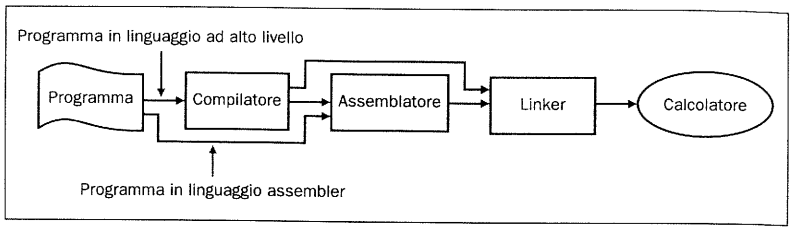


Figura A.6 Il linguaggio assembler può essere scritto direttamente dal programmatore oppure ottenuto come uscita di un compilatore.

equivalente scritto in linguaggio macchina o in assembler. Il linguaggio ad alto livello è chiamato *linguaggio sorgente*, e l'uscita del compilatore *linguaggio destinazione*.

D'altro lato il linguaggio assembler è un linguaggio in cui programmare. Una volta questo era il ruolo dominante; oggi invece, grazie alla disponibilità di memorie più estese e di compilatori migliori, la maggior parte dei programmatori utilizza i linguaggi ad alto livello e scende molto raramente, ammesso che lo faccia, al livello delle istruzioni effettivamente eseguite dal calcolatore. Nonostante ciò scrivere programmi in assembler è ancora importante nel caso in cui velocità o occupazione di memoria siano critiche, o quando si vogliano sfruttare caratteristiche dell'hardware non accessibili da un linguaggio di programmazione ad alto livello.

Benché quest'appendice sia focalizzata sul linguaggio assembler MIPS, i linguaggi assembler della maggior parte dei calcolatori si assomigliano molto. Le istruzioni ed i modi di indirizzamento aggiuntivi dei calcolatori CISC come il VAX (vedere l'estensione Web III all'indirizzo *www.mkp.com/cod2e.htm*) possono rendere i programmi più brevi ma non cambiano il processo con cui questi vengono assemblati, né aggiungono loro le potenzialità dei linguaggi ad alto livello (come il controllo del tipo delle variabili o la strutturazione dei cicli).

Quando usare il linguaggio assembler

La necessità di programmare in assembler invece che in un linguaggio ad alto livello si verifica quando la velocità di esecuzione o l'occupazione di memoria di un programma sono elementi critici. Si consideri ad esempio un calcolatore che controlla un dispositivo meccanico, ad esempio il freno di un'automobile; i calcolatori inseriti all'interno di altri strumenti, come l'automobile, sono detti *calcolatori embedded*. Questi calcolatori devono rispondere in modo rapido ed affidabile agli eventi esterni, e dato che i compilatori introducono dell'incertezza sul costo delle operazioni in termini di tempo, i programmatori hanno difficoltà ad assicurare che un programma scritto in un linguaggio ad alto livello risponda in un intervallo di tempo definito (ad esempio 1 millisecondo dopo che un sensore ha rilevato che una ruota sta slittando). Un programmatore assembler, invece, ha un forte controllo sull'esecuzione delle istruzioni. In più nel caso di applicazioni embedded ridurre le dimensioni di un programma in modo che diventi memorizzabile su chip di memoria più piccoli riduce il costo del sistema.

Un approccio ibrido, nel quale gran parte del programma è scritto in un linguaggio ad alto livello ed alcune sezioni critiche sono in assembler, sfrutta il meglio di ciascun linguaggio. Solitamente i programmi utilizzano gran parte del tempo di esecuzione per una piccola porzione del codice sorgente; questa osservazione è

proprio il principio che sta dietro l'utilizzo delle memorie cache (vedere il Paragrafo 7.2).

Mediante opportuni strumenti di analisi si possono misurare i tempi di esecuzione delle varie parti di un programma e capire quali sono le porzioni critiche per quanto riguarda il tempo di esecuzione. In molti casi è possibile rendere più veloci queste porzioni di codice migliorando le strutture dati o gli algoritmi. Talvolta però per ottenere miglioramenti significativi delle prestazioni occorre ri-codificare in assembler la porzione critica.

Questo miglioramento non indica necessariamente che il compilatore del linguaggio ad alto livello ha sbagliato: in generale i compilatori sono più efficaci dei programmatori nel produrre codice macchina di elevata qualità per l'intero programma. I programmatori però sono in grado di capire più a fondo gli algoritmi ed il comportamento di un programma rispetto ad un compilatore e con uno sforzo di impegno e di ingegnosità possono migliorare piccole porzioni del programma. In particolare nello scrivere il loro codice i programmatori spesso considerano simultaneamente più procedure, mentre i compilatori in genere trattano ciascuna procedura separatamente dalle altre e seguono delle rigide convenzioni per l'utilizzo dei registri di interfaccia fra una procedura e l'altra. Mantenendo nei registri le variabili usate più frequentemente, anche fra una procedura e l'altra, i programmatori possono far sì che un programma venga eseguito più velocemente.

Un altro grande vantaggio del linguaggio assembler è la possibilità di utilizzare istruzioni specializzate, ad esempio nella copia di stringhe o per il confronto con una data configurazione. Nella maggior parte dei casi i compilatori non sono in grado di determinare quando un ciclo può essere sostituito da una sola istruzione, cosa che invece il programmatore riesce a fare facilmente.

Nel futuro la maggiore abilità del programmatore rispetto al compilatore diventerà probabilmente sempre più difficile da mantenere, in quanto le tecniche su cui si basano i compilatori diventano sempre più sofisticate e le pipeline sempre più complesse (vedere il Capitolo 6).

Un'ultima ragione per usare l'assembler si ha nel caso in cui per un dato calcolatore non sia disponibile nessun linguaggio ad alto livello; molti calcolatori, perché vecchi o specializzati, non possiedono un compilatore, e quindi l'unica possibilità per il programmatore è quella di usare il linguaggio assembler.

Gli svantaggi del linguaggio assembler

Il linguaggio assembler ha molti inconvenienti, che fanno sì che il suo uso sia molto limitato. Forse il principale svantaggio è che i programmi scritti in assembler sono per definizione specifici per un dato calcolatore e devono essere totalmente riscritti per essere eseguiti su una macchina diversa. La rapida evoluzione dei calcolatori descritta nel Capitolo 1 fa sì che le architetture diventino obsolete in breve tempo. Un programma assembler rimane strettamente legato all'architettura originale anche dopo che il calcolatore è superato da macchine nuove, più veloci e con un miglior rapporto prezzo-prestazioni.

Un altro svantaggio consiste nel fatto che i programmi assembler sono più lunghi dei programmi corrispondenti scritti in un linguaggio ad alto livello: ad esempio il programma C di figura A.5 è lungo 11 linee di codice, mentre il programma assembler di figura A.4 è composto da 31 linee. Nel caso di programmi più complessi il rapporto fra codice assembler e codice in linguaggio ad alto livello (detto *fattore di espansione*) può essere anche molto maggiore del fattore tre di questo

esempio. Sfortunatamente studi empirici hanno dimostrato che i programmatori grosso modo scrivono lo stesso numero di linee di codice al giorno qualunque sia il linguaggio, assembler o ad alto livello; ciò significa che i programmatori sono circa x volte più produttivi se scrivono con un linguaggio ad alto livello, dove x è il fattore di espansione del linguaggio assembler.

A peggiorare le cose contribuisce il fatto che programmi più lunghi sono più difficili da leggere e contengono un maggior numero di errori. Il linguaggio assembler aggrava ancora di più il problema perché non supporta la programmazione strutturata: costrutti usati frequentemente come le istruzioni *if-then* o i cicli devono essere costruiti tramite salti. I programmi risultano difficili da capire perché il lettore deve ricostruire mentalmente tutti i costrutti ad alto livello a partire dai frammenti di codice, e ciascuna istanza del costrutto può essere leggermente diversa. Si guardi ad esempio la figura A.4 e si risponda alle seguenti domande: quale tipo di ciclo è stato usato? Quali sono i limiti inferiore e superiore?

Approfondimento: I compilatori possono produrre direttamente codice macchina senza affidarsi all'assemblatore; in generale questi compilatori sono più veloci di quelli che invocano l'assemblatore durante il processo di compilazione. Un compilatore che genera linguaggio macchina deve però occuparsi di molti aspetti che normalmente gestisce l'assemblatore, come risolvere gli indirizzi e codificare le istruzioni in forma di numeri binari. Il compromesso è fra la velocità di compilazione e la semplicità del compilatore.

Approfondimento: Nonostante queste considerazioni, molte applicazioni embedded sono scritte in linguaggi ad alto livello. Molte di queste applicazioni sono programmi lunghi e complessi che devono essere estremamente affidabili. I programmi assembler sono più lunghi e più difficili da leggere e da scrivere dei programmi ad alto livello; questo aumenta di molto il costo di tali programmi e rende estremamente difficile verificarne la correttezza. Queste considerazioni hanno condotto il dipartimento della difesa degli Stati Uniti, che finanzia lo sviluppo e il mantenimento di molti sistemi embedded di grande complessità, a sviluppare Ada (un altro linguaggio ad alto livello) per costruire sistemi di questo tipo.

A.2 Gli assemblatori

L'assemblatore traduce un file scritto in codice assembler in un file composto da istruzioni e dati in formato binario; il processo di traduzione è costituito da due passi fondamentali. Il primo passo consiste nel determinare gli indirizzi delle locazioni di memoria etichettate, in modo tale che le relazioni fra nomi simbolici ed indirizzi siano note quando viene fatta la traduzione delle istruzioni. Il secondo passo traduce tutte le istruzioni assembler combinando gli equivalenti numerici di codice operativo, identificatori di registri ed etichette in un'istruzione macchina valida. Come mostrato in figura A.1, l'assemblatore produce un file di uscita, chiamato *file oggetto*, che contiene le istruzioni macchina, i dati e le informazioni di supporto.

In generale un file oggetto non può essere eseguito direttamente perché fa riferimento a procedure o dati contenuti in altri file. Un'etichetta è detta *esterna* (o anche *globale*) se è possibile fare riferimento all'oggetto etichettato anche a partire da altri file oltre a quello in cui è definita; un'etichetta è detta invece *locale* se l'oggetto etichettato può essere usato solo all'interno del file in cui è definito. Nel caso della

maggior parte degli assemblatori, in assenza di indicazioni le etichette sono locali e le etichette globali devono essere dichiarate esplicitamente. Dal momento che è possibile fare riferimento a procedure e variabili globali a partire da più file di un programma, queste richiedono delle etichette esterne. Le etichette locali celano il proprio nome a moduli presenti in altri file e non devono essere visibili ad essi; le funzioni del C di tipo static, che possono essere chiamate soltanto da funzioni presenti nello stesso file, sono un esempio di etichette locali. Inoltre i nomi generati dal compilatore (ad esempio il nome associato ad un'istruzione di inizio ciclo) sono locali, in modo tale che il compilatore non debba creare nomi diversi per ciascun file.

Etichette locali e globali

Esempio Si consideri il programma di figura A.4. La procedura ha un'etichetta esterna (globale) `main`; inoltre contiene due etichette locali, `ciclo` e `str`, che sono visibili soltanto all'interno di questo file assembler. Infine la procedura contiene anche il riferimento non risolto ad un'etichetta esterna `printf`, che è la procedura di libreria per la stampa di un valore. A quale delle etichette di figura A.4 può essere fatto riferimento a partire da un altro file?

Soluzione Solo le etichette globali sono visibili al di fuori del file, e quindi l'unica etichetta a cui può essere fatto riferimento a partire da un altro file è `main`.

Dal momento che l'assemblatore elabora individualmente e separatamente ciascuno dei file che compongono un programma, conosce soltanto gli indirizzi delle etichette locali. L'assemblatore dipende da un altro strumento, il linker, per riunire un insieme di file oggetto e di librerie in un file eseguibile, risolvendo i riferimenti alle etichette esterne. L'assemblatore coopera con il linker fornendogli l'elenco delle etichette e dei riferimenti non risolti.

Persino le etichette locali, tuttavia, rappresentano un'interessante sfida per l'assemblatore: contrariamente a quanto avviene nella maggior parte dei linguaggi di programmazione ad alto livello, in assembler le etichette possono essere utilizzate prima di essere definite: nell'esempio di figura A.4 l'etichetta `str` è usata dall'istruzione `la` prima di essere definita. Consentire i *riferimenti in avanti*, come in questo caso, forza l'assemblatore ad eseguire la conversione in due passi: prima l'assemblatore trova tutte le etichette, e poi traduce le istruzioni. Nell'esempio, quando l'assemblatore incontra l'istruzione `la` non sa ancora dove si trova la parola di memoria etichettata come `str`, e neppure se l'etichetta `str` si riferisce ad un'istruzione o a un dato.

Durante il primo passo l'assemblatore legge ciascuna riga del file assembler e la suddivide nelle sue componenti; questi oggetti, detti *elementi lessicali*, sono parole, numeri, e segni di punteggiatura. Ad esempio la linea di codice

```
ble   $t0, 100, ciclo
```

contiene sei elementi lessicali: il codice operativo `ble`, l'identificatore di registro `$t0`, una virgola, il numero `100`, una virgola, ed il simbolo `ciclo`.

Se una linea inizia con un'etichetta, l'assemblatore registra nella sua *tabella dei simboli* il nome dell'etichetta e l'indirizzo della parola di memoria relativa all'istru-

zione. L'assemblatore quindi calcola il numero di parole di memoria occupate dall'istruzione riportata nella linea di codice: tenendo traccia della dimensione delle istruzioni l'assemblatore può determinare la posizione dell'istruzione successiva. Per calcolare la dimensione di un'istruzione di lunghezza variabile, come nel caso del VAX, l'assemblatore deve esaminare in dettaglio ciascuna istruzione; al contrario se le istruzioni hanno lunghezza fissa, come nel caso del MIPS, basta un esame più superficiale. L'assemblatore esegue un processo analogo per calcolare lo spazio richiesto dalla porzione relativa ai dati. Quando viene raggiunta la fine del file assembler la tabella dei simboli contiene l'indirizzo di ciascuna etichetta definita nel file.

L'assemblatore utilizza le informazioni contenute nella tabella dei simboli durante la seconda passata sul file assembler, quando viene prodotto l'effettivo codice macchina. Esso esamina nuovamente tutte le linee di codice, e se la linea contiene un'istruzione combina la rappresentazione binaria del codice operativo e degli operandi (identificatori di registro o indirizzi di memoria) in un'istruzione valida. Il processo è simile a quello usato nel Paragrafo 3.4. Le istruzioni e le parole di dato che contengono riferimenti a simboli esterni definiti in un altro file non possono essere completamente assemblati (non sono risolti) dal momento che l'indirizzo del simbolo non si trova nella tabella dei simboli. L'assemblatore non si preoccupa dei riferimenti non risolti, dato che le etichette corrispondenti sono verosimilmente definite in un altro file.

Quadro d'insieme

Il linguaggio assembler è un linguaggio di programmazione. La principale differenza rispetto ai linguaggi ad alto livello, come BASIC, C o Java, sta nel fatto che consente di utilizzare solo un numero limitato di tipi di dato e di controlli sul flusso delle istruzioni. I programmi scritti in assembler non specificano il tipo del valore contenuto in una variabile, ma il programmatore deve fare in modo di applicare le operazioni appropriate (intere o in virgola mobile) a ciascun valore. In più in assembler tutte le operazioni di controllo del flusso devono essere realizzate tramite dei *goto*. Questi due fattori rendono la programmazione assembler di qualunque calcolatore, il MIPS o l'80x86, più difficile e maggiormente soggetta ad errori rispetto alla scrittura di programmi nei linguaggi ad alto livello.

Approfondimento: Se è importante che l'assemblatore sia veloce, questo procedimento in due passi può essere realizzato con una sola passata sul file assembler, usando una tecnica detta *backpatching*. Durante questa passata l'assemblatore costruisce una rappresentazione binaria (a volte incompleta) di ciascuna istruzione. Se l'istruzione fa riferimento ad un'etichetta che non è ancora stata definita, l'assemblatore memorizza etichetta ed istruzione in una tabella; quando l'etichetta viene definita, l'assemblatore consulta la tabella per trovare tutte le istruzioni che contengono un riferimento in avanti a questa etichetta. L'assemblatore quindi torna indietro e corregge la rappresentazione binaria inserendo l'indirizzo dell'etichetta. Questa tecnica rende più veloce il processo di traduzione, in quanto l'assemblatore legge il file di ingresso una sola volta. È tuttavia necessario che l'assemblatore mantenga in memoria l'intera rappresentazione binaria del programma, ponendo dei limiti alle dimensioni dei programmi che possono essere assemblati.

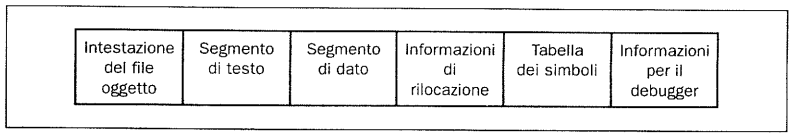


Figura A.7 Il file oggetto. Un assemblatore Unix produce un file oggetto costituito da sei parti distinte.

Il formato del file oggetto

Gli assemblatori producono i file oggetto. Nel sistema operativo Unix un file oggetto contiene sei parti distinte (vedere figura A.7):

- L'*intestazione del file oggetto* descrive le dimensioni e la posizione delle altre parti del file.
- Il *segmento di testo* contiene il codice in linguaggio macchina delle procedure del file sorgente; tali procedure possono essere non eseguibili per via di riferimenti non risolti.
- Il *segmento di dato* contiene la rappresentazione binaria dei dati del file sorgente; anche i dati possono essere incompleti per via di riferimenti non risolti ad etichette definite in altri file.
- Le *informazioni di rilocazione* identificano le parole di istruzione o di dato che dipendono da indirizzi assoluti; questi riferimenti devono cambiare se porzioni del programma sono spostate da una posizione di memoria ad un'altra.
- La *tabella dei simboli* associa gli indirizzi alle etichette esterne ed elenca i riferimenti non risolti.
- Le *informazioni per il debugger* contengono una descrizione concisa del modo in cui è stato compilato il programma, in modo tale che il debugger possa trovare a quale indirizzo corrispondono le linee del codice sorgente e possa stampare le strutture dati in forma leggibile.

L'assemblatore produce un file oggetto che contiene la rappresentazione binaria del programma e dei dati, e alcune informazioni aggiuntive che aiutano a collegare insieme le varie parti del programma. Queste informazioni di rilocazione sono necessarie perché l'assemblatore non conosce a priori la porzione di memoria che una procedura o un blocco di dati occuperà dopo che sarà stato collegato al resto del programma. Le procedure ed i dati di un file sono memorizzati in porzioni contigue di memoria, ma l'assemblatore non sa dove queste porzioni saranno collocate. L'assemblatore inoltre fornisce al linker alcuni elementi della tabella dei simboli; in particolare deve memorizzare quali simboli esterni sono definiti e quali riferimenti non risolti compaiono in un file.

Approfondimento: Per comodità gli assemblatori assumono che ciascun file parta dallo stesso indirizzo (ad esempio la locazione 0) aspettandosi che sia il linker a *rilocare* il codice e i dati al momento di assegnare le locazioni di memoria. L'assemblatore produce le *informazioni di rilocazione*, che contengono un elemento per ciascuna parola di istruzione o di dato del file che fa riferimento ad un indirizzo assoluto. Nel caso del MIPS solo le chiamate a procedura e le istruzioni di trasferimento da e verso la memoria possono fare riferimento ad indirizzi assoluti. Le istruzioni che usano l'indirizzamento relativo al PC non necessitano di rilocazione.

Funzionalità aggiuntive

Gli assemblatori hanno una varietà di funzionalità aggiuntive che aiutano a scrivere programmi in modo più veloce ed agevole, ma che non modificano in modo sostanziale il linguaggio assembler. Ad esempio le *direttive per la codifica dei dati* consentono al programmatore di descrivere i dati in modo più conciso e naturale rispetto alla rappresentazione binaria.

Nella figura A.4, la direttiva

```
.asciiz "La somma da 0 a 100 vale %d\n"
```

copia in memoria i caratteri della stringa. Si confronti questa linea di codice con la soluzione alternativa, che consiste nello scrivere ciascun carattere come valore ASCII (la figura 3.15 descrive la codifica ASCII dei caratteri):

```
.byte 76, 97, 32, 115, 111, 109, 109, 97
.byte 32, 100, 97, 32, 48, 32, 97, 32
.byte 49, 48, 48, 32, 118, 97, 108, 101
.byte 32, 37, 100, 10, 0
```

La direttiva `.asciiz` è più leggibile dato che rappresenta i caratteri come lettere e non come numeri binari; l'assemblatore è in grado di tradurre i caratteri nell'opportuna rappresentazione binaria in modo molto più veloce e preciso di un essere umano. Le direttive per la codifica dei dati specificano i dati in una forma leggibile dall'uomo che poi l'assemblatore traduce in codifica binaria. Altre direttive di questo tipo sono descritte nel Paragrafo A.10.

Esempio

Direttive per le stringhe

Definire la sequenza di byte prodotta dalla seguente direttiva:

```
.asciiz   "Se qualcosa puo' andar male, lo fara'"
```

Soluzione

```
.byte 83,  101, 32,  113, 117, 97,  108, 99
.byte 111, 115, 97,  32,  112, 117, 111, 39
.byte 32,  97,  110, 100, 97,  114, 32,  109
.byte 97,  108, 101, 44,  32,  108, 111, 32
.byte 102, 97,  114, 97,  39,  0
```

Le *macro* sono un semplice meccanismo di ricerca e sostituzione che consente di dare un nome alle sequenze di istruzioni usate frequentemente. Invece di scrivere ripetutamente le stesse istruzioni ogni volta che sono necessarie, un programmatore può invocare una macro e l'assemblatore sostituisce la chiamata alla macro con la corrispondente sequenza di istruzioni. Le macro, come le procedure, consentono al programmatore di creare nuovi livelli di astrazione per operazioni di tipo comune. A differenza delle procedure, però, le macro non provocano una chiamata a procedura ed il conseguente ritorno durante l'esecuzione del programma: sono sostituite dal corpo della macro stessa direttamente dall'assemblatore. Dopo la sostituzione il codice assembler ottenuto è indistinguibile da un programma equivalente scritto senza fare ricorso alle macro.

Esempio

Le macro

Come esempio si supponga che un programmatore debba stampare molti numeri. La procedura di libreria `printf` accetta come argomenti una stringa di formato ed uno o più valori da stampare. Il programmatore può stampare il numero intero contenuto nel registro `$7` utilizzando le seguenti istruzioni:

```
          .data
int_str:  .asciiz "%d"
          .text
          la   $a0, int_str# carica l'indirizzo della
                           # stringa nel primo argomento
          mov  $a1, $7     # carica il valore nel secondo
                           # argomento
          jal  printf      # chiama la procedura printf
```

La direttiva `.data` indica all'assemblatore di memorizzare la stringa nel segmento di dati del programma, mentre la direttiva `.text` dice di memorizzare le istruzioni nel segmento di testo.

Stampare molti numeri in questo modo però è piuttosto noioso e produce un programma lungo e difficile da capire. Un'alternativa consiste nell'introdurre una macro, `print_int`, che stampa un numero intero:

```
          .data
int_str:  .asciiz "%d"
          .text
          .macro  print_int($arg)
          la      $a0, int_str # carica l'indirizzo della
                               # stringa nel primo
argomento
          mov     $a1, $arg    # carica il parametro della
                               # macro (arg) nel secondo
                               # argomento
          jal     printf       # chiama la procedura printf
          .end_macro
print_int($7)
```

La macro ha un *parametro formale*, `$arg`, che dà il nome all'argomento della macro. Quando la macro viene espansa, l'argomento della chiamata viene sostituito al parametro formale in tutto il corpo della macro; l'assemblatore quindi sostituisce la chiamata con il nuovo corpo della macro. Nella prima chiamata a `print_int` l'argomento è `$7`, e quindi la macro si espande nel codice

```
la   $a0, int_str
mov  $a1, $7
jal  printf
```

Durante una seconda chiamata a `print_int`, ad esempio `print_int($t0)`, l'argomento è `$t0` e la macro viene espansa come

```
la   $a0, int_str
mov  $a1, $t0
jal  printf
```

Come viene espansa la chiamata `print_int($a0)`?

Soluzione

```
la   $a0, int_str
mov  $a1, $a0
jal  printf
```

Questo esempio mostra uno degli inconvenienti delle macro: un programmatore che usa questa macro deve sapere che `print_int` utilizza al suo interno il registro `$a0`, e che quindi non può stamparne correttamente il valore.

Interfaccia Hardware Software

Alcuni assemblatori prevedono anche delle *pseudoistruzioni*, che sono istruzioni consentite dall'assemblatore ma non implementate via hardware. Il Capitolo 3 contiene molti esempi di come l'assemblatore MIPS aggiunga pseudoistruzioni e modi di indirizzamento al relativamente limitato set di istruzioni MIPS. Ad esempio il Paragrafo 3.5 descrive come l'assemblatore converta l'istruzione `blt` in due altre istruzioni: `slt` e `bne`. Estendendo il set di istruzioni, l'assemblatore MIPS semplifica il linguaggio assembler senza complicare l'hardware. Molte pseudoistruzioni potrebbero anche essere simulate con delle macro, ma l'assemblatore MIPS genera un codice migliore nel caso delle pseudoistruzioni, perché può usare un registro dedicato (`$at`) ed è in grado di ottimizzare il codice generato.

Approfondimento: Gli assemblatori permettono di assemblare i frammenti di codice in *modo condizionato*, il che permette al programmatore di includere o escludere gruppi di istruzioni durante il processo. Questa funzionalità è particolarmente utile quando diverse versioni del programma differiscono soltanto di poco; invece di mantenere i programmi in file separati (il che complica molto la correzione degli errori nelle parti comuni) solitamente i programmatori uniscono le versioni in un unico file. Il codice specifico ad una versione è assemblato in modalità condizionata, in modo da poter essere escluso quando vengono elaborate versioni diverse del programma.

Se macro e assemblaggio condizionato sono utili, perché gli assemblatori in ambiente Unix raramente li supportano? Una ragione è che la maggior parte dei programmatori che utilizzano questo ambiente usano linguaggi ad alto livello come il C: la maggior parte del codice assembler è prodotto dai compilatori, che trovano più conveniente ripetere linee di codice piuttosto che definire delle macro. Un'altra ragione consiste nel fatto che già altri strumenti Unix (come `cpp`, il preprocessore C, o `mp4`, un elaboratore di macro di tipo generale) consentono l'utilizzo delle macro e dell'assemblaggio condizionato per i programmi assembler.

A.3 I linker

La *compilazione separata* consente ad un programma di essere suddiviso in frammenti memorizzati in file diversi; ciascun file contiene un insieme di procedure e di strutture dati correlate fra loro in modo logico, che costituiscono uno dei *moduli* di un programma più corposo. Ciascun file può essere compilato e assemblato indipendentemente dagli altri, quindi i cambiamenti ad un singolo modulo non richie-

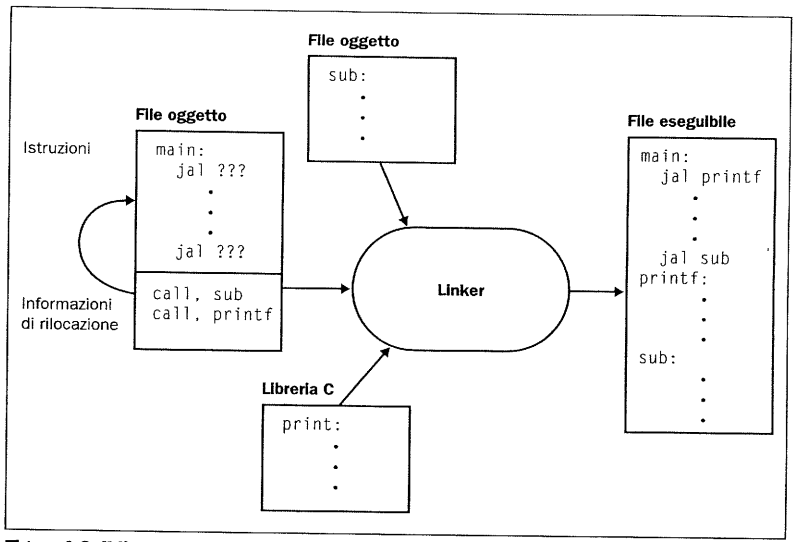


Figura A.8 Il linker esegue una ricerca all'interno dell'insieme dei file oggetto e delle librerie di programma per trovare le procedure non locali usate dal programma, combina il tutto in un unico file eseguibile e risolve i riferimenti fra procedure che si trovano in file diversi.

dono di ricompilare l'intero programma. Come descritto in precedenza, la compilazione separata necessita di un successivo passo di collegamento (link) per combinare i file oggetto dei vari moduli e risolvere i riferimenti mancanti.

Lo strumento che unisce i file si chiama *linker* (vedere figura A.8), e svolge tre tipi di compiti:

- Cerca nelle librerie le procedure usate dal programma.
- Determina le locazioni di memoria che ciascun modulo occuperà e riloca le istruzioni aggiornando i riferimenti assoluti.
- Risolve i riferimenti fra i file.

Il compito principale del linker consiste nell'assicurare che un programma non contenga etichette non definite. Esso mette in corrispondenza i simboli esterni ed i riferimenti non risolti nei vari file: un simbolo esterno contenuto in un file risolve un riferimento all'interno di un altro file se entrambi si riferiscono ad un'etichetta con lo stesso nome. Riferimenti per i quali non è stata trovata una corrispondenza indicano che è stato usato un simbolo non definito in nessuna parte del programma.

A questo livello i riferimenti non risolti non implicano necessariamente un errore del programmatore: il programma potrebbe aver fatto riferimento ad una procedura di libreria il cui codice non era contenuto nei file oggetto forniti al linker. Dopo aver trovato le corrispondenze dei simboli all'interno del programma, il linker cerca all'interno delle librerie di sistema i riferimenti a procedure e strutture dati predefinite. Le librerie di base contengono procedure che leggono e scrivono dati, che allocano e rilasciano la memoria e che svolgono operazioni sui numeri; altre librerie contengono procedure per accedere a basi di dati o per gestire finestre. Un programma che contiene riferimenti non risolti a simboli che non si trovano in nes-

suna libreria è sbagliato e non è accettato dal linker. Quando un programma usa una procedura di libreria, il linker estrae il codice della procedura dalla libreria e lo incorpora nel segmento di testo del programma. A sua volta la procedura può fare riferimento ad altre procedure di libreria, quindi il linker continua ad estrarre altre procedure finché tutti i riferimenti esterni sono stati risolti o finché non viene trovata una procedura.

Una volta che tutti i riferimenti esterni sono stati risolti, il linker determina le locazioni di memoria che verranno occupate da ciascun modulo. Dal momento che i file sono stati assemblati separatamente, l'assemblatore non sa dove saranno memorizzate le istruzioni ed i dati di ciascun modulo relativamente agli altri moduli. Quando il linker decide la collocazione di un modulo in memoria, tutti i riferimenti assoluti devono essere *rilocati* per riflettere la posizione reale. Dato che il linker ha accesso alle informazioni di rilocazione, che identificano tutti i riferimenti che hanno bisogno di questa operazione, può risolvere il problema in modo efficiente.

Il linker produce un file eseguibile che può essere eseguito su un calcolatore; di solito questo file ha lo stesso formato del file oggetto, a parte la mancanza dei riferimenti non risolti e delle informazioni di rilocazione.

A.4 Il loader

Se un programma dopo la fase di link non contiene errori, allora può essere eseguito. Inizialmente il programma risiede su di un dispositivo di memorizzazione secondaria, ad esempio un disco; nei sistemi Unix il kernel (nucleo) del sistema operativo trasferisce il programma in memoria e dà avvio all'esecuzione. Per avviare l'esecuzione di un programma il sistema operativo esegue i seguenti passi:

1. Legge l'intestazione del file eseguibile per determinare la dimensione dei segmenti di testo e di dato.
2. Crea un nuovo spazio di indirizzamento per il programma; tale spazio è grande a sufficienza per contenere i segmenti di testo e di dato, ed il segmento di stack (vedere Paragrafo A.5).
3. Copia le istruzioni e i dati dal file eseguibile al nuovo spazio di indirizzamento.
4. Copia nello stack gli argomenti passati al programma.
5. Inizializza i registri del calcolatore; in generale la maggior parte dei registri viene azzerata, mentre al puntatore dello stack viene assegnato l'indirizzo della prima locazione libera dello stack (vedere Paragrafo A.5).
6. Salta ad una procedura di inizializzazione che copia gli argomenti del programma dallo stack nei registri e che chiama la procedura `main` del programma; quando `main` restituisce il controllo alla procedura di inizializzazione, questa termina il programma con la chiamata di sistema exit.

A.5 L'utilizzo della memoria

I prossimi paragrafi approfondiscono la descrizione dell'architettura MIPS presentata nella prima parte del libro. I capitoli precedenti si sono concentrati essenzialmente sull'hardware e sulle relazioni con il software a basso livello; questi paragrafi invece si focalizzano su come i programmatori in linguaggio assembler usino l'hardware MIPS e descrivono un insieme di convenzioni seguite da molti di questi sistemi. L'hardware in generale non impone queste convenzioni, che rappresentano invece un accordo fra i programmatori per seguire lo stesso insieme di regole, in modo che il software sviluppato da persone diverse possa essere compatibile e sfruttare in modo efficace l'architettura MIPS.

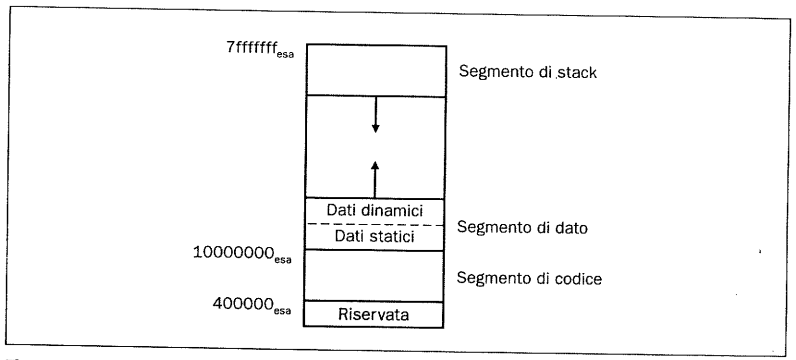


Figura A.9 L'organizzazione della memoria.

I sistemi basati su processori MIPS generalmente suddividono la memoria in tre parti (figura A.9). La prima parte, nella zona bassa dello spazio di indirizzamento (a partire dall'indirizzo 400000_{esa}), rappresenta il *segmento di testo*, che memorizza le istruzioni del programma.

La seconda parte, sopra il segmento di testo, è il *segmento di dato*, ulteriormente diviso in due parti. I *dati statici* (a partire dall'indirizzo 10000000_{esa}) contengono gli oggetti di dimensione nota al compilatore, e la cui vita (l'intervallo di tempo durante il quale il programma può farvi accesso) è l'intera durata dell'esecuzione del programma. In C ad esempio le variabili globali sono allocate in modo statico, dato che possono essere utilizzate in qualunque istante dell'esecuzione. Il linker assegna agli oggetti statici le locazioni di memoria all'interno del segmento di dato, e risolve i riferimenti a questi oggetti.

Immediatamente sopra i dati statici si trovano i *dati dinamici*; questi dati, come indica il nome, sono allocati durante l'esecuzione del programma. Nei programmi C la procedura di libreria `malloc` trova e restituisce un nuovo blocco di memoria. Dato che il compilatore non è in grado di prevedere quanta memoria allocherà il programma, il sistema operativo espande l'area dedicata ai dati dinamici per venire incontro alle richieste. Come indica la freccia verso l'alto in figura, la procedura `malloc` espande l'area dinamica con la chiamata di sistema `sbrk`, che fa sì che il sistema operativo aggiunga nuove pagine allo spazio di indirizzamento virtuale del programma (vedere il Paragrafo 7.3) estendendo il segmento dei dati dinamici.

La terza parte, il *segmento di stack*, si trova in cima allo spazio di indirizzamento virtuale (a partire dall'indirizzo $7fffffff_{esa}$). Come nel caso dei dati dinamici, la dimensione massima dello stack di un programma non è nota a propri, e quindi man mano che il programma inserisce nuovi valori nello stack il sistema operativo espande il segmento di stack verso il basso, nella direzione del segmento di dato.

La divisione in tre parti della memoria non è l'unica possibile. Ha però due importanti caratteristiche: i due segmenti che possono espandersi in modo dinamico sono più distanti possibile, e possono crescere fino ad usare l'intero spazio di indirizzamento del programma.

Interfaccia Hardware Software

Dal momento che il segmento di dato inizia ben al di sopra del programma (all'indirizzo 10000000_{esa}), le istruzioni load e store non possono riferirsi direttamente ad oggetti con il loro campo di offset a 16 bit (vedere il Paragrafo 3.4). Ad esempio per caricare la parola che si trova all'indirizzo 10008000_{esa} del segmento di dato nel registro `$v0` sono necessarie due istruzioni:

```
lui  $s0, 0x1000        # 0x1000 significa 1000 in base 16
                        # o 4096 in base 10
lw   $v0, 0x8000($s0)   # 0x10000000 + 0x8000 = 0x10008000
```

La sequenza di caratteri *0x* prima di un numero indica che questo è espresso in notazione esadecimale; ad esempio 0x8000 rappresenta 8000_{esa}, pari a 32768_{dieci}.

Per evitare di ripetere l'istruzione `lui` ad ogni load e store, i sistemi MIPS generalmente usano un registro (`$gp`) come *puntatore globale* al segmento dei dati statici. Questo registro contiene l'indirizzo 10008000_{esa}, per cui le istruzioni di load e store possono usare il loro campo offset di 16 bit (con segno) per accedere ai primi 64KB del segmento dei dati statici. Grazie al puntatore globale si può riscrivere l'esempio precedente con una sola istruzione:

```
lw   $v0, 0($gp)
```

Naturalmente il puntatore globale fa sì che gli accessi a locazioni comprese fra 10000000_{esa} e 10010000_{esa} siano più veloci di quelli a tutte le altre. Il compilatore MIPS solitamente memorizza le *variabili globali* in quest'area, dato che hanno una posizione fissa in memoria e si adattano meglio rispetto ad altri dati globali, ad esempio i vettori.

A.6 Convenzioni per le chiamate a procedura

Quando le procedure di un programma sono compilate separatamente è necessario stabilire delle convenzioni che governino l'uso dei registri: per poter lavorare su una procedura il compilatore deve sapere quali registri può usare e quali sono riservati ad altre procedure. Le regole sull'uso dei registri sono dette *convenzioni sull'uso dei registri* o *convenzioni per le chiamate a procedura;* come indica il nome queste regole sono più che altro convenzioni seguite dal software piuttosto che regole imposte dall'hardware. Molti compilatori e programmatori però cercano di rispettare queste convenzioni, dal momento che violarle significa generare errori spesso insidiosi.

Le convenzioni sulle chiamate descritte in questo paragrafo sono quelle usate dal compilatore gcc; il compilatore nativo MIPS utilizza una convenzione più complessa, leggermente più veloce.

La CPU MIPS contiene 32 registri general-purpose, numerati da 0 a 31; il registro `$0` contiene sempre il valore 0 (forzato via hardware).

- I registri `$at` (1), `$k0` (26) e `$k1` (27) sono riservati all'assemblatore e al sistema operativo, e non devono essere usati dai programmi utente o dai compilatori.
- I registri `$a0` - `$a3` (4 - 7) sono usati per passare alle procedure i primi quattro parametri (i restanti sono passati tramite lo stack). I registri `$v0` e `$v1` (2 e 3) sono usati per restituire i valori delle funzioni.
- I registri `$t0` - `$t9` (8 - 15, 24 e 25) sono registri salvati dal chiamante, e sono usati per memorizzare valori temporanei che non devono essere mantenuti durante le chiamate a procedura (vedere il Paragrafo 3.6).
- I registri `$s0` - `$s7` (16 - 23) sono registri salvati dal chiamato, e sono usati per memorizzare valori duraturi che devono essere mantenuti durante le chiamate a procedura.
- Il registro `$gp` (28) è il puntatore globale, che punta a metà di un blocco di memoria da 64K all'interno del segmento dei dati statici.
- Il registro `$sp` (29) è il puntatore allo stack, e punta all'ultima locazione libera dello stack. Il registro `$fp` (30) è il puntatore al frame. L'istruzione `jal` scrive nel registro `$ra` (31) l'indirizzo di ritorno da una procedura. Questi ultimi due registri sono analizzati nel paragrafo successivo.

Le abbreviazioni di due lettere ed i nomi dei registri (ad esempio `$sp` per il puntatore allo stack) riflettono il loro uso nelle chiamate a procedura. Per descrivere la convenzione verranno usati i nomi dei registri e non il loro numero; la tabella di figura A.10 elenca i registri specificandone l'uso previsto.

Chiamate a procedura

Questo paragrafo descrive ciò che si verifica quando una procedura (*chiamante*) chiama un'altra procedura (*chiamata*). I programmatori che scrivono in linguaggi ad alto livello (come il C o il Pascal) non vedono mai i dettagli di ciò che avviene durante le chiamate a procedura: è il compilatore a prendersi cura di questa gestione a basso livello. Chi programma in assembler, al contrario, deve realizzare in modo esplicito le operazioni di chiamata e di ritorno dalle procedure.

Gran parte della gestione associata alle chiamate a procedura ruota intorno ad un blocco di memoria detto *frame di chiamata a procedura;* questa memoria è utilizzata per molti scopi:

- mantenere i valori passati come parametri alla procedura;
- salvare i registri che una procedura potrebbe modificare, ma che il programma chiamante vuole inalterati;
- fornire spazio per le variabili locali alle procedure.

Nella maggioranza dei linguaggi di programmazione le operazioni di chiamata e di ritorno dalle procedure seguono un ordine di tipo strettamente last-in, first-out (LIFO, cioè l'ultimo elemento che entra nella coda è servito per primo), quindi la memoria può essere allocata e rilasciata all'interno dello stack; per questo motivo tali blocchi di memoria a volte sono chiamati *stack frame*.

Nome del registro	Numero	Uso
$zero	0	costante 0
$at	1	riservato all'assemblatore
$v0	2	valutazione di espressioni e risultato di una funzione
$v1	3	valutazione di espressioni e risultato di una funzione
$a0	4	argomento 1
$a1	5	argomento 2
$a2	6	argomento 3
$a3	7	argomento 4
$t0	8	temporaneo (non mantenuto durante le chiamate)
$t1	9	temporaneo (non mantenuto durante le chiamate)
$t2	10	temporaneo (non mantenuto durante le chiamate)
$t3	11	temporaneo (non mantenuto durante le chiamate)
$t4	12	temporaneo (non mantenuto durante le chiamate)
$t5	13	temporaneo (non mantenuto durante le chiamate)
$t6	14	temporaneo (non mantenuto durante le chiamate)
$t7	15	temporaneo (non mantenuto durante le chiamate)
$s0	16	temporaneo salvato (mantenuto durante le chiamate)
$s1	17	temporaneo salvato (mantenuto durante le chiamate)
$s2	18	temporaneo salvato (mantenuto durante le chiamate)
$s3	19	temporaneo salvato (mantenuto durante le chiamate)
$s4	20	temporaneo salvato (mantenuto durante le chiamate)
$s5	21	temporaneo salvato (mantenuto durante le chiamate)
$s6	22	temporaneo salvato (mantenuto durante le chiamate)
$s7	23	temporaneo salvato (mantenuto durante le chiamate)
$t8	24	temporaneo (non mantenuto durante le chiamate)
$t9	25	temporaneo (non mantenuto durante le chiamate)
$k0	26	riservato al nucleo del sistema operativo
$k1	27	riservato al nucleo del sistema operativo
$gp	28	puntatore all'area globale
$sp	29	puntatore allo stack
$fp	30	puntatore al frame
$ra	31	indirizzo di ritorno (usato nelle chiamate a funzione)

Figura A.10 Nomi dei registri e convenzioni sul loro uso.

La figura A.11 mostra un tipico stack frame, un blocco di memoria compreso fra il puntatore al frame (`$fp`), che punta alla prima parola del frame, ed il puntatore allo stack (`$sp`), che punta all'ultima parola del frame. Lo stack si sviluppa verso il basso, nella direzione di indirizzi di memoria decrescenti, per cui il puntatore al frame si trova al di sopra del puntatore allo stack. La procedura in esecuzione usa il puntatore al frame per accedere in modo veloce ai valori contenuti all'interno del suo stack frame. Ad esempio con l'istruzione seguente è possibile caricare nel registro `$v0` un argomento che si trova nello stack frame:

```
lw    $v0, 0($fp)
```

È possibile realizzare uno stack frame in molti modi diversi; in ogni caso però programma chiamato e chiamante devono stabilire una ben definita sequenza di passi. I passi descritti nel seguito descrivono la convenzione per le chiamate a procedura adottata nella maggior parte dei calcolatori MIPS. Tale convenzione entra in gioco in tre momenti durante le chiamate a procedura: immediatamente prima che

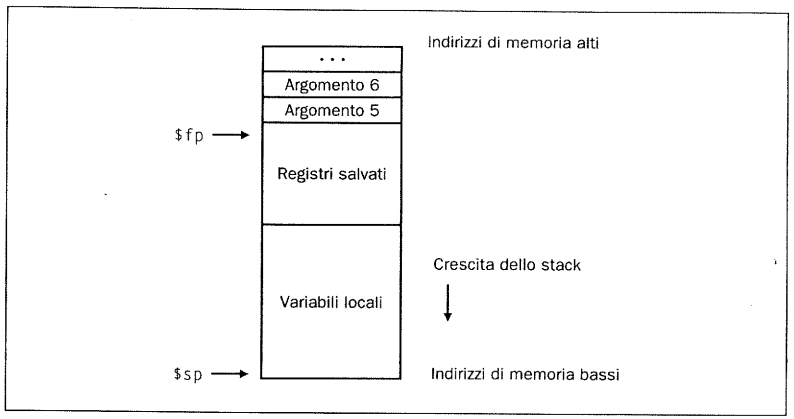


Figura A.11 Struttura di uno stack frame. Il puntatore al frame (`$fp`) punta alla prima parola dello stack frame relativo alla procedura in esecuzione. Il puntatore allo stack (`$sp`) punta all'ultima parola del frame. I primi quattro argomenti sono passati tramite registro, quindi il quinto è il primo ad essere memorizzato nello stack.

il programma chiamante invochi il chiamato, appena inizia l'esecuzione del programma chiamato, ed immediatamente prima che il chiamato restituisca il controllo al chiamante. Nella prima fase il programma chiamante mette gli argomenti relativi alla chiamata a procedura in posizioni note ed esegue le seguenti operazioni:

1. Passare gli argomenti. Per convenzione i primi quattro argomenti sono passati tramite i registri `$a0` - `$a3`; i restanti argomenti sono memorizzati nello stack e si trovano all'inizio dello stack frame della procedura chiamata.
2. Salvare i registri di competenza del chiamante. La procedura chiamata può usare questi registri (`$a0` - `$a3` e `$t0` - `$t9`) senza salvare il loro valore precedente; se il chiamante prevede usare questi registri dopo la chiamata deve salvarne il valore prima di eseguire la chiamata.
3. Eseguire un'istruzione `jal` (vedere il Paragrafo 3.6), che salta alla prima istruzione del programma chiamato e salva l'indirizzo di ritorno nel registro `$ra`.

Prima che la procedura chiamata inizi ad eseguire le proprie istruzioni, deve fare alcune operazioni per creare il suo stack frame:

1. Allocare memoria per lo stack frame sottraendo la dimensione del frame dal puntatore allo stack.
2. Salvare nel frame i registri di competenza del chiamato. Il programma chiamato deve salvare il valore dei registri `$s0` - `$s7`, `$fp`, e `$ra` prima di modificarli, dal momento che il chiamante si aspetta di trovarli inalterati dopo la chiamata. Il registro `$fp` è salvato da tutte le procedure che allocano un nuovo stack frame. Il registro `$ra` deve essere salvato soltanto se il programma chiamato chiama a sua volta un'altra procedura. Allo stesso modo devono essere salvati i registri `$s0` - `$s7` se effettivamente usati dal programma chiamato.
3. Inizializzare il puntatore a frame aggiungendo a `$sp` la dimensione del frame meno quattro e memorizzando questo valore nel registro `$fp`.

Interfaccia Hardware Software

La convenzione del MIPS sull'uso dei registri fa sì che esistano sia registri il cui salvataggio è di competenza del programma chiamato che registri il cui salvataggio è di competenza del programma chiamante, dal momento che entrambi i meccanismi sono vantaggiosi in circostanze diverse. I registri del primo tipo sono adatti a mantenere valori di lunga durata, come le variabili del programma utente; durante l'esecuzione di una procedura questi registri sono salvati soltanto se il programma chiamato prevede di usarli. Quelli del secondo tipo sono adatti a mantenere valori di breve durata che non vengono mantenuti dopo una chiamata a procedura, come i dati usati per calcolare gli indirizzi di memoria; durante l'esecuzione di una procedura il programma chiamato può usare questi registri anche per valori temporanei di breve durata.

Infine il programma chiamato restituisce il controllo al chiamante eseguendo le seguenti operazioni:

1. Se il programma chiamato è una funzione che restituisce un valore, mette tale valore nel registro `$v0`.
2. Ripristina tutti i registri di competenza del chiamato che erano stati salvati all'inizio della procedura.
3. Libera lo stack frame sommando a `$sp` la dimensione del frame.
4. Restituisce il controllo eseguendo un salto all'indirizzo contenuto in `$ra`.

Approfondimento: I linguaggi di programmazione che non consentono le procedure ricorsive (procedure che chiamano se stesse direttamente o indirettamente attraverso una catena di chiamate) non hanno bisogno di allocare il frame nello stack. In un linguaggio non ricorsivo è possibile allocare in modo statico il frame di ciascuna procedura, dato che una sola chiamata alla stessa procedura può essere attiva in un dato momento. Le vecchie versioni del Fortran proibivano la ricorsione, dal momento che l'allocazione statica dei frame consentiva la generazione di codice che poteva essere eseguito più velocemente sui vecchi calcolatori. In architetture di tipo load-store come il MIPS, tuttavia, lo stack frame può essere altrettanto veloce perché il puntatore al frame punta direttamente allo stack frame attivo, il che permette alla singola istruzione di load o di store di accedere ai valori contenuti nel frame. In più la ricorsione è una tecnica di programmazione di grande utilità.

Esempio di chiamata a procedura

Come esempio si consideri il programma C

```
main()
{
  printf("Il fattoriale di 10 vale %d\n", fatt(10));
}

int fatt(int n)
{
  if (n < 1)
    return (1);
  else
    return (n * fatt(n - 1));
}
```

che calcola e stampa 10! (il fattoriale di 10, cioè 10! = 10 × 9 × ... × 1). `fatt` è una procedura ricorsiva che calcola $n!$ moltiplicando n per $(n - 1)!$. Il codice assembler di questa procedura mostra in che modo i programmi gestiscano lo stack frame.

All'inizio la procedura `main` crea il proprio stack frame e salva i due registri di competenza del chiamato che prevede di utilizzare: `$fp` e `$ra`. Il frame è più grande di quanto richiesto dai due registri, dal momento che la convenzione di chiamata impone che la dimensione minima di uno stack frame sia di 24 byte. Tale dimensione minima è sufficiente a contenere i quattro registri argomento (`$a0 - $a3`) e l'indirizzo di ritorno `$ra`, allineati alla doppia parola (doubleword). Dato che `main` deve anche salvare `$fp`, il suo stack frame deve avere due parole in più (si ricordi che lo stack frame è sempre multiplo di una quantità pari a due parole).

```
        .text
        .globl main
main:
        subu    $sp, $sp, 32  # lo stack frame è grande 32 byte
        sw      $ra, 20($sp)  # salva l'indirizzo di ritorno
        sw      $fp, 16($sp)  # salva il vecchio puntatore al frame
        addu    $fp, $sp, 28  # inizializza il puntatore al frame
```

La procedura `main` chiama quindi la procedura che calcola il fattoriale passandole 10 come unico argomento. Dopo che `fatt` termina, `main` chiama la procedura di libreria `printf` passandole il formato della stringa ed il valore restituito da `fatt`:

```
        li      $a0, 10       # mette l'argomento (10) in $a0
        jal     fatt          # chiama la funzione che calcola il
                              # fattoriale

        la      $a0, $LC      # mette la stringa di formato in $a0
        move    $a1, $v0      # chiama la funzione di stampa
```

Infine, dopo la stampa del fattoriale, il programma `main` termina la sua esecuzione; prima però deve ripristinare il valore dei registri salvati e rilasciare il proprio stack frame:

```
        lw      $ra, 20($sp)  # ripristina l'indirizzo di ritorno
        lw      $fp, 16($sp)  # ripristina il puntatore al frame
        addu    $sp, $sp, 32  # rilascia lo stack frame
        jr      $ra           # restituisce il controllo al
                              # programma chiamante

        .rdata
$LC:
        .ascii "Il fattoriale di 10 vale %d\n\000"
```

La struttura della procedura che calcola il fattoriale è simile a quella del `main`. Per prima cosa crea uno stack frame e salva i registri di propria competenza che prevede di usare; oltre a salvare `$ra` e `$fp`, `fatt` salva anche il suo argomento (`$a0`) che userà per la chiamata ricorsiva:

```
        .text
fatt:
        subu    $sp, $sp, 32  # lo stack frame è grande 32 byte
        sw      $ra, 20($sp)  # salva l'indirizzo di ritorno
        sw      $fp, 16($sp)  # salva il vecchio puntatore al frame
        addu    $fp, $sp, 28  # inizializza il puntatore al frame
        sw      $a0, 0($fp)   # salva l'argomento (n)
```

Il cuore della procedura `fatt` esegue il calcolo (vedere la procedura in C): controlla se l'argomento è maggiore di 0 e se non lo è restituisce 1. Se l'argomento è maggiore di 0, la procedura chiama ricorsivamente se stessa per calcolare `fatt(n-1)` e moltiplica tale valore per *n*:

```
        lw      $v0, 0($fp)   # carica n
        bgtz    $v0, $L2      # salta se n > 0
        li      $v0, 1        # restituisce 1
        j       $L1           # salta al codice che termina la
                              # procedura

$L2:
        lw      $v1, 0($fp)   # carica n
        subu    $v0, $v1, 1   # calcola n-1
        move    $a0, $v0      # sposta il valore in $a0
        jal     fatt          # chiama la funzione che calcola il
                              # fattoriale

        lw      $v1, 0($fp)   # carica n
        mul     $v0, $v0, $v1 # calcola fatt(n-1) * n
```

Infine la procedura ripristina i registri salvati e restituisce il valore nel registro `$v0`:

```
$L1:                          # il risultato è in $v0
        lw      $ra, 20($sp)  # ripristina $ra
        lw      $fp, 16($sp)  # ripristina $fp
        addu    $sp, $sp, 32  # rilascia lo stack
        j       $ra           # restituisce il controllo al
                              # programma chiamante
```

Esempio

Lo stack nelle procedure ricorsive

La figura A.12 mostra lo stack al momento della chiamata `fatt(7)`. La procedura `main` viene eseguita per prima, quindi il suo stack frame si trova in alto; `main` chiama `fatt(10)`, il cui stack frame è nella posizione successiva. Ciascuna chiamata ricorsiva a `fatt` calcola il fattoriale del numero immediatamente precedente, e lo stack frame riflette l'ordine (LIFO) di queste chiamate. Come appare lo stack alla fine della chiamata `fatt(10)`?

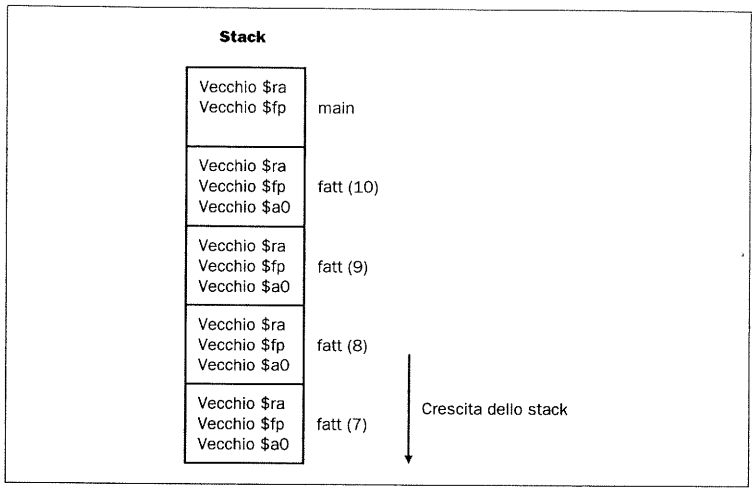
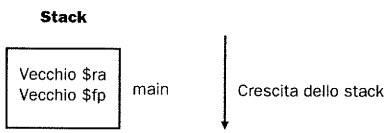


Figura A.12 Gli stack frame durante la chiamata `fatt(7)`.

Soluzione



Approfondimento: La differenza fra il compilatore MIPS ed il compilatore gcc è che il primo solitamente non usa un puntatore a frame, quindi tale registro è disponibile come ulteriore registro di tipo save, `$s8`. Tale modifica fa risparmiare un paio di istruzioni durante la chiamata ed il ritorno dalle procedure, ma complica la generazione del codice perché la procedura è costretta ad accedere allo stack frame tramite `$sp`, il cui valore può cambiare durante l'esecuzione della procedura se vengono memorizzati altri dati nello stack.

Un altro esempio di chiamata a procedura

Come ulteriore esempio si consideri la seguente procedura che calcola la funzione `tak`, uno dei benchmark più usati, creato da Ikuo Takeuchi. Questa funzione non calcola niente di significativo, ma è un programma altamente ricorsivo che illustra le convenzioni del MIPS sulle chiamate a procedura.

```
int tak(int x, int y, int z)
{
  if (x < y)
    return 1 + tak (tak (x - 1, y, z),
      tak (y - 1, z, x),
      tak (z - 1, x, y));
  else
    return z;
}
```

```
int main()
{
  tak(18, 12, 6);
}
```

Il codice assembler di questo programma è riportato nel seguito. Per prima cosa la funzione `tak` salva il suo registro di ritorno nel proprio stack frame, ed i suoi argomenti nei registri di competenza del chiamato, dato che la procedura potrebbe utilizzare chiamate che necessitano dell'uso dei registri `$a0` - `$a2` e `$ra`. La funzione usa i registri di competenza del chiamato per mantenere valori che persistono durante tutta la vita della funzione, il che include parecchie chiamate che potenzialmente possono modificare il contenuto dei registri.

```
        .text
        .globl tak
tak:
        subu    $sp, $sp, 40
        sw      $ra, 32($sp)

        sw      $s0, 16($sp)  # x
        move    $s0, $a0
        sw      $s1, 20($sp)  # y
        move    $s1, $a1
        sw      $s2, 24($sp)  # z
        move    $s2, $a2
        sw      $s3, 28($sp)  # temporaneo
```

La procedura quindi inizia l'esecuzione controllando se `y < x`; se questo non si verifica salta all'etichetta `L1`, più in basso.

```
        bge     $s1, $s0, L1  # se (x < y) salta a L1
```

Se `y < x` viene eseguito il corpo della procedura, che contiene quattro chiamate ricorsive. La prima chiamata usa all'incirca gli stessi argomenti della procedura padre:

```
        addu    $a0, $s0, -1
        move    $a1, $s1
        move    $a2, $s2
        jal     tak           # tak (x - 1, y, z)
        move    $s3, $v0
```

Si noti che il risultato della prima chiamata ricorsiva viene salvato nel registro `$s3`, in modo da poter essere usato più avanti.

Ora la funzione prepara gli argomenti per la successiva chiamata ricorsiva.

```
        addu    $a0, $s1, -1
        move    $a1, $s2
        move    $a2, $s0
        jal     tak           # tak (y - 1, z, x)
```

Nelle istruzioni seguenti, il risultato di questa chiamata ricorsiva viene salvato nel registro `$s0`; prima però è necessario leggere per l'ultima volta il valore del primo argomento, che era stato salvato in questo registro.


```
        addu    $a0, $s2, -1
        move    $a1, $s0
        move    $a2, $s1
        move    $s0, $v0
        jal     tak           # tak (z - 1, x, y)
```

Dopo le tre chiamate ricorsive interne, è il momento della chiamata ricorsiva finale; dopo di essa il risultato è in `$v0`, ed il controllo salta alla fine della funzione.

```
        move    $a0, $s3
        move    $a1, $s0
        move    $a2, $v0
        jal     tak           # tak (tak(...),tak(...), tak(...))
        addu    $v0, $v0, 1
        j       L2
```

Il codice dopo l'etichetta `L1` è relativo alla seconda parte del ciclo *if-then-else*: sposta semplicemente il valore dell'argomento `z` nel registro di ritorno per prepararsi alla parte conclusiva della funzione.

```
L1:
        move    $v0, $s2
```

Il codice sottostante rappresenta la parte conclusiva della funzione, che ripristina i registri salvati e restituisce al chiamante il risultato della funzione.

```
L2:
        lw      $ra, 32($sp)
        lw      $s0, 16($sp)
        lw      $s1, 20($sp)
        lw      $s2, 24($sp)
        lw      $s3, 28($sp)
        addu    $sp, $sp, 40
        j       $ra
```

Il programma `main` chiama la procedura `tak` con gli argomenti iniziali, quindi prende il risultato del calcolo (7) e lo stampa usando la chiamata di sistema SPIM che stampa i numeri interi.

```
        .globl main
main:
        subu    $sp, $sp, 24
        sw      $ra, 16($sp)
        li      $a0, 18
        li      $a1, 12
        li      $a2, 6
        jal     tak           # tak (18, 12, 6)
        move    $a0, $v0
        li      $v0, 1        # chiamata di sistema print_int
        syscall
        lw      $ra, 16($sp)
        addu    $sp, $sp, 24
        j       $ra
```

A.7 Eccezioni e interruzioni

Il Paragrafo 5.6 descrive le funzionalità MIPS per la gestione delle eccezioni, in risposta sia ad eccezioni causate da errori durante l'esecuzione di un'istruzione che ad interruzioni esterne provocate da dispositivi di I/O; questo paragrafo descrive in maggior dettaglio la gestione di eccezioni ed interruzioni. Nei processori MIPS una parte della CPU, detta *coprocessore 0*, mantiene le informazioni necessarie ai programmi per gestire eccezioni ed interruzioni. Il simulatore SPIM del MIPS non implementa tutti i registri del coprocessore 0, dal momento che molti di questi non sono utili in un simulatore o sono parte del sottosistema di memoria, che SPIM non implementa. SPIM fornisce i seguenti registri del coprocessore 0:

Nome del registro	Numero del registro	Uso
BadVAddr	8	registro contenente l'indirizzo di memoria a cui è stato fatto riferimento
Stato	12	maschera delle interruzioni e bit di abilitazione
Causa	13	tipo dell'interruzione e bit dell'interruzione pendente
EPC	14	registro contenente l'indirizzo dell'istruzione che ha causato l'interruzione

Questi quattro registri sono parte del set dei registri del coprocessore 0, ed è possibile accedervi con le istruzioni `lwc0`, `mfc0`, `mtc0` e `swc0`. Quando si verifica un'eccezione il registro EPC contiene l'indirizzo dell'istruzione in esecuzione; se l'istruzione ha fatto un accesso in memoria che ha provocato l'eccezione, il registro BadVAddr contiene la locazione di memoria a cui è stato fatto riferimento. Gli altri due registri sono composti da diversi campi, e sono descritti nel seguito.

La figura A.13 mostra i campi del registro Stato implementati dal simulatore SPIM del MIPS. Il campo `maschera delle interruzioni` contiene un bit per ciascuno dei cinque possibili livelli di interruzione hardware e dei tre possibili livelli di interruzione software; un bit a 1 abilita l'interruzione corrispondente, mentre un bit a 0 la disabilita. I 6 bit meno significativi del registro Stato implementano uno stack di profondità tre per i bit `kernel/utente` e `abilitazione delle interruzioni`. Il bit `kernel/utente` vale 0 se il programma si trovava nel kernel (nucleo di massima priorità) quando si è verificata l'interruzione, e 1 se era in modalità utente; se il bit di `abilitazione delle interruzioni` è a 1 le interru-

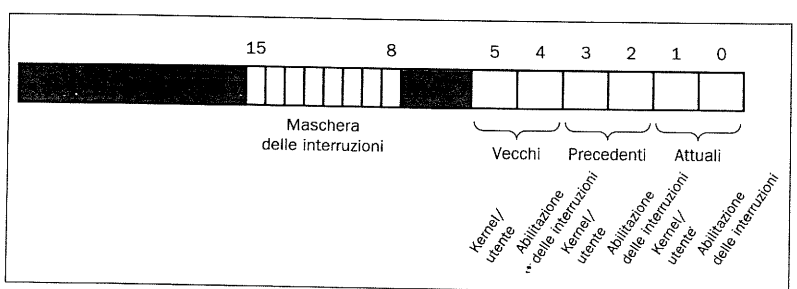


Figura A.13 Il registro Stato.

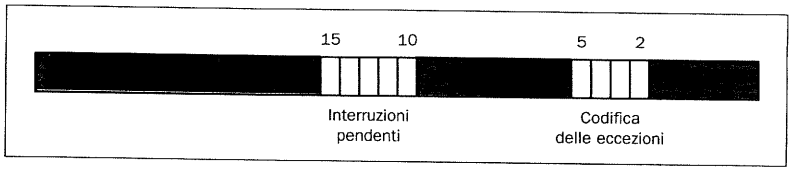


Figura A.14 Il registro Causa. Nei processori MIPS reali questo registro contiene dei campi aggiuntivi che riportano se l'istruzione che ha causato l'interruzione era eseguita in un branch delay slot, quale coprocessore ha causato l'eccezione, oppure se c'è un'interruzione software pendente.

zioni sono abilitate, se è a 0 sono disabilitate. Quando si verifica un'interruzione questi 6 bit sono scalati a sinistra di 2 posizioni, per cui i bit attuali diventano i bit precedenti, ed i bit precedenti diventano bit vecchi (i bit vecchi vengono scartati). Entrambi i bit attuali sono posti a 0, per cui il gestore delle interruzioni viene eseguito nel kernel con le interruzioni disabilitate.

La figura A.14 mostra i campi del registro Causa implementati dal simulatore SPIM. I cinque bit delle `interruzioni pendenti` corrispondono ai cinque livelli di interruzione; un bit assume il valore 1 quando si verifica un'interruzione non ancora servita del corrispondente livello. Il campo Codifica delle eccezioni descrive la causa dell'eccezione secondo la seguente codifica:

Numero	Nome	Descrizione
0	INT	interruzione esterna
4	ADDRL	eccezione dovuta ad errore di indirizzamento (caricamento o prelievo di un'istruzione)
5	ADDRS	eccezione dovuta ad errore di indirizzamento (memorizzazione)
6	IBUS	errore sul bus durante il prelievo di un'istruzione
7	DBUS	errore sul bus durante il caricamento o la memorizzazione di un dato
8	SYSCALL	eccezione dovuta ad una chiamata di sistema
9	BKPT	eccezione dovuta ad un breakpoint
10	RI	eccezione dovuta ad un'istruzione riservata
12	OVF	eccezione dovuta ad overflow aritmetico

Eccezioni ed interruzioni fanno sì che il processore MIPS trasferisca il controllo ad un frammento di codice, chiamato *gestore delle interruzioni*, che si trova all'indirizzo 80000080_{esa} (nel kernel, non nello spazio di indirizzamento dell'utente). Questo codice esamina la causa dell'eccezione ed esegue un salto alla porzione appropriata del sistema operativo, che in risposta all'eccezione termina il processo che l'ha provocata o svolge una determinata azione. Un processo che ha provocato un errore, come il tentativo di eseguire un'operazione non implementata, viene terminato dal sistema operativo, mentre eccezioni come i page fault hanno lo scopo di richiedere un servizio al sistema operativo, ad esempio in questo caso prendere una pagina di memoria dal disco: il sistema operativo elabora queste richieste e riavvia il processo. L'ultimo tipo di eccezioni è quello relativo alle interruzioni da parte dei dispositivi esterni, che generalmente fanno sì che il sistema operativo sposti dei dati da o verso un dispositivo di I/O e riavvii il processo interrotto. Il codice dell'esempio successivo descrive un semplice gestore delle interruzioni, che chiama una procedura per stampare un messaggio ogni volta che si verifica un'eccezione (ma non un'interruzione). Tale codice è simile a quello usato dal simulatore

SPIM, eccetto il fatto di non stampare un messaggio di errore al verificarsi di un'interruzione.

Esempio

Il gestore delle interruzioni

Per prima cosa il gestore delle interruzioni salva il contenuto dei registri `$a0` e `$a1`, che userà in seguito per passare gli argomenti. Il gestore delle interruzioni non può memorizzare sullo stack i vecchi valori di questi registri, come farebbe una normale procedura, perché la causa dell'interruzione potrebbe essere un riferimento alla memoria attraverso un valore non consentito (ad esempio 0) del puntatore allo stack. Invece di usare lo stack il gestore delle interruzioni salva il contenuto dei registri in due locazioni di memoria, `save0` e `save1`. Se la stessa procedura di interruzione può essere a sua volta interrotta due locazioni non sono sufficienti, dato che la seconda interruzione sovrascriverebbe i valori salvati durante la prima interruzione. Questo semplice gestore delle interruzioni, però, termina l'esecuzione della procedura prima di riabilitare le interruzioni, quindi il problema non sussiste.

```
.ktext 0x80000080
sw   $a0, save0      # il gestore non è rientrante e non può
sw   $a1, save1      # usare lo stack per salvare $a0 e $a1
                     # non è necessario salvare $k0 e $k1
```

Quindi il gestore delle interruzioni sposta il contenuto dei registri Causa e EPC nei registri della CPU; questi due registri non sono parte del set di registri della CPU, ma sono registri del coprocessore 0, la parte della CPU che si occupa della gestione delle interruzioni. L'istruzione `mfc0 $k0, $13` sposta il contenuto del registro 13 del coprocessore 0 (il registro Causa) nel registro `$k0` della CPU. Si noti che il gestore delle interruzioni non deve necessariamente salvare i registri `$k0` e `$k1` perché i programmi utente non usano tali registri. Il gestore delle interruzioni usa i valori all'interno del registro Causa per controllare se l'eccezione è stata provocata da un'interruzione (vedere la tabella precedente); in questo caso l'eccezione è ignorata, mentre in caso contrario il gestore chiama la procedura `print_excp` per stampare un messaggio di avvertimento.

```
mcf0 $k0, $13        # sposta Causa in $k0
mcf0 $k1, $14        # sposta EPC in $k1

sgt  $v0, $k0, 0x44  # ignora le interruzioni
bgtz $v0, fatto

mov  $a0, $k0        # sposta Causa in $a0
mov  $a1, $k1        # sposta EPC in $a1
jal  print_excp      # stampa il messaggio di errore dovuto
                     # all'eccezione
```

Prima di terminare, il gestore delle interruzioni ripristina il contenuto dei registri `$a0` e `$a1` e poi esegue l'istruzione `rfe` (ritorno da un'eccezione), che ripristina i bit della maschera delle interruzioni e kernel/utente che erano presenti in precedenza nel registro Stato. Questo fa sì che il processore ritorni allo stato precedente al verificarsi dell'eccezione e che si prepari a riprendere l'esecuzione del programma. Il gestore delle interruzioni quindi restituisce il controllo al programma saltando all'istruzione successiva a quella che ha causato l'eccezione.

```
fatto:
        lw      $a0, save0
        lw      $a1, save1
        addiu   $k1, $k1, 4  # non riesegue l'istruzione
                             # che ha provocato l'eccezione
        rfe                  # ripristina lo stato delle
                             # interruzioni
        jr      $k1

        .kdata
save0:  .word 0
save1:  .word 0
```

Approfondimento: Nei processori MIPS reali il processo di ritorno dal gestore delle interruzioni è più complesso. L'istruzione `rfe` deve essere eseguita nel branch delay slot dell'istruzione `jr` (vedere l'approfondimento del Paragrafo 6.1) che restituisce il controllo al programma utente, in modo che nessuna istruzione di gestione delle interruzioni venga eseguita con i bit della maschera delle interruzioni e kernel/utente propri del programma utente. In più il gestore delle interruzioni non sempre può saltare all'istruzione seguente a quella contenuta nel registro EPC: ad esempio se l'istruzione che ha provocato l'eccezione era in un branch delay slot (vedere il Capitolo 6), l'istruzione seguente potrebbe non essere quella che si trova nella posizione di memoria successiva.

A.8 Ingresso e uscita

SPIM simula un dispositivo di I/O: un terminale mappato in memoria (memory-mapped). Quando un programma è in esecuzione, SPIM connette al processore il proprio terminale (o una finestra di console separata nella versione `xspim` per X-window). Un programma MIPS eseguito con SPIM può leggere i caratteri inseriti dall'utente e in più, se il programma scrive dei caratteri sul terminale, questi appaiono sul terminale SPIM o sulla finestra della console. Un'eccezione a questa regola è il control-C: questo carattere non viene passato al programma, ma interrompe l'esecuzione di SPIM tornando alla modalità di inserzione comandi. Quando il programma interrompe l'esecuzione (ad esempio per via di un control-C o perché è stato incontrato un breakpoint) il terminale è nuovamente connesso a `spim` ed è quindi nuovamente possibile inserire dei comandi. Per poter usare il dispositivo di I/O mappato in memoria occorre avviare `spim` o `xspim` con l'opzione `-mapped_io` (vedere oltre).

Il terminale è composto da due unità indipendenti: un *ricevitore* ed un *trasmettitore*; il ricevitore legge i caratteri inseriti da tastiera, mentre il trasmettitore li visualizza sullo schermo. Le due unità sono completamente indipendenti, il che significa ad esempio che i caratteri inseriti da tastiera non sono automaticamente visualizzati sullo schermo. Un programma deve realizzare questo in modo esplicito, leggendo un carattere dal ricevitore e scrivendolo nel trasmettitore.

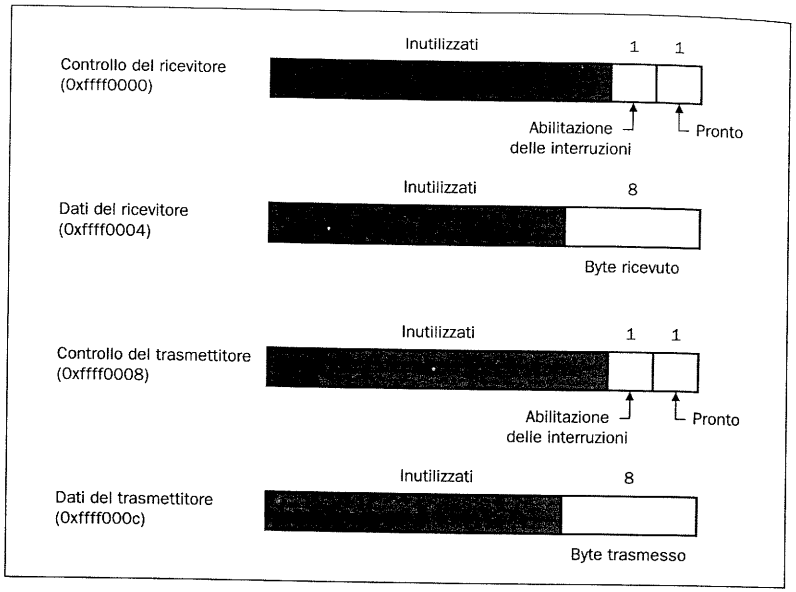


Figura A.15 Il terminale è controllato da quattro registri del dispositivo, ciascuno dei quali è visto come una locazione di memoria ad un determinato indirizzo. Solo pochi bit di questi registri sono effettivamente usati, gli altri sono sempre 0 in lettura, e non possono essere scritti.

Il programma controlla il terminale tramite i quattro registri del dispositivo mappato in memoria mostrati in figura A.15. Il termine "mappato in memoria" sta ad indicare che ciascuno dei registri è visto come una speciale locazione di memoria. Il *registro di Controllo del Ricevitore* si trova all'indirizzo $ffff0000_{esa}$, e di questo solo due bit sono effettivamente usati; il bit 0 è chiamato di "pronto": se è a 1 significa che la tastiera ha inviato un carattere, ma che questo non è ancora stato letto dal registro dei Dati del Ricevitore. Il bit di pronto può solo essere letto: le operazioni di scrittura sono ignorate; tale bit cambia da 0 a 1 quando viene introdotto un carattere da tastiera, e da 1 a 0 quando il carattere è letto dal registro dei Dati del Ricevitore.

Il bit 1 del registro di Controllo del Ricevitore è quello di abilitazione delle interruzioni della tastiera; questo bit può essere sia letto che scritto dal programma. Il bit di abilitazione delle interruzioni è inizialmente posto a 0; se un programma lo mette a 1, il terminale genera un'interruzione di livello 0 quando il bit di pronto è uguale a 1. Affinché però l'interruzione abbia effetto anche sul processore, l'interruzione deve essere abilitata anche nel registro Stato (vedere il Paragrafo A.7). Tutti gli altri bit del registro di Controllo del Ricevitore sono inutilizzati.

Il secondo registro del dispositivo è il *registro dei Dati del Ricevitore* e si trova all'indirizzo $ffff0004_{esa}$. Gli 8 bit meno significativi di questo registro contengono l'ultimo carattere introdotto da tastiera, mentre tutti gli altri bit sono a 0; questo registro può solo essere letto, e cambia solamente quando un nuovo carattere è letto da tastiera. La lettura del registro dei Dati del Ricevitore rimette a 0 il bit di pronto del registro di Controllo del Ricevitore.

Il terzo registro del dispositivo è il *registro di Controllo del Trasmettitore* e si trova all'indirizzo $ffff0008_{esa}$; di questo registro sono usati solo i due bit meno significativi, che si comportano in modo molto simile ai corrispondenti bit del registro di Controllo del Ricevitore. Il bit 0 è chiamato di "pronto", ed è in sola lettura; se è a 1 il trasmettitore è pronto ad accettare un nuovo carattere in uscita, mentre se è a 0 il trasmettitore è ancora occupato a scrivere il carattere precedente. Il bit 1 è quello di abilitazione delle interruzioni, e può essere sia letto che scritto; se è a 1 il terminale genera un'interruzione di livello 1 quando il bit di pronto è uguale a 1.

L'ultimo registro del dispositivo è il *registro dei Dati del Trasmettitore* e si trova all'indirizzo $ffff000c_{esa}$. Quando viene scritto un valore in questa locazione, i suoi 8 bit meno significativi (cioè il corrispondente carattere ASCII come da figura 3.15) sono inviati alla console. Quando questo registro viene scritto, il bit di pronto del registro di Controllo del Trasmettitore viene posto a 0; tale bit rimane a 0 fino a quando non è trascorso un tempo sufficiente a trasmettere il carattere al terminale, e poi ritorna a 1. Il registro dei Dati del Trasmettitore può essere scritto solo quando il bit di pronto del registro di Controllo del Trasmettitore è a 1; se il trasmettitore non è pronto, le operazioni di scrittura sul registro dei Dati del Trasmettitore sono ignorate (sembra che l'operazione di scrittura abbia buon fine ma il carattere non appare in uscita).

I calcolatori reali necessitano di un certo tempo per inviare i caratteri sulla linea seriale che connette i terminali ai calcolatori; questi ritardi sono simulati da SPIM: ad esempio, dopo che il trasmettitore inizia a scrivere un carattere, il suo bit di pronto è portato a 0 per un po' di tempo. SPIM misura il tempo in istruzioni eseguite, e non in cicli di clock veri e propri; questo significa che il trasmettitore non è pronto finché il processore non esegue un certo numero di istruzioni. Se si prova ad interrompere la macchina e si osserva il bit di pronto, si vede che non è cambiato; se invece si lascia continuare l'esecuzione dopo un po' il bit torna a 1.

A.9 SPIM

SPIM è un simulatore software che permette l'esecuzione di programmi scritti per i processori MIPS R2000/R3000; il nome SPIM è semplicemente MIPS letto al contrario. SPIM è in grado di leggere ed eseguire immediatamente file scritti in linguaggio assembler o (nel caso di alcuni sistemi) file eseguibili MIPS. SPIM è un sistema completo per l'esecuzione di programmi MIPS: contiene un debugger e fornisce alcuni servizi analoghi a quelli di un sistema operativo. SPIM è più lento di un calcolatore reale (di un fattore pari a 100 o maggiore), ma il suo basso costo e l'ampia disponibilità non sono assolutamente ottenibili da un hardware reale!

Una domanda ovvia è: perché usare un simulatore quando molti possiedono stazioni di lavoro al cui interno ci sono dei chip MIPS significativamente più veloci di SPIM? Un primo motivo è che queste stazioni di lavoro non sono universalmente disponibili, un secondo è che il rapido progresso nella direzione di calcolatori nuovi e più veloci potrebbe rendere obsolete queste macchine (vedere il Capitolo 1). La tendenza attuale è quella di rendere i calcolatori più veloci eseguendo diverse istruzioni in modo concorrente: tale tendenza fa sì che le architetture siano più difficili da capire e programmare. L'architettura MIPS può essere considerata l'esempio tipico di una semplice, lineare macchina RISC.

I simulatori inoltre possono rappresentare un ambiente di programmazione migliore delle macchine reali, dato che sono in grado di riconoscere più errori e di fornire maggiori funzionalità rispetto ai calcolatori reali. SPIM, ad esempio, possiede un'interfaccia X-window che funziona meglio della maggior parte dei debugger delle macchine reali.

Infine i simulatori sono un utile strumento per studiare i calcolatori ed i programmi eseguiti su di essi. Dal momento che sono implementati via software, e non su silicio, possono essere facilmente modificati per aggiungere nuove istruzioni, per costruire nuovi sistemi come i multiprocessore, o semplicemente per raccogliere dati.

La simulazione di una macchina virtuale

L'architettura MIPS, come quella di molti calcolatori RISC, è difficile da programmare in modo diretto per via dei salti condizionati ritardati, delle operazioni di caricamento ritardate, e dei modi di indirizzamento limitati. La difficoltà è tollerabile, dato che questi calcolatori sono stati progettati per essere programmati in linguaggi ad alto livello e presentano quindi un'interfaccia adatta ai compilatori piuttosto che ai programmatori in linguaggio assembler. Buona parte della complessità della programmazione è dovuta alle istruzioni ritardate. Un *salto condizionato ritardato* richiede due cicli di esecuzione (vedere gli approfondimenti dei Paragrafi 6.1 e 6.6); nel secondo ciclo viene eseguita l'istruzione immediatamente seguente il salto. Questa istruzione può svolgere del lavoro utile che normalmente sarebbe stato eseguito prima del salto, ma può anche essere una `nop` (nessuna operazione). Analogamente le operazioni di *caricamento ritardato* richiedono due cicli, per cui l'istruzione immediatamente seguente a quella di load non può usare il valore prelevato dalla memoria (vedere il Paragrafo 6.2).

Saggiamente il MIPS ha scelto di nascondere questo livello di complessità facendo sì che il suo assemblatore implementasse una *macchina virtuale*; questo calcolare virtuale sembra avere salti condizionati ed operazioni di caricamento non ritardati, ed un set di istruzioni più ricco di quello dell'hardware reale: l'assemblatore *riorganizza* le istruzioni per riempire gli slot di ritardo. Inoltre il calcolatore virtuale fornisce delle *pseudoistruzioni*, che appaiono come istruzioni vere e proprie nei programmi scritti in codice assembler; l'hardware però ignora del tutto le pseudoistruzioni, per cui l'assemblatore deve tradurle in sequenze equivalenti formate da istruzioni macchina effettive. Ad esempio l'hardware MIPS fornisce istruzioni che consentono di saltare solo nel caso in cui il contenuto di un registro è uguale o è diverso da 0; altri salti condizionati, come nel caso di un registro maggiore o uguale ad un altro, sono realizzate confrontando i due registri e saltando se il risultato del confronto è vero (cioè diverso da zero).

Se non diversamente specificato SPIM simula la macchina virtuale, più ricca; può però simulare anche l'hardware puro. Nel seguito viene descritta la macchina virtuale e vengono semplicemente menzionate le caratteristiche che non appartengono all'hardware effettivo. Nel fare ciò si segue la convenzione usata da chi programma in linguaggio assembler MIPS (e dai compilatori), che utilizza abitualmente la macchina estesa. Per una descrizione delle macchine reali si consiglia di leggere Gerry Kane a Joe Heinrich, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliff, NJ, 1992.

Introduzione a SPIM

Il resto di questo paragrafo contiene una descrizione completa e piuttosto dettagliata di SPIM. Tutti questi dettagli non devono essere fonte di preoccupazione per il lettore, ed il volume delle informazioni non deve offuscare il fatto che SPIM è un programma semplice e facile da usare. Questa parte contiene una guida veloce su SPIM, che permetterà al lettore di imparare a caricare, trovare gli errori ed eseguire semplici programmi MIPS. SPIM esiste in più versioni; una di queste, chiamata `spim`, è un programma con interfaccia testuale utilizzante una sola linea di comando che richiede un semplice terminale alfanumerico. Funziona come la maggior parte dei programmi di questo tipo: si scrive una linea di testo, si schiaccia il tasto `invio` e `spim` esegue il comando introdotto.

Una versione più sofisticata, chiamata `xspim`, funziona nell'ambiente X-window dei sistemi Unix, e richiede quindi un monitor grafico. `xspim`, però, è molto più facile da imparare e da usare perché i suoi comandi sono sempre visibili sullo schermo e perché mostra continuamente il contenuto dei registri della macchina. Un'altra versione, `PCspim`, è compatibile con Windows 3.1, Windows 95 e Windows NT. Le versioni per UNIX, Windows e DOS di SPIM sono disponibili all'indirizzo *www.mkp.com/cod2e.htm*.

Dato che in molti usano e preferiscono `xspim`, questo paragrafo si occupa soltanto di questo programma; il lettore che intenda usare qualunque versione di `spim` è invitato a non saltare il paragrafo, ma a leggerlo e poi a consultare il Paragrafo "Opzioni SPIM per la linea di comando", per capire come fare le stesse cose con i comandi `spim`. Per avere più informazioni sull'uso di `PCspim` consultare invece *www.mkp.com/cod2e.htm*.

Per avviare `xspim`, digitare `xspim` sulla linea di comando del sistema (%):

```
% xspim
```

All'interno del sistema `xspim` potrebbe trovarsi in un'area di disco insolita, e quindi potrebbe essere necessario per prima cosa eseguire un comando per aggiungere tale directory al percorso di ricerca.

Quando parte, `xspim` visualizza sullo schermo una finestra di grandi dimensioni (vedere figura A.16), divisa in cinque pannelli:

- Il pannello in alto è detto *display dei registri*, e mostra il valore di tutti i registri nella CPU e nella FPU del MIPS; questo pannello è aggiornato ogni volta che il programma interrompe l'esecuzione.
- Il pannello sottostante contiene i *pulsanti di controllo* che consentono di introdurre i comandi di `xspim`; i pulsanti ed i relativi dettagli saranno descritti nel seguito.
- Il pannello successivo, detto *segmenti di testo*, riporta sia le istruzioni del programma utente che il codice di sistema caricato automaticamente quando `xspim` inizia l'esecuzione. Ciascuna istruzione è visualizzata su una linea del tipo:

  ```
  [0x00400000] 0x8fa40000 lw $4, 0($29) ; 89: lw $a0,
  0($sp)
  ```

  Il primo numero che compare sulla linea, fra parentesi quadre, è l'indirizzo esadecimale dell'istruzione. Il secondo numero è la codifica numerica dell'istruzione, di nuovo rappresentata come numero esadecimale; il terzo elemento è la descrizione mnemonica dell'istruzione. Tutto ciò che segue il

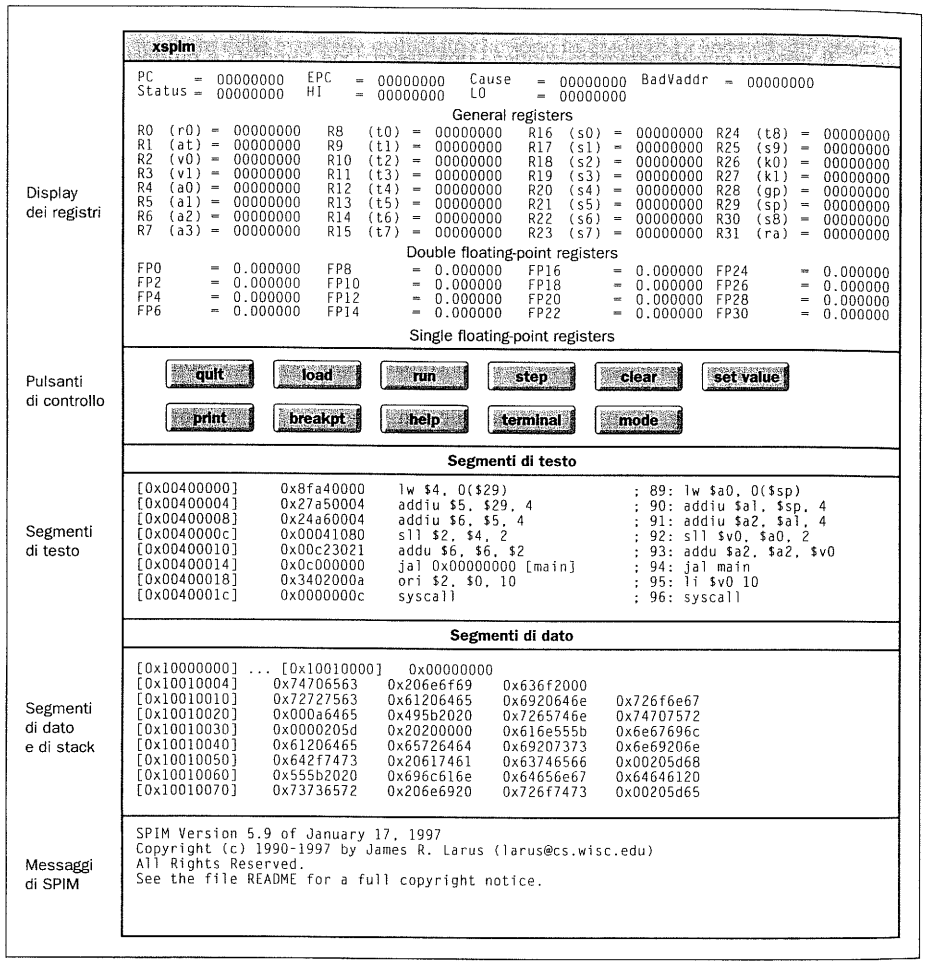


Figura A.16 L'interfaccia X-window di SPIM: `xspim`.

punto e virgola è l'effettiva linea di codice del programma utente che ha generato l'istruzione; il numero 89 è il numero della linea del file in cui si trova l'istruzione. Talvolta dopo il punto e virgola non c'è nulla: questo significa che l'istruzione è stata generata da SPIM come parte della traduzione di una pseudoistruzione.

- Il pannello successivo, chiamato *segmenti di dato e di stack*, mostra i dati che si trovano nella memoria del programma, e i dati nello stack del programma.
- Il pannello in basso è quello dei *messaggi di SPIM*, che `xspim` usa per visualizzare i messaggi; questo è il pannello in cui compaiono le segnalazioni d'errore.

Ora si vedrà come caricare ed eseguire un programma. La prima cosa da fare è premere il pulsante `load` (il secondo della prima linea di pulsanti) con il tasto sinistro del mouse; tale operazione dice a `xspim` di visualizzare una piccola finestra di dialogo che contiene un'area di testo e due o tre bottoni. Bisogna spostare il cursore del mouse sopra l'area di testo, scrivere il nome del file che contiene il codice assembler e scegliere il bottone `assembly file` all'interno della finestra. Se si vuole cambiare idea basta premere il bottone `abort command` e `xspim` farà scomparire la finestra di dialogo. Dopo la selezione di `assembly file`, `xspim` fa sparire la finestra di dialogo, carica il programma e ridisegna lo schermo per mostrare istruzioni e dati. Muovendo il mouse per posizionare il cursore sulla barra di scorrimento a sinistra del segmento di testo, e facendo clic con il tasto sinistro del mouse sulla parte chiara della barra, si può far scorrere il pannello del testo in modo da vedere tutte le istruzioni del programma.

Per eseguire il programma si deve scegliere il pulsante `run` dal pannello di controllo di `xspim`: viene visualizzata una finestra di dialogo con due aree di testo e due bottoni. Nella maggior parte dei casi le aree di testo contengono già i valori corretti per l'esecuzione del programma, e quindi si possono ignorare scegliendo semplicemente il bottone `ok`, che dice a `xspim` di eseguire il programma. Si noti che mentre il programma è in esecuzione `xspim` non visualizza il display dei registri, dato che i valori dei registri cambiano in continuazione: si può sempre sapere se `xspim` sta eseguendo un programma guardando questo pannello. Per interrompere l'esecuzione del programma bisogna assicurarsi che il cursore del mouse sia in una qualunque posizione all'interno della finestra di `xspim`, e premere il tasto control-C. Questo fa sì che `xspim` visualizzi una finestra di dialogo con due bottoni; prima di fare qualunque azione su questa finestra, si può osservare il contenuto dei registri e della memoria per capire ciò che sta facendo il programma. Quando questo è chiaro, si può continuare l'esecuzione scegliendo il bottone `continue` o interromperla con il bottone `abort command`.

Se il programma legge o scrive dati via terminale, `xspim` visualizza un'altra finestra, detta *console*: tutti i caratteri che il programma scrive appaiono sulla console, ed i dati di ingresso del programma devono essere introdotti in questa finestra.

Si supponga che il programma non si comporti come previsto: che cosa fare? SPIM ha due funzionalità che aiutano a trovare gli errori nei programmi; la prima, e forse la più utile, è l'esecuzione passo-passo, che permette di eseguire il programma un passo alla volta. Scegliendo il pulsante `step` appare un'altra finestra di dialogo, che contiene due aree di testo e tre bottoni. La prima area di testo chiede il numero di istruzioni da eseguire ogni volta che viene premuto il tasto del mouse; la maggior parte delle volte il valore suggerito 1 è una buona scelta. L'altra area di testo chiede gli argomenti da passare al programma all'inizio dell'esecuzione; la maggior parte delle volte si può ignorare quest'area, perché contiene un valore appropriato. Il bottone `step` esegue il programma per il numero di istruzioni definite nell'area di testo; se il numero è pari a 1, `xspim` esegue l'istruzione successiva del programma, aggiorna i display e restituisce il controllo all'utente. Il bottone `continue` interrompe la modalità passo-passo e prosegue l'esecuzione del programma; infine il bottone `abort command` interrompe l'esecuzione passo-passo e lascia il programma bloccato dov'era.

Che cosa fare se il programma è già in esecuzione da parecchio tempo quando compare l'errore? Una possibilità è eseguirlo passo-passo fino al punto in cui si è verificato l'errore, ma questo può richiedere molto tempo, ed è facile annoiarsi a tal punto da diventare disattenti e non ricordarsi di cercare l'errore al momento giusto.

Un'alternativa migliore è usare un *breakpoint* (o punto di interruzione), che dice a `xspim` di interrompere l'esecuzione immediatamente prima di una data istruzione. Per inserire un breakpoint basta selezionare il pulsante `breakpt` che compare nella seconda fila: il programma `xspim` fa comparire una finestra di dialogo con un'area di testo e diversi bottoni. Nell'area di testo va scritto l'indirizzo dell'istruzione in corrispondenza della quale si vuole interrompere l'esecuzione o, se l'istruzione ha un'etichetta globale, il nome di questa etichetta; i breakpoint sulle etichette sono un modo comodo di interrompere l'esecuzione alla prima istruzione di una procedura. Per inserire effettivamente il breakpoint, bisogna selezionare il bottone `add`; a questo punto si può eseguire il programma.

Quando SPIM sta per eseguire l'istruzione corrispondente ad un breakpoint, `xspim` visualizza una finestra di dialogo con l'indirizzo dell'istruzione e due bottoni; il bottone `continue` prosegue l'esecuzione del programma, mentre `abort command` la interrompe. Per cancellare un breakpoint è sufficiente scriverne l'indirizzo e scegliere il bottone `delete`; infine `list` dice a `xspim` di visualizzare (nel pannello in basso) l'elenco di tutti i breakpoint definiti.

L'esecuzione passo-passo e l'introduzione di breakpoint fanno sì che con ottime probabilità gli errori vengano trovati rapidamente; come correggerli, a questo punto? Bisogna ritornare all'applicazione con cui il programma è stato scritto e modificarlo. Per fare nuovamente eseguire il programma è necessaria un'altra copia di SPIM, che può essere ottenuta in due modi: uscendo da `xspim` con il pulsate `quit`, o cancellando lo schermo di `xspim` e ricaricando il programma. Per ricaricare il programma è *necessario* azzerare la memoria, così che i residui del vecchio programma non interferiscano con il nuovo. Per fare questo basta scegliere il pulsante `clear`: se si mantiene premuto il tasto sinistro del mouse compare un menu composto da due elementi; spostando il mouse sopra l'elemento `memory & register` e rilasciando il tasto si fa sì che `xspim` azzeri memoria e registri, e che il processore ritorni nello stesso stato in cui si trovava quando `xspim` era partito per la prima volta. Ora è possibile caricare il nuovo programma.

Gli altri pulsanti di `xspim` svolgono operazioni che occasionalmente possono essere utili; quando il lettore è a proprio agio con `xspim`, può consultare la descrizione riportata nel seguito per capire la funzione dei vari pulsanti, e per scoprire come può risparmiare tempo e fatica usandoli.

Opzioni SPIM per la linea di comando

Entrambe le versioni Unix di SPIM (`spim`, la versione per terminale alfanumerico, e `xspim`, la versione X-windows) accettano i seguenti comandi di linea:

`-bare`	Simula una macchina MIPS pura, senza pseudoistruzioni o modi di indirizzamento addizionali forniti dall'assemblatore. Implica `-quiet`.
`-asm`	Simula la macchina virtuale MIPS fornita dall'assemblatore. È l'opzione di default.
`-pseudo`	Consente al codice assembler in ingresso di contenere delle pseudoistruzioni. È l'opzione di default.
`-nopseudo`	Non consente al codice assembler in ingresso di contenere delle pseudoistruzioni.

`-notrap`	Non carica il gestore delle eccezioni ed il codice di inizializzazione (start-up) standard. Questo gestore gestisce le eccezioni: quando si verifica un'eccezione SPIM salta alla locazione 80000080_{esa}, che deve contenere il codice adatto a servire l'eccezione. In più questo file contiene del codice di inizializzazione che chiama la procedura `main`. Senza la procedura di inizializzazione SPIM inizia l'esecuzione all'istruzione etichettata `__start`.
`-trap`	Carica il gestore delle eccezioni ed il codice di inizializzazione standard. È l'opzione di default.
`-noquiet`	Visualizza un messaggio di errore quando si verifica un'eccezione. È l'opzione di default.
`-quiet`	Non visualizza un messaggio di errore quando si verifica un'eccezione.
`-nomapped_io`	Disabilita la funzionalità di I/O mappato in memoria (vedere Paragrafo A.8). È l'opzione di default.
`-mapped_io`	Abilita la funzionalità di I/O mappato in memoria (vedere Paragrafo A.8). I programmi che usano le `syscall` SPIM (vedere la parte relativa alle "Chiamate di Sistema", più avanti) per leggere da terminale *non* possono usare contemporaneamente la funzionalità di I/O mappato in memoria.
`-file`	Carica ed esegue il codice assembler contenuto nel file.
`-execute`	Carica ed esegue il codice contenuto nel file eseguibile MIPS *a.out*. Questo comando è disponibile solo quando SPIM è usato su un sistema che contiene un processore MIPS.
`-s<seg> dim`	Pone la dimensione iniziale del segmento di memoria *seg* pari a *dim* byte; i segmenti di memoria si chiamano `text`, `data`, `stack`, `ktext` e `kdata`. Il segmento `text` contiene le istruzioni del programma; il segmento `data` contiene i dati del programma; il segmento `stack` contiene lo stack durante l'esecuzione. Oltre ad eseguire un programma, SPIM esegue anche del codice di sistema che gestisce interruzioni ed eccezioni; tale codice risiede in una parte separata dello spazio di indirizzamento detta *kernel*. Il segmento `ktext` contiene le istruzioni di tale codice, mentre `kdata` ne contiene i dati; non esiste un segmento `kstack` dato che il codice di sistema usa lo stesso stack del programma. Ad esempio la coppia di argomenti `-sdata 2000000` inizializza la dimensione del segmento di dato a 2.000.000 byte.
`-l<seg> dim`	Pone come limite alla crescita del segmento di memoria *seg* un valore pari a *dim* byte. I segmenti di memoria che possono crescere sono `data`, `stack` e `kdata`.

L'interfaccia per terminale alfanumerico (`spim`)

La versione Unix più semplice di SPIM si chiama `spim`: non richiede un monitor grafico e può essere eseguita da qualsiasi terminale. Benché `spim` possa essere più

difficile da imparare, funziona in modo identico a `xspim` e fornisce le stesse funzionalità.

L'interfaccia per terminale alfanumerico di `spim` fornisce i seguenti comandi:

`exit`	Esce dal simulatore.
`read "file"`	Carica in SPIM *file*, un file scritto in linguaggio assembler; se il file è già stato letto precedentemente da SPIM il sistema deve essere reinizializzato (vedere sotto `reinizializze`) o le etichette globali avranno definizioni multiple.
`load "file"`	Sinonimo di `read`.
`execute "a.out"`	Carica in SPIM il file eseguibile MIPS *a.out*. Questo comando è disponibile solo se SPIM è usato su un sistema che contiene un processore MIPS.
`run <ind>`	Inizia l'esecuzione di un programma. Se è inserito l'indirizzo facoltativo *ind* il programma inizia a quell'indirizzo, in caso contrario inizia all'etichetta globale `__start`, che è solitamente il codice di inizializzazione standard che chiama la procedura all'etichetta globale `main`.
`step <N>`	Esegue *N* istruzioni del programma (il valore di default è 1). Visualizza le istruzioni nel momento in cui sono eseguite.
`continue`	Riprende l'esecuzione del programma (non passo-passo).
`print $N`	Visualizza il registro *N*.
`print $fN`	Visualizza il registro in virgola mobile *N*.
`print ind`	Visualizza il contenuto della memoria all'indirizzo *ind*.
`print_sym`	Visualizza nomi ed indirizzi delle etichette globali note a SPIM. Le etichette sono locali per default, e diventano globali solo quando sono dichiarate in una direttiva assembler `.globl` (vedere la parte sulla "Sintassi Assembler" nel Paragrafo A.10).
`reinizialize`	Azzera memoria e registri.
`breakpoint ind`	Inserisce un breakpoint all'indirizzo *ind*; *ind* può essere un indirizzo di memoria o un'etichetta simbolica.
`delete ind`	Cancella tutti i breakpoint all'indirizzo *ind*.
`list`	Elenca tutti i breakpoint.
`.`	Il resto della linea è un'istruzione assembler che viene scritta nella memoria.
`<a capo>`	Un a capo riesegue il comando precedente.
`?`	Visualizza un messaggio di aiuto.

La maggior parte dei comandi possono essere abbreviati con la parte iniziale non ambigua del nome (ad esempio `ex`, `re`, `l`, `ru`, `s`, `p`). Comandi più critici, come `reinizialize`, richiedono un'abbreviazione più lunga.

L'interfaccia X-window (`xspim`)

La guida "Introduzione a SPIM" spiega i comandi `xspim` più comuni; `xspim` possiede però altri comandi che all'occasione possono essere utili. Questo paragrafo fornisce un elenco completo dei comandi.

La versione X di SPIM, `xspim`, ha un'apparenza diversa ma opera nello stesso modo di `spim`. La finestra X-window ha cinque pannelli (vedere figura A.16); il pannello superiore mostra il contenuto dei registri, il cui valore viene costantemente aggiornato, a parte quando il programma è in esecuzione.

Il pannello successivo contiene i pulsanti che controllano il simulatore:

`quit`	Esce dal simulatore.
`load`	Carica in SPIM un file sorgente o eseguibile.
`run`	Avvia l'esecuzione del programma.
`step`	Avvia l'esecuzione passo-passo del programma.
`clear`	Reinizializza i registri o la memoria.
`set value`	Inserisce un valore in un registro o in una locazione di memoria.
`print`	Visualizza il valore di un registro o di una locazione di memoria.
`breakpt`	Inserisce o cancella un breakpoint, o elenca tutti i breakpoint.
`help`	Visualizza un messaggio di aiuto.
`terminal`	Porta in primo piano o nasconde la finestra della console.
`mode`	Definisce le modalità operative di SPIM.

I due pannelli successivi visualizzano la memoria; quello in alto mostra le istruzioni dei segmenti di testo dell'utente e del kernel. Queste istruzioni sono vere istruzioni MIPS e non pseudoistruzioni: SPIM traduce le pseudoistruzioni dell'assemblatore in un numero di istruzioni variabile da una a tre; ciascuna istruzione sorgente appare come commento della prima delle istruzioni in cui è tradotta. Le prime istruzioni del segmento di testo rappresentano il codice di inizializzazione di default (`___start`), che carica `argc` e `argv` nei registri ed invoca la procedura `main`. Il pannello in basso mostra i segmenti di dato e dello stack; entrambi i pannelli sono aggiornati durante l'esecuzione del programma.

Il pannello inferiore è utilizzato per visualizzare i messaggi di SPIM; esso non mostra le uscite del programma: quando un programma legge o scrive dei dati, il suo I/O è in una finestra a parte, chiamata *console*, che appare quando necessario.

Caratteristiche sorprendenti

Benché SPIM simuli fedelmente un calcolatore MIPS è pur sempre un simulatore, ed alcune caratteristiche non sono identiche a quelle di un vero calcolatore. La differenza più ovvia è che la temporizzazione delle istruzioni e del sistema di memoria non sono identiche: SPIM non simula le cache o la latenza della memoria, e non riflette fedelmente i ritardi dovuti alle operazioni in virgola mobile o alle moltiplicazioni e divisioni. Un'altra sorpresa (che però è tale e quale anche sui calcolatori reali) è rappresentata dal fatto che le pseudoistruzioni sono espanse in diverse istruzioni macchina: quando si esegue un programma passo-passo o si esamina il contenuto della memoria, le istruzioni visualizzate sono diverse da quelle del programma sorgente. La corrispondenza fra i due set di istruzioni è abbastanza semplice, dato che le istruzioni non vengono riorganizzate per riempire gli slot di ritardo.

L'ordinamento dei byte

I processori possono numerare i byte all'interno di una parola in modo tale che il byte che ha il numero più basso sia quello contenente i bit più significativi o quello contenente i meno significativi; la convenzione usata da un calcolatore è chiamata *ordinamento dei byte*. I processori MIPS possono operare sia con l'ordinamento *big-endian* sia con quello *little-endian*. Ad esempio in un calcolatore big-endian la direttiva `.byte 0, 1, 2, 3` risulterebbe in una parola di memoria contenente

Numero del byte			
0	1	2	3

mentre in un calcolatore little-endian la parola conterrebbe

Numero del byte			
0	1	2	3

SPIM opera con entrambi gli ordinamenti: il tipo di ordinamento è quello del calcolatore su cui viene eseguito il programma di simulazione. Ad esempio su una DECstation 3100 o su un Intel 80x86 SPIM segue l'ordinamento little-endian, mentre su un Macintosh o su una Sun SPARC segue l'ordinamento big-endian.

Chiamate di sistema

SPIM fornisce alcuni servizi analoghi a quelli di un sistema operativo grazie all'istruzione di chiamata di sistema (`syscall`). Per richiedere un servizio, il programma carica il codice della chiamata di sistema (vedere figura A.17) nel registro `$v0` e gli argomenti nei registri `$a0` - `$a3` (o in `$f12` nel caso di valori in virgola mobile). Le chiamate di sistema che forniscono un valore in uscita mettono il risultato nel registro `$v0` (o in `$f0` nel caso di valori in virgola mobile). Il seguente codice, ad esempio, stampa "`risposta = 5`":

```
        .data
str:
        .asciiz  "risposta = "
        .text
        li       $v0, 4      # codice della chiamata di sistema per
                             # print_string
        la       $a0, str    # indirizzo della stringa da stampare
        syscall              # stampa della stringa

        li       $v0, 1      # codice della chiamata di sistema per
                             # print_int
        li       $a0, 5      # numero intero da stampare
        syscall              # stampa del numero
```

La chiamata di sistema `print_int` stampa sulla console il numero intero che le viene passato come argomento; `print_float` stampa un numero in virgola mobile in singola precisione; `print_double` stampa un numero in virgola mobile in doppia precisione; `print_string`, infine, stampa sulla console la stringa che è stata passata come argomento terminandola con il carattere Null.

Servizio	Codice della chiamata di sistema	Argomenti	Risultato
`print_int`	1	`$a0` = intero	
`print_float`	2	`$f12` = virgola mobile	
`print_double`	3	`$f12` = doppia precisione	
`print_string`	4	`$a0` = stringa	
`read_int`	5		intero (in `$v0`)
`read_float`	6		virgola mobile (in `$f0`)
`read_double`	7		doppia precisione (in `$f0`)
`read_string`	8	`$a0` = buffer, `$a1` = lunghezza	
`sbrk`	9	`$a0` = quantità	indirizzo (in `$v0`)
`exit`	10		

Figura A.17 Servizi di sistema.

Le chiamate di sistema `read_int`, `read_float` e `read_double` leggono un'intera linea in ingresso fino al carattere a capo (incluso); i caratteri che seguono il numero sono ignorati. `read_string` ha la stessa sintassi della procedura di libreria Unix `fgets`: legge fino a n - 1 caratteri scrivendoli in un buffer e terminando la stringa con il carattere Null. Se ci sono meno di n - 1 caratteri sulla linea corrente, `read_string` li legge fino al carattere a capo (incluso) e termina la stringa con il carattere Null. *Attenzione:* i programmi che usano queste chiamate di sistema per leggere da terminale non devono usare l'I/O mappato in memoria (vedere il Paragrafo A.8).

`sbrk` infine restituisce il puntatore ad un blocco di memoria contenente n ulteriori byte, ed `exit` interrompe l'esecuzione di un programma.

A.10 Il linguaggio assembler MIPS R2000

Un processore MIPS è composto da un'unità di elaborazione per i numeri interi (la CPU) e da un insieme di coprocessori dedicati a compiti ausiliari o che operano su tipi di dati diversi, ad esempio in virgola mobile (figura A.18). SPIM simula due coprocessori. Il coprocessore 0 gestisce le eccezioni, le interruzioni ed il sistema di memoria virtuale; SPIM simula gran parte delle prime due funzionalità e trascura completamente i dettagli del sottosistema di memoria. Il coprocessore 1 è l'unità di elaborazione in virgola mobile; SPIM simula la maggior parte delle funzionalità di questa unità.

I modi di indirizzamento

Il MIPS è un'architettura di tipo load-store, il che significa che soltanto le istruzioni di load e di store possono accedere alla memoria; le operazioni di calcolo lavorano soltanto sui valori all'interno dei registri. L'hardware di base permette un unico modo di indirizzamento: `c(rx)`, che usa come indirizzo la somma del valore immediato `c` e del contenuto del registro `rx`. La macchina virtuale invece consente i seguenti modi di indirizzamento per le istruzioni di load e di store:

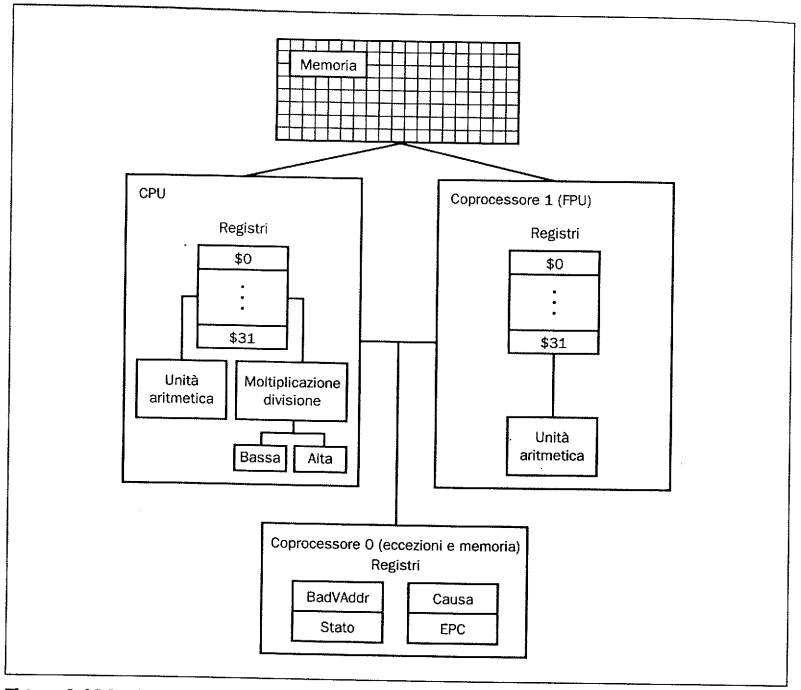


Figura A.18 La CPU e la FPU MIPS R2000.

Formato	Calcolo dell'indirizzo
(registro)	contenuto del registro
imm	immediato
imm (registro)	immediato + contenuto del registro
etichetta	indirizzo dell'etichetta
etichetta ± imm	indirizzo dell'etichetta + o - immediato
etichetta ± imm (registro)	indirizzo dell'etichetta + o - (immediato + contenuto del registro)

La maggior parte delle istruzioni di load e di store operano soltanto su dati allineati; una quantità è *allineata* se il suo indirizzo di memoria è un multiplo della sua dimensione espressa in byte. Un oggetto delle dimensioni di mezza parola (due byte) deve quindi essere memorizzato ad indirizzi pari, mentre un oggetto delle dimensioni di una parola (quattro byte) deve essere memorizzato ad indirizzi multipli di quattro. Il MIPS consente però ad alcune istruzioni di manipolare dati non allineati (`lwl`, `lwr`, `swl` e `swr`).

Approfondimento: L'assemblatore MIPS (e anche SPIM) traduce i modi di indirizzamento complessi in una o più istruzioni dedicate al calcolo dell'indirizzo che precedono quella di load o di store. Si supponga ad esempio che l'etichetta `tabella` si riferisca alla locazione di memoria 0x10000004, e che un programma contenga l'istruzione

```
ld    $a0, tabella + 4($a1)
```


L'assemblatore tradurrebbe questa istruzione in

```
lui   $at, 4096
addu  $at, $at, $a1
lw    $a0, 8($at)
```

La prima istruzione carica i bit più significativi dell'indirizzo dell'etichetta nel registro `$at`, riservato all'assemblatore; la seconda istruzione somma il contenuto del registro `$a1` all'indirizzo parziale dell'etichetta. Infine l'istruzione di load usa il modo di indirizzamento hardware per sommare i bit meno significativi dell'indirizzo dell'etichetta e l'offset dell'istruzione originale al valore contenuto nel registro `$at`.

La sintassi del linguaggio assembler

In un file assembler i commenti iniziano con il carattere #; tutto ciò che segue il # viene ignorato, fino a fine riga.

Gli identificatori sono sequenze di caratteri alfanumerici, caratteri di sottolineatura (_) e punti (.) che non iniziano con un numero. I codici operativi delle istruzioni sono riservati e *non possono* essere usati come identificatori. Le etichette si trovano all'inizio di una linea di codice e sono seguite dal carattere due punti, come ad esempio nel caso seguente:

```
            .data
elemento:   .word 1
            .text
            .globl main     # deve essere globale
main:       lw $t0, elemento
```

I numeri sono in base 10, se non diversamente specificato; se sono preceduti da *0x* sono interpretati come numeri esadecimali. Quindi 256 e 0x100 indicano lo stesso valore.

Le stringhe sono delimitate dalle virgolette (”); i caratteri speciali all'interno di una stringa seguono la convenzione del linguaggio C:

- a capo (newline) \n
- tabulatore (tab) \t
- virgolette \”

SPIM supporta un sottoinsieme delle direttive dell'assemblatore MIPS:

`.align n`	Allinea il dato successivo a blocchi di 2^n byte; ad esempio `.align 2` allinea alla parola (word) il valore successivo. `.align 0` elimina l'allineamento automatico delle direttive `.half`, `.word`, `.float` e `.double` fino a quando compare la successiva direttiva `.data` o `.kdata`.
`.ascii str`	Mette in memoria la stringa *str*, ma non aggiunge alla fine il carattere Null.
`.asciiz str`	Mette in memoria la stringa *str* terminata dal carattere Null.
`.byte b1, ..., bn`	Memorizza gli *n* valori in byte consecutivi della memoria.
`.data <ind>`	Gli elementi successivi sono memorizzati nel segmento

di dato. Se è presente l'argomento facoltativo *ind*, gli elementi successivi sono memorizzati a partire dall'indirizzo *ind*.

`.double d1, ..., dn` Memorizza gli *n* valori in virgola mobile in doppia precisione in locazioni consecutive della memoria.

`.extern simb dim` Dichiara che il dato memorizzato in *simb* è grande *dim* byte, e che è un'etichetta globale. Questa direttiva abilita l'assemblatore a memorizzare il dato nella porzione del segmento di dato accessibile efficientemente mediante il registro `$gp`.

`.float f1, ..., fn` Memorizza gli *n* valori in virgola mobile in singola precisione in locazioni consecutive della memoria.

`.global simb` Dichiara che l'etichetta *simb* è globale, e che ad essa è possibile fare riferimento da altri file.

`.half h1, ..., hn` Memorizza gli *n* valori su 16 bit in halfword (mezze parole) consecutive della memoria.

`.kdata <ind>` Gli elementi successivi sono memorizzati nel segmento di dato del kernel. Se è presente l'argomento facoltativo *ind*, gli elementi successivi sono memorizzati a partire dall'indirizzo *ind*.

`.ktext <ind>` Gli elementi successivi sono memorizzati nel segmento di testo del kernel. In SPIM questi elementi possono essere solo istruzioni o parole (vedere la direttiva `.word` poco oltre). Se è presente l'argomento facoltativo *ind*, gli elementi successivi sono memorizzati a partire dall'indirizzo *ind*.

`.set noat` e `.set at` La prima direttiva fa in modo che SPIM non produca segnalazioni di errore se istruzioni successive usano il registro `$at`, mentre la seconda riabilita la segnalazione. Dal momento che le pseudoistruzioni vengono espanse in codice che utilizza il registro `$at`, i programmatori devono stare molto attenti quando lasciano dei valori in questo registro.

`.space n` Alloca *n* byte nel segmento corrente (che nel caso di SPIM deve essere il segmento di dato).

`.text <ind>` Gli elementi successivi sono memorizzati nel segmento di testo dell'utente. In SPIM questi elementi possono essere solo istruzioni o parole (vedere la direttiva `.word` poco oltre). Se è presente l'argomento facoltativo *ind*, gli elementi successivi sono memorizzati a partire dall'indirizzo *ind*.

`.word w1, ..., wn` Memorizza gli *n* valori su 32 bit in parole consecutive della memoria.

SPIM non distingue le varie parti del segmento di dato (`.data`, `.rdata` e `.sdata`).

La codifica delle istruzioni MIPS

La figura A.19 mostra come un'istruzione venga codificata per ottenere un numero binario; ciascuna colonna contiene la codifica di un campo (gruppo di bit contigui) di un'istruzione. I numeri riportati sul margine sinistro sono i valori di un campo; ad esempio il codice operativo `j` ha il valore 2 nel campo codice operativo. Il testo in cima alle colonne dà il nome al campo e specifica quali bit di un'istruzione esso occupa; ad esempio il campo `op` è contenuto nei bit 26-31 di un'istruzione. Questo campo è in grado di codificare la maggior parte delle istruzioni, ma alcuni gruppi di istruzioni usano dei campi aggiuntivi per distinguere istruzioni correlate; ad esempio le diverse istruzioni in virgola mobile sono specificate dai bit 0-5. Le frecce che partono dalla prima colonna mostrano quali codici operativi usano i campi aggiuntivi.

Il formato delle istruzioni

Il resto della presente appendice descrive sia le istruzioni implementate dall'hardware MIPS che le pseudoistruzioni fornite dall'assemblatore MIPS. I due tipi di istruzioni possono essere distinti facilmente: nel caso delle istruzioni effettive sono descritti anche i campi della rappresentazione binaria; ad esempio in

Somma (con overflow)

`add rd, rs, rt`	0	rs	rt	rd	0	0x20
	6	5	5	5	5	6

l'istruzione `add` consiste di sei campi; la dimensione in bit di ciascun campo è rappresentata dal numero al di sotto del campo stesso. Questa istruzione inizia con 6 bit pari a 0; gli identificatori dei registri iniziano con una *r*, per cui il campo successivo è un identificatore di registro lungo 5 bit dal nome `rs`. Questo è lo stesso registro che compare come secondo argomento nella rappresentazione simbolica dell'assembler mostrata a sinistra della stessa riga. Un altro campo comunemente usato è imm_{16}, che è una costante (valore immediato) su 16 bit.

Le pseudoistruzioni seguono più o meno le stesse convenzioni, ma le informazioni sulla codifica delle istruzioni non compaiono. Ad esempio

Moltiplicazione (senza overflow)

`mul rdest, rsrc1, src2` *pseudoistruzione*

Nelle pseudoistruzioni `rdest` e `rsrc` sono registri, mentre `src2` può essere sia un registro che un valore immediato. In generale l'assemblatore e SPIM traducono una forma più generale di un'istruzione (ad esempio `add $v1, $a0, 0x55`) in una forma più specializzata (ad esempio `addi $v1, $a0, 0x55`).

Istruzioni aritmetiche e logiche

Valore assoluto

`abs rdest, rsrc` *pseudoistruzione*

Memorizza il valore assoluto del registro `rsrc` nel registro `rdest`.

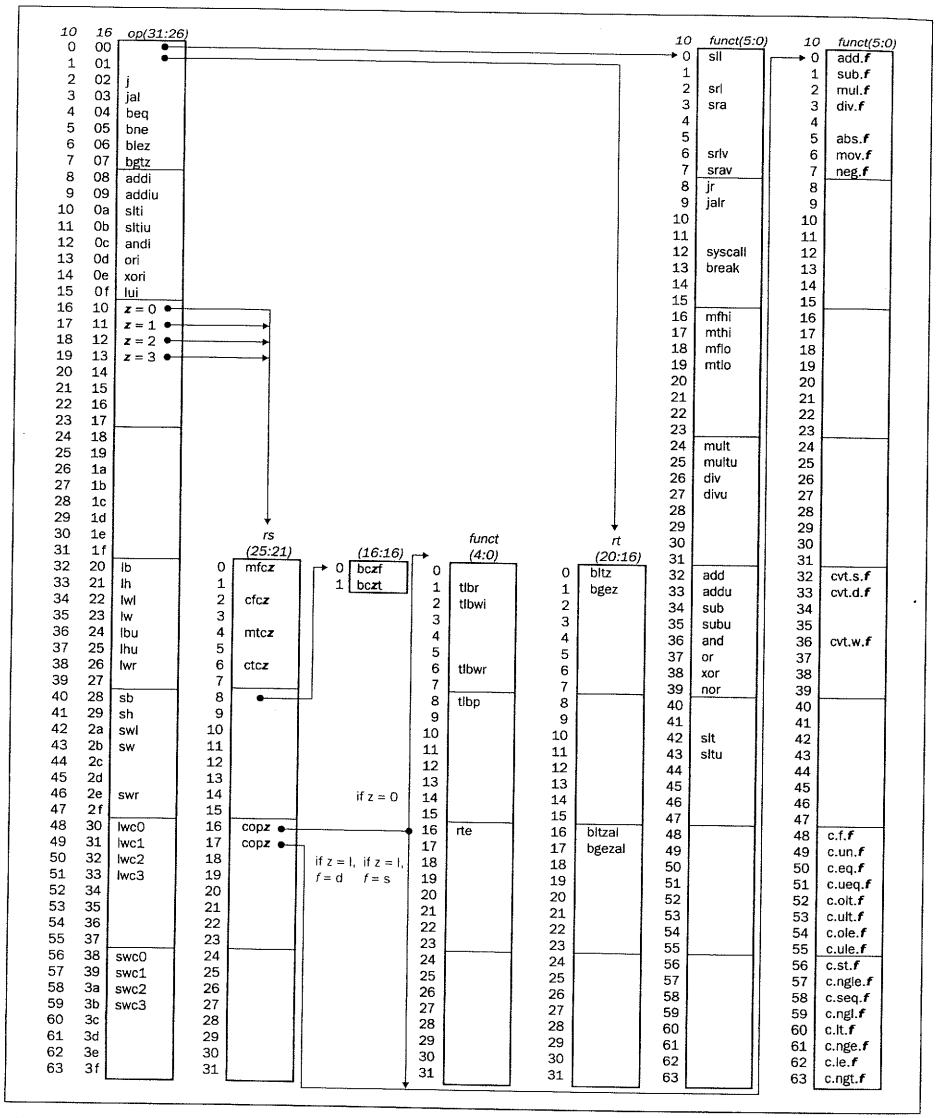


Figura A.19 La mappa dei codici operativi MIPS. I valori di ciascun campo sono mostrati alla sua sinistra. La prima colonna mostra i valori in base 10, e la seconda in base 16, del campo op (bit da 31 a 26) riportato nella terza colonna; il campo op specifica completamente le operazioni MIPS eccetto che per 6 valori: 0, 1, 16, 17, 18 e 19: queste operazioni sono determinate da altri campi, identificati dalle frecce. L'ultimo campo (funct) usa "*f*" per indicare "s" se rs = 16 e op = 17, o "d" se rs = 17 e op = 17. Il secondo campo (rs) usa "z" per indicare "0", "1", "2" o "3" se op = 16, 17, 18 o 19, rispettivamente. Se rs = 16 l'operazione è specificata altrove: se $z = 0$ le operazioni sono specificate dal quarto campo (bit da 4 a 0), mentre se $z = 1$ le operazioni si trovano nell'ultimo campo con f = s. Se rs = 17 e $z = 1$ le operazioni sono nell'ultimo campo con f = d.

Somma (con overflow)

```
add rd, rs, rt
```

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

Somma (senza overflow)

```
addu rd, rs, rt
```

0	rs	rt	rd	0	0x21
6	5	5	5	5	6

Memorizza la somma dei registri `rs` e `rt` nel registro `rd`.

Somma immediata (con overflow)

```
addi rt, rs, imm
```

8	rs	rt	imm
6	5	5	16

Somma immediata (senza overflow)

```
addiu rt, rs, imm
```

9	rs	rt	imm
6	5	5	16

Memorizza la somma del registro `rs` e della costante *imm* (estendendo il segno) nel registro `rt`.

AND

```
and rd, rs, rt
```

0	rs	rt	rd	0	0x24
6	5	5	5	5	6

Memorizza l'AND logico dei registri `rs` e `rt` nel registro `rd`.

AND immediato

```
andi rt, rs, imm
```

0xc	rs	rt	imm
6	5	5	16

Memorizza l'AND logico del registro `rs` e della costante *imm* (estendendola con degli zeri) nel registro `rt`.

Divisione (con overflow)

```
div rs, rt
```

0	rs	rt	0	0x1a
6	5	5	10	6

Divisione (senza overflow)

```
divu rs, rt
```

0	rs	rt	0	0x1b
6	5	5	10	6

Divide il contenuto del registro `rs` per il contenuto del registro `rt`; lascia il quoziente nel registro `lo` ed il resto nel registro `hi`. Si noti che se uno degli operandi è negativo l'architettura MIPS non specifica il resto, che dipende dalla convenzione usata dal calcolatore su cui SPIM lavora.

Divisione (con overflow)

`div rdest, rsrc1, src2` *pseudoistruzione*

Divisione (senza overflow)

`divu rdest, rsrc1, src2` *pseudoistruzione*

Memorizza il quoziente della divisione di `rsrc1` per `src2` nel registro `rdest`.

Moltiplicazione

`mult rs, rt`

0	rs	rt	0	0x18
6	5	5	10	6

Moltiplicazione senza segno

`multu rs, rt`

0	rs	rt	0	0x19
6	5	5	10	6

Moltiplica i registri `rs` e `rt`; lascia la parola meno significativa del prodotto nel registro `lo` e quella più significativa nel registro `hi`.

Moltiplicazione (con overflow)

`mul rdest, rsrc1, src2` *pseudoistruzione*

Moltiplicazione (senza overflow)

`mulo rdest, rsrc1, src2` *pseudoistruzione*

Moltiplicazione senza segno (con overflow)

`mulou rdest, rsrc1, src2` *pseudoistruzione*

Memorizza il prodotto dei registri `rsrc1` e `src2` nel registro `rdest`.

Negazione (con overflow)

`neg rdest, rsrc` *pseudoistruzione*

Negazione (senza overflow)

`negu rdest, rsrc` *pseudoistruzione*

Cambia il segno del registro `rsrc` e lo memorizza nel registro `rdest`.

NOR

`nor rd, rs, rt`

0	rs	rt	rd	0	0x27
6	5	5	5	5	6

Memorizza il NOR logico dei registri `rs` e `rt` nel registro `rd`.

NOT

`not rdest, rsrc` *pseudoistruzione*

Memorizza la negazione logica (complemento a 1) del registro `rsrc` nel registro `rdest`.

OR

`or rd, rs, rt`

0	rs	rt	rd	0	0x25
6	5	5	5	5	6

Memorizza l'OR logico dei registri `rs` e `rt` nel registro `rd`.

OR immediato

`ori rt, rs, imm`

0xd	rs	rt	imm
6	5	5	16

Memorizza l'OR logico del registro `rs` e della costante *imm* (estendendola con degli zeri) nel registro `rt`.

Resto

`rem rdest, rsrc1, rsrc2` *pseudoistruzione*

Resto senza segno

`remu rdest, rsrc1, rsrc2` *pseudoistruzione*

Memorizza il resto della divisione di `rsrc1` per `rsrc2` nel registro `rdest`. Si noti che se uno degli operandi è negativo l'architettura MIPS non specifica il resto, che dipende dalla convenzione usata dal calcolatore su cui SPIM lavora.

Scalamento (shift) logico a sinistra

`sll rd, rt, shamt`

0	rs	rt	rd	shamt	0
6	5	5	5	5	6

Scalamento (shift) logico a sinistra di entità variabile

`sllv rd, rt, rs`

0	rs	rt	rd	0	4
6	5	5	5	5	6

Scalamento (shift) aritmetico a destra

`sra rd, rt, shamt`

0	rs	rt	rd	shamt	3
6	5	5	5	5	6

Scalamento (shift) aritmetico a destra di entità variabile

`srav rd, rt, rs`

0	rs	rt	rd	0	7
6	5	5	5	5	6

Scalamento (shift) logico a destra

`srl rd, rt, shamt`

0	rs	rt	rd	shamt	2
6	5	5	5	5	6

Scalamento (shift) logico a destra di entità variabile

`srlv rd, rt, rs`

0	rs	rt	rd	0	6
6	5	5	5	5	6

Scala a sinistra (a destra) il registro `rt` di un numero di posizioni indicato dalla costante `shamt` o dal registro `rs`, e memorizza il risultato nel registro `rd`. Si noti che l'argomento `rs` è ignorato nel caso di `sll`, `sra` e `srl`.

Rotazione a sinistra

`rol rdest, rsrc1, rsrc2` *pseudoistruzione*

Rotazione a destra

`ror rdest, rsrc1, rsrc2` *pseudoistruzione*

Esegue una rotazione a sinistra (a destra) del registro `rsrc1` di un numero di posizioni indicato dal registro `rsrc2`, e memorizza il risultato nel registro `rdest`.

Sottrazione (con overflow)

`sub rd, rs, rt`

0	rs	rt	rd	0	0x22
6	5	5	5	5	6

Sottrazione (senza overflow)

`subu rd, rs, rt`

0	rs	rt	rd	0	0x23
6	5	5	5	5	6

Memorizza la differenza fra i registri `rs` e `rt` nel registro `rd`.

OR esclusivo

`xor rd, rs, rt`

0	rs	rt	rd	0	0x26
6	5	5	5	5	6

Memorizza l'OR esclusivo logico (XOR) dei registri `rs` e `rt` nel registro `rd`.

XOR immediato

`xori rt, rs, imm`

0xe	rs	rt	imm
6	5	5	16

Memorizza l'OR esclusivo logico (XOR) del registro `rs` e della costante *imm* (estendendola con degli zeri) nel registro `rt`.

Istruzioni per la manipolazione di costanti

Load upper immediato

`lui rt, imm`

0xf	0	rt	imm
6	5	5	16

Carica la costante `imm`, da 16 bit, nella mezza parola più significativa del registro `rt`; i bit meno significativi del registro sono posti a 0.

Load immediato

`li rdest, imm` *pseudoistruzione*

Copia la costante `imm` nel registro `rdest`.

Istruzioni per il confronto

Set less than

`slt rd, rs, rt`

0	rs	rt	rd	0	0x2a
6	5	5	5	5	6

Set less than senza segno

`sltu rd, rs, rt`

0	rs	rt	rd	0	0x2b
6	5	5	5	5	6

Mette a 1 il registro `rd` se il registro `rs` è minore del registro `rt`; in caso contrario lo mette a 0.

Set less than immediato

`slti rd, rs, imm`

0xa	rs	rd	imm
6	5	5	16

Set less than immediato senza segno

`sltiu rd, rs, imm`

0xb	rs	rd	imm
6	5	5	16

Mette a 1 il registro `rd` se il registro `rs` è minore della costante *imm* (estendendo il segno); in caso contrario lo mette a 0.

Set equal

`seq rdest, rsrc1, rsrc2` *pseudoistruzione*

Mette a 1 il registro `rdest` se il registro `rsrc1` è uguale al registro `rsrc2`; in caso contrario lo mette a 0.

Set greater than equal

`sge rdest, rsrc1, rsrc2` *pseudoistruzione*

Set greater than equal senza segno

`sgeu rdest, rsrc1, rsrc2` *pseudoistruzione*

Mette a 1 il registro `rdest` se il registro `rsrc1` è maggiore o uguale al registro `rsrc2`; in caso contrario lo mette a 0.

Set greater than

`sgt rdest, rsrc1, rsrc2` *pseudoistruzione*

Set greater than senza segno

`sgtu rdest, rsrc1, rsrc2` *pseudoistruzione*

Mette a 1 il registro `rdest` se il registro `rsrc1` è maggiore del registro `rsrc2`; in caso contrario lo mette a 0.

Set less than equal

`sle rdest, rsrc1, rsrc2` *pseudoistruzione*

Set less than equal senza segno

`sleu rdest, rsrc1, rsrc2` *pseudoistruzione*

Mette a 1 il registro `rdest` se il registro `rsrc1` è minore o uguale al registro `rsrc2`; in caso contrario lo mette a 0.

Set not equal

`sne rdest, rsrc1, rsrc2` *pseudoistruzione*

Mette a 1 il registro `rdest` se il registro `rsrc1` è diverso dal registro `rsrc2`; in caso contrario lo mette a 0.

Istruzioni di salto condizionato

Le istruzioni di salto condizionato usano un campo *offset* di 16 bit (con segno); possono quindi saltare ad una locazione posta $2^{15} - 1$ *istruzioni* (non byte) in avanti, o 2^{15} istruzioni all'indietro. L'istruzione *jump* (salto incondizionato) contiene invece un campo di 26 bit per specificare l'indirizzo.

Nella descrizione successiva gli offset non sono specificati, e le istruzioni saltano ad un'etichetta; questa è la forma utilizzata nella maggior parte dei programmi assembler, essendo difficile calcolare la distanza fra le istruzioni dal momento che le pseudoistruzioni sono espanse in più istruzioni.

Branch

`b etichetta` *pseudoistruzione*

Esegue un salto incondizionato all'istruzione identificata dall'etichetta.

Branch coprocessor z vero

`bczt etichetta`

0x1z	8	1	offset
6	5	5	16

Branch coprocessor z falso

`bczf etichetta`

0x1z	8	0	offset
6	5	5	16

Esegue un salto condizionato pari al numero di istruzioni specificato dall'offset se il flag di condizione del coprocessore z è vero (falso); z può essere 0, 1, 2 o 3. L'unità di elaborazione in virgola mobile corrisponde a z = 1.

Branch on equal

`beq rs, rt, etichetta`

4	rs	rt	offset
6	5	5	16

Esegue un salto condizionato pari al numero di istruzioni specificato dall'offset se il registro `rs` è uguale al registro `rt`.

Branch on greater than equal zero

`bgez rs, etichetta`

1	rs	1	offset
6	5	5	16

Esegue un salto condizionato pari al numero di istruzioni specificato dall'offset se il registro `rs` è maggiore o uguale a 0.

Branch on greater than equal zero and link

`bgezal rs, etichetta`

1	rs	0x11	offset
6	5	5	16

Esegue un salto condizionato pari al numero di istruzioni specificato dall'offset se il registro `rs` è maggiore o uguale a 0; salva l'indirizzo dell'istruzione successiva nel registro 31.

Branch on greater than zero

`bgtz rs, etichetta`

7	rs	0	offset
6	5	5	16

Esegue un salto condizionato pari al numero di istruzioni specificato dall'offset se il registro `rs` è maggiore di 0.

Branch on less than equal zero

`blez rs, etichetta`

6	rs	0	offset
6	5	5	16

Esegue un salto condizionato pari al numero di istruzioni specificato dall'offset se il registro `rs` è minore o uguale a 0.

Branch on less than and link

`bltzal rs, etichetta`

1	rs	0x10	offset
6	5	5	16

Esegue un salto condizionato pari al numero di istruzioni specificato dall'offset se il registro `rs` è minore di 0; salva l'indirizzo dell'istruzione successiva nel registro 31.

Branch on less than zero

`bltz rs, etichetta`

1	rs	0	offset
6	5	5	16

Esegue un salto condizionato pari al numero di istruzioni specificato dall'offset se il registro `rs` è minore di 0.

Branch on not equal

`bne rs, rt, etichetta`

5	rs	rt	offset
6	5	5	16

Esegue un salto condizionato pari al numero di istruzioni specificato dall'offset se il registro `rs` è diverso dal registro `rt`.

Branch on equal zero

`beqz rsrc, etichetta` *pseudoistruzione*

Esegue un salto condizionato all'istruzione identificata dall'etichetta se il registro `rsrc` è uguale a 0.

Branch on greater than equal

`bge rsrc1, rsrc2, etichetta` *pseudoistruzione*

Branch on greater than equal senza segno

`bgeu rsrc1, rsrc2, etichetta` *pseudoistruzione*

Esegue un salto condizionato all'istruzione identificata dall'etichetta se il registro `rsrc1` è maggiore o uguale al registro `rsrc2`.

Branch on greater than

`bgt rsrc1, rsrc2, etichetta` *pseudoistruzione*

Branch on greater than senza segno

`bgtu rsrc1, src2, etichetta` *pseudoistruzione*

Esegue un salto condizionato all'istruzione identificata dall'etichetta se il registro `rsrc1` è maggiore del registro `rsrc2`.

Branch on less than equal

`ble rsrc1, rsrc2, etichetta` *pseudoistruzione*

Branch on less than equal senza segno

`bleu rsrc1, rsrc2, etichetta` *pseudoistruzione*

Esegue un salto condizionato all'istruzione identificata dall'etichetta se il registro `rsrc1` è minore o uguale al registro `rsrc2`.

Branch on less than

`blt rsrc1, rsrc2, etichetta` *pseudoistruzione*

Branch on less than senza segno

`bltu rsrc1, rsrc2, etichetta` *pseudoistruzione*

Esegue un salto condizionato all'istruzione identificata dall'etichetta se il registro `rsrc1` è minore del registro `rsrc2`.

Branch on not equal zero

`bnez rsrc, etichetta` *pseudoistruzione*

Esegue un salto condizionato all'istruzione identificata dall'etichetta se il registro `rsrc` è diverso da 0.

Istruzioni di salto incondizionato

Jump

`j destinazione`

2	destinazione
6	26

Esegue un salto incondizionato all'istruzione destinazione.

Jump and link

`jal destinazione`

3	destinazione
6	26

Esegue un salto incondizionato all'istruzione destinazione; salva l'indirizzo dell'istruzione successiva nel registro 31.

Jump and link register

`jalr rs, rd`

0	rs	0	rd	0	9
6	5	5	5	5	6

Esegue un salto incondizionato all'istruzione il cui indirizzo si trova nel registro `rs`; salva l'indirizzo dell'istruzione successiva nel registro `rd` (se omesso viene usato il registro 31).

Jump register

`jr rs`

0	rs	0	8
6	5	15	6

Esegue un salto incondizionato all'istruzione il cui indirizzo si trova nel registro `rs`.

Istruzioni di load

Load indirizzo

`la rdest, indirizzo` *pseudoistruzione*

Carica nel registro `rdest` l'*indirizzo* calcolato (non il contenuto della locazione).

Load byte

`lb rt, indirizzo`

0x20	rs	rt	offset
6	5	5	16

Load byte senza segno

`lbu rt, indirizzo`

0x24	rs	rt	offset
6	5	5	16

Carica nel registro `rt` il byte che si trova all'*indirizzo*; il bit di segno del byte è esteso da `lb` ma non da `lbu`.

Load halfword

`lh rt, indirizzo`

0x21	rs	rt	offset
6	5	5	16

Load halfword senza segno

`lhu rt, indirizzo`

0x25	rs	rt	offset
6	5	5	16

Carica nel registro `rt` il valore di 16 bit (halfword) che si trova all'*indirizzo*; il bit di segno della halfword è esteso da `lh` ma non da `lhu`.

Load word

`lw rt, indirizzo`

0x23	rs	rt	offset
6	5	5	16

Carica nel registro `rt` il valore di 32 bit (parola, o word) che si trova all'*indirizzo*.

Load word coprocessor

`lwcz rt, indirizzo`

0x3z	rs	rt	offset
6	5	5	16

Carica nel registro `rt` del coprocessore z (0-3) la parola che si trova all'*indirizzo*; l'unità di elaborazione in virgola mobile corrisponde a z = 1.

Load word sinistra

`lwl rt, indirizzo`

0x26	rs	rt	offset
6	5	5	16

Load word destra

`lwr rt, indirizzo`

0x23	rs	rt	offset
6	5	5	16

Carica nel registro `rt` il byte a sinistra (a destra) della parola che si trova all'*indirizzo* (anche se non allineata).

Load doubleword

`ld rdest, indirizzo` *pseudoistruzione*

Carica nei registri `rdest` e `rdest+1` il valore di 64 bit (doppia parola, o doubleword) che si trova all'*indirizzo*.

Load halfword non allineata

`ulh rdest, indirizzo` *pseudoistruzione*

Load halfword non allineata senza segno

`ulhu rdest, indirizzo` *pseudoistruzione*

Carica nel registro `rdest` il valore di 16 bit (halfword) che si trova all'*indirizzo* (anche se non allineato); il bit di segno della halfword è esteso da `ulh` ma non da `ulhu`.

Load word non allineata

`ulw rdest, indirizzo` *pseudoistruzione*

Carica nel registro `rdest` il valore di 32 bit (parola, o word) che si trova all'*indirizzo* (anche se non allineata).

Istruzioni di store

Store byte

`sb rt, indirizzo`

0x28	rs	rt	offset
6	5	5	16

Memorizza in *indirizzo* il byte meno significativo del registro `rt`.

Store halfword

`sh rt, indirizzo`

0x29	rs	rt	offset
6	5	5	16

Memorizza in *indirizzo* la mezza parola (halfword) meno significativa del registro `rt`.

Store word

`sw rt, indirizzo`

0x2b	rs	rt	offset
6	5	5	16

Memorizza in *indirizzo* la parola (word) contenuta nel registro `rt`.

Store word coprocessor

`swcz rt, indirizzo`

0x32	rs	rt	offset
6	5	5	16

Memorizza in *indirizzo* la parola (word) contenuta nel registro `rt` del coprocessore *z*; l'unità di elaborazione in virgola mobile corrisponde a *z* = 1.

Store word sinistra

`swl rt, indirizzo`

0x2a	rs	rt	offset
6	5	5	16

Store word destra

`swr rt, indirizzo`

0x2e	rs	rt	offset
6	5	5	16

Memorizza in *indirizzo* (anche se non allineato) i byte a sinistra (destra) del registro `rt`.

Store doubleword

`sd rsrc, indirizzo` *pseudoistruzione*

Memorizza in *indirizzo* il valore di 64 bit (doppia parola, o doubleword) contenuto nei registri `rsrc` e `rsrc+1`.

Store halfword non allineata

`ush rsrc, indirizzo` *pseudoistruzione*

Memorizza in *indirizzo* (anche se non allineato) il valore di 16 bit (halfword) contenuto nel registro `rsrc`.

Store word non allineata

`usw rsrc, indirizzo` *pseudoistruzione*

Memorizza in *indirizzo* (anche se non allineato) la parola (word) contenuta nel registro `rsrc`.

Istruzioni per il trasferimento dei dati

Move

`move rdest, rsrc` *pseudoistruzione*

Copia il contenuto del registro `rsrc` nel registro `rdest`.

Move from hi

`mfhi rd`

0	0	rd	0	0x10
6	10	5	5	6

Move from lo

`mflo rd`

0	0	rd	0	0x12
6	10	5	5	6

L'unità di moltiplicazione e divisione produce il risultato in due registri aggiuntivi, `hi` e `lo`; queste istruzioni copiano i valori da e verso tali registri. Le pseudoistruzioni di moltiplicazione, divisione e resto che fanno sì che l'unità sembri lavorare con i normali registri in realtà ricopiano i risultati alla fine del calcolo.

Copia il contenuto del registro `hi` (`lo`) nel registro `rd`.

Move to hi

`mthi rs`

0	rs	0	0x11
6	5	15	6

Move to lo

`mtlo rs`

0	rs	0	0x13
6	5	15	6

Copia il contenuto del registro `rs` nel registro `hi` (`lo`).

Move from coprocessor *z*

`mfcz rt, rd`

0x1*z*	0	rt	rd	0
6	5	5	5	11

I coprocessori hanno un proprio insieme di registri; queste istruzioni copiano i valori fra tali registri ed i registri della CPU.

Copia il contenuto del registro `rd` del coprocessore *z* nel registro della CPU `rt`; l'unità di elaborazione in virgola mobile corrisponde a *z* = 1.

Move double from coprocessor 1

`mfc1.d rdest, frsrc1` *pseudoistruzione*

Copia il contenuto dei registri in virgola mobile `frsrc1` e `frsrc1+1` nei registri della CPU `rdest` e `rdest+1`.

Move to coprocessor *z*

`mtcz rd, rt`

0x1*z*	4	rt	rd	0
6	5	5	5	11

Copia il contenuto del registro della CPU `rt` nel registro `rd` del coprocessore *z*.

Istruzioni in virgola mobile

Il MIPS ha un coprocessore per l'elaborazione dei numeri in virgola mobile (il coprocessore 1) che può operare sia su numeri in singola precisione (32 bit) che su numeri in doppia precisione (64 bit). Questo coprocessore ha i propri registri, numerati da `$f0` a `$f31`; dato che questi registri sono lunghi solo 32 bit ne occorre una coppia per contenere dati in doppia precisione, per cui soltanto i registri in virgola mobile con numero pari possono contenere numeri di questo tipo.

I valori vengono spostati dentro o fuori da questi registri, una parola (32 bit) alla volta, con le istruzioni `lwc1`, `swc1`, `mtc1` e `mfc1` descritte in precedenza, o con le pseudoistruzioni `l.s`, `l.d`, `s.s` e `s.d` descritte più avanti. Il flag aggiornato dalle operazioni di confronto fra numeri in virgola mobile è letto dalla CPU con le istruzioni `bc1t` e `bc1f`.

Nel caso delle istruzioni vere e proprie riportate nel seguito, i bit 21-26 valgono 0 per la singola precisione, e 1 per la doppia precisione; nel caso delle pseudoistruzioni `fdest` è un registro in virgola mobile (ad esempio `$f2`).

Valore assoluto di numeri in virgola mobile in doppia precisione

`abs.d fd, fs`

0x11	1	0	fs	fd	5
6	5	5	5	5	6

Valore assoluto di numeri in virgola mobile in singola precisione

`abs.s fd, fs`

0x11	0	0	fs	fd	5
6	5	5	5	5	6

Calcola il valore assoluto del numero in virgola mobile in doppia (singola) precisione contenuto nel registro `fs`, e mette il risultato nel registro `fd`.

Somma di numeri in virgola mobile in doppia precisione

`add.d fd, fs, ft`

0x11	1	ft	fs	fd	0
6	5	5	5	5	6

Somma di numeri in virgola mobile in singola precisione

`add.s fd, fs, ft`

0x11	0	ft	fs	fd	0
6	5	5	5	5	6

Calcola la somma dei numeri in virgola mobile in doppia (singola) precisione contenuti nei registri `fs` e `ft`, e mette il risultato nel registro `fd`.

Compare equal in doppia precisione

`c.eq.d fs, ft`

0x11	1	ft	fs	fd	FC	2
6	5	5	5	5	2	4

Compare equal in singola precisione

`c.eq.s fs, ft`

0x11	0	ft	fs	fd	FC	2
6	5	5	5	5	2	4

Confronta il numero in virgola mobile in doppia (singola) precisione contenuto nel registro `fs` con quello contenuto nel registro `ft`, e pone a vero il flag di condizione se sono uguali. Per controllare il valore di questo flag si usano le istruzioni `bc1t` o `bc1f`.

Compare less than equal in doppia precisione

`c.le.d fs, ft`

0x11	1	ft	fs	0	FC	2
6	5	5	5	5	2	4

Compare less than equal in singola precisione

`c.le.s fs, ft`

0x11	0	ft	fs	0	FC	2
6	5	5	5	5	2	4

Confronta il numero in virgola mobile in doppia (singola) precisione contenuto nel registro `fs` con quello contenuto nel registro `ft`, e pone a vero il flag di condizione se il primo è minore o uguale del secondo. Per controllare il valore di questo flag si usano le istruzioni `bc1t` o `bc1f`.

Compare less than in doppia precisione

`c.lt.d fs, ft`

0x11	1	ft	fs	0	FC	0xc
6	5	5	5	5	2	4

Compare less than in singola precisione

`c.lt.s fs, ft`

0x11	0	ft	fs	0	FC	0xc
6	5	5	5	5	2	4

Confronta il numero in virgola mobile in doppia (singola) precisione contenuto nel registro `fs` con quello contenuto nel registro `ft`, e pone a vero il flag di condizione se il primo è minore del secondo. Per controllare il valore di questo flag si usano le istruzioni `bc1t` o `bc1f`.

Conversione da singola precisione a doppia precisione

`cvt.d.s fd, fs`

0x11	1	0	fs	fd	0x21
6	5	5	5	5	6

Conversione da intero a doppia precisione

`cvt.d.w fd, fs`

0x11	0	0	fs	fd	0x21
6	5	5	5	5	6

Converte il numero in virgola mobile in singola precisione o il numero intero contenuto nel registro `fs` in un numero in doppia precisione, che memorizza nel registro `fd`.

Conversione da doppia precisione a singola precisione

`cvt.s.d fd, fs`

0x11	1	0	fs	fd	0x20
6	5	5	5	5	6

Conversione da intero a singola precisione

`cvt.s.w fd, fs`

0x11	0	0	fs	fd	0x20
6	5	5	5	5	6

Converte il numero in virgola mobile in doppia precisione o il numero intero contenuto nel registro `fs` in un numero in singola precisione, che memorizza nel registro `fd`.

Conversione da doppia precisione ad intero

`cvt.w.d fd, fs`

0x11	1	0	fs	fd	0x24
6	5	5	5	5	6

Conversione da singola precisione ad intero

`cvt.w.s fd, fs`

0x11	0	0	fs	fd	0x24
6	5	5	5	5	6

Converte il numero in virgola mobile in doppia o in singola precisione contenuto nel registro `fs` in un numero intero, che memorizza nel registro `fd`.

Divisione di numeri in virgola mobile in doppia precisione

`div.d fd, fs, ft`

0x11	1	ft	fs	fd	3
6	5	5	5	5	6

Divisione di numeri in virgola mobile in singola precisione

`div.s fd, fs, ft`

0x11	0	ft	fs	fd	3
6	5	5	5	5	6

Calcola il quoziente dei numeri in virgola mobile in doppia (singola) precisione contenuti nei registri `fs` e `ft`, e mette il risultato nel registro `fd`.

Load di numeri in virgola mobile in doppia precisione

`l.d fdest, indirizzo` *pseudoistruzione*

Load di numeri in virgola mobile in singola precisione

`l.s fdest, indirizzo` *pseudoistruzione*

Carica nel registro `fdest` il numero in virgola mobile in doppia (singola) precisione che si trova all'*indirizzo*.

Move di numeri in virgola mobile in doppia precisione

`mov.d fd, fs`

0x11	1	0	fs	fd	6
6	5	5	5	5	6

Move di numeri in virgola mobile in singola precisione

`mov.s fd, fs`

0x11	0	0	fs	fd	6
6	5	5	5	5	6

Copia il numero in virgola mobile in doppia (singola) precisione contenuto nel registro `fs` nel registro `fd`.

Moltiplicazione di numeri in virgola mobile in doppia precisione

`mul.d fd, fs, ft`

0x11	1	ft	fs	fd	2
6	5	5	5	5	6

Moltiplicazione di numeri in virgola mobile in singola precisione

`mul.s fd, fs, ft`

0x11	0	ft	fs	fd	2
6	5	5	5	5	6

Calcola il prodotto dei numeri in virgola mobile in doppia (singola) precisione contenuti nei registri `fs` e `ft`, e mette il risultato nel registro `fd`.

Negazione in doppia precisione

`neg.d fd, fs`

0x11	1	0	fs	fd	7
6	5	5	5	5	6

Negazione in singola precisione

`neg.s fd, fs`

0x11	0	0	fs	fd	7
6	5	5	5	5	6

Cambia il segno del numero in virgola mobile in doppia (singola) precisione contenuto nel registro `fs` e lo memorizza nel registro `fd`.

Store di numeri in virgola mobile in doppia precisione

`s.d fdest, indirizzo` *pseudoistruzione*

Store di numeri in virgola mobile in singola precisione

`s.s fdest, indirizzo` *pseudoistruzione*

Memorizza in *indirizzo* il numero in virgola mobile in doppia (singola) precisione contenuto nel registro `fdest`.

Sottrazione di numeri in virgola mobile in doppia precisione

`sub.d fd, fs, ft`

0x11	1	ft	fs	fd	1
6	5	5	5	5	6

Sottrazione di numeri in virgola mobile in singola precisione

`sub.s fd, fs, ft`

0x11	0	ft	fs	fd	1
6	5	5	5	5	6

Calcola la differenza dei numeri in virgola mobile in doppia (singola) precisione contenuti nei registri `fs` e `ft`, e mette il risultato nel registro `fd`.

Istruzioni per la gestione di eccezioni ed interruzioni

Ritorno da eccezione

`rfe`

0x10	1	0	0x20
6	1	19	6

Ripristina il registro Stato.

Chiamata di sistema

`syscall`

0	0	0xc
6	20	6

Il registro `$v0` contiene il numero della chiamata di sistema (vedere figura A.17), fra quelle previste da SPIM.

Break

```
break codice
```

0	codice	0xd
6	20	6

Causa un'eccezione identificata da *codice*; l'eccezione 1 è riservata al debugger.

Nessuna operazione

```
nop
```

0	0	0	0	0	0
6	5	5	5	5	6

Non fa nulla.

A.11 Note conclusive

Il programmatore che scrive in linguaggio assembler rinuncia alle vantaggiose funzionalità dei linguaggi ad alto livello (come le strutture dati, il controllo sul tipo delle variabili o i costrutti per il controllo del flusso) per ottenere in cambio il completo controllo sull'esecuzione delle istruzioni. Vincoli esterni che influenzano alcune applicazioni, come il tempo di risposta o la dimensione del programma, possono richiedere che il programmatore presti attenzione alla singola istruzione. Il costo di questo livello di attenzione, però, sta nel fatto che i programmi assembler sono più lunghi, richiedono un maggior tempo di sviluppo, e sono più difficili da mantenere rispetto ai programmi scritti in linguaggi ad alto livello.

Inoltre tre fattori tendono a ridurre la necessità di scrivere programmi in linguaggio assembler. Il primo fattore è il miglioramento dei compilatori: quelli moderni producono codice generalmente confrontabile con il miglior codice scritto da un programmatore umano, e talvolta anche migliore. La seconda tendenza è l'introduzione di nuovi processori, non solo più veloci ma, nel caso dei processori che eseguono molte istruzioni contemporaneamente, molto più difficili da programmare manualmente; in più la rapida evoluzione dei calcolatori di oggi favorisce i linguaggi ad alto livello, che non sono legati ad una singola architettura. L'ultima tendenza è verso lo sviluppo di applicazioni sempre più complesse, caratterizzate da interfacce grafiche sofisticate e da un numero maggiore di funzionalità rispetto alle precedenti: applicazioni di grandi dimensioni sono scritte da gruppi di programmatori, e richiedono caratteristiche di modularità e di controllo semantico fornite solo dai linguaggi ad alto livello.

Letture consigliate

Kane, G., e J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ.
Tutti i dettagli sul set di istruzioni MIPS e sulla programmazione assembler di questi calcolatori.

Aho, A., R. Sethi, e J. Ullman [1985]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA.
Un po' datato e carente per quanto riguarda le architetture moderne, ma pur sempre il riferimento classico sui compilatori.

A.12 Parole chiave

In questa appendice sono stati introdotti alcuni termini importanti. Coloro che non sono sicuri del significato dei termini sono invitati a fare riferimento al Glossario, in cui compaiono le definizioni complete.

backpatching
compilazione separata
convenzione sull'uso dei registri o di chiamata a procedura
dati statici
direttiva assembler
etichetta esterna o globale
etichetta locale
frame di chiamata a procedura o stack frame
gestore delle interruzioni
indirizzamento assoluto
informazioni di rilocazione
linguaggio macchina
linguaggio sorgente
macchina virtuale
macro
parametro formale
procedure ricorsive
registro salvato dal chiamante
registro salvato dal chiamato
riferimento in avanti
riferimento non risolto
segmento di dato
segmento di stack
segmento di testo
tabella dei simboli

A.13 Esercizi

A.1 [5] <§A.5> Il Paragrafo A.5 descrive come è suddivisa la memoria nella maggior parte dei sistemi MIPS; proporre una suddivisione diversa che raggiunga il medesimo scopo.

A.2 [20] <§A.6> Riscrivere il codice della procedura `fatt` usando un numero minore di istruzioni.

A.3 [5] <§A.7> In un programma utente, è sempre sicuro utilizzare i registri `$k0` e `$k1`?

A.4 [25] <§A.7> Il Paragrafo A.7 contiene il codice di un gestore delle interruzioni molto semplice. Un problema serio di questo gestore è che disabilita le interruzioni per un tempo molto lungo, il che significa che le interruzioni di un dispositivo di I/O molto veloce possono andar perse. Scrivere un gestore delle interruzioni migliore, che possa essere interrotto e che abiliti le interruzioni il più rapidamente possibile.

A.5 [15] <§A.7> Il semplice gestore delle interruzioni considerato ritorna sempre all'istruzione che segue quella che ha causato l'eccezione. Questo funziona bene purché l'istruzione che ha causato l'eccezione non sia nello slot di ritardo di un salto condizionato, nel qual caso l'istruzione successiva dovrebbere essere la destinazione del salto. Scrivere un gestore delle interruzioni migliore, che usi il registro EPC per determinare quale istruzione debba essere eseguita dopo l'eccezione.

A.6 [5] <§A.9> Usando SPIM, scrivere e verificare un programma di somma che legga ripetutamente dei numeri interi e li sommi di volta in volta. Il programma deve terminare quando riceve in ingresso uno 0, stampando la somma calcolata fino a quel momento.

A.7 [5] <§A.9> Usando SPIM, scrivere e verificare un programma che legga tre numeri interi e stampi la somma dei due maggiori. Usare le chiamate di sistema di SPIM descritte alla fine del Paragrafo A.9. In caso di condizioni di parità si può agire in modo arbitrario.

A.8 [5] <§A.9> Usando SPIM, scrivere e verificare un programma che legga un numero intero positivo con le chiamate di sistema di SPIM. Se il numero introdotto non è positivo il programma deve terminare con il messaggio "Dato non valido"; in caso contrario deve stampare il nome delle cifre che compongono il numero, ciascuna separata da uno ed un solo spazio. Se ad esempio l'utente ha scritto "728" l'uscita del programma deve essere "sette due otto".

A.9 [25] <§A.9> Scrivere e verificare un programma scritto in assembler MIPS che calcoli e stampi i primi 100 numeri primi; un numero n è primo se non è divisibile per nessun numero a parte se stesso e l'unità. È necessario scrivere due procedure:

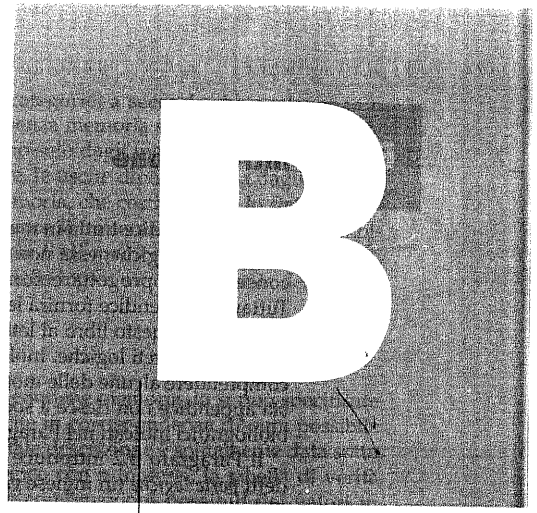
- `primo (n)`: restituisce 1 se n è primo, e 0 se non lo è;
- `main ()`: esegue un ciclo sui numeri interi, verificando se ciascuno di essi è primo; stampa i primi 100 numeri primi.

Controllare la correttezza del programma utilizzando SPIM.

A.10 [10] <§§A.6, A.9> Usando SPIM, scrivere e verificare un programma ricorsivo per risolvere il classico rompicapo matematico della Torre di Hanoi (essendo ricorsivo, il programma richiederà l'uso di stack frame). Il rompicapo consiste di tre pioli (1, 2 e 3) e n dischi (il numero n può variare, e valori tipici sono compresi fra 1 e 8). Il disco 1 è più piccolo del disco 2, che a sua volta è più piccolo del disco 3, e così via, fino al disco n che è il più largo. All'inizio tutti i dischi sono infilati nel piolo 1, con il disco n sul fondo, il disco $n - 1$ subito sopra, fino al disco 1 che si trova il alto. Lo scopo è spostare tutti i dischi nel piolo 2. È possibile spostare un solo disco per volta, cioè il disco più in alto di uno qualsiasi dei pioli al di sopra dei dischi di uno degli altri due pioli; inoltre non si può mettere un disco più grande sopra un disco più piccolo.

Il programma C riportato nel seguito può essere utile nella stesura del programma assembler.

```
/* sposta gli n dischi più piccoli dal piolo inizio al piolo
   fine usando il piolo extra */
void hanoi(int n, int inizio, int fine, int extra) {
  if(n !=0) {
    hanoi(n-1, inizio, extra, fine);
    print_string("Sposto il disco ");
    print_int(n);
    print_string("dal piolo ");
    print_int(inizio);
    print_string("al piolo ");
    print_int(fine);
    print_string(".\n");
    hanoi(n-1, extra, fine, inizio);
  }
}
main() {
  int n;
  print_string("Inserisci il numero di dischi> ");
  n = read_int();
  hanoi(n, 1, 2, 3);
  return 0;
}
```

A P P E N D I C E

Fondamenti sulla progettazione di reti logiche

Ho sempre amato questa parola: Booleano.

Claude Shannon
IEEE Spectrum, Aprile 1992
(La tesi di laurea di Shannon dimostrò che l'algebra inventata da George Boole nel diciannovesimo secolo poteva modellare il funzionamento degli interruttori elettrici)

A.8 [5] <§A.9> Usando SPIM, scrivere e verificare un programma che legga un numero intero positivo con le chiamate di sistema di SPIM. Se il numero introdotto non è positivo il programma deve terminare con il messaggio "Dato non valido"; in caso contrario deve stampare il nome delle cifre che compongono il numero, ciascuna separata da uno ed un solo spazio. Se ad esempio l'utente ha scritto "728" l'uscita del programma deve essere "sette due otto".

A.9 [25] <§A.9> Scrivere e verificare un programma scritto in assembler MIPS che calcoli e stampi i primi 100 numeri primi; un numero n è primo se non è divisibile per nessun numero a parte se stesso e l'unità. È necessario scrivere due procedure:

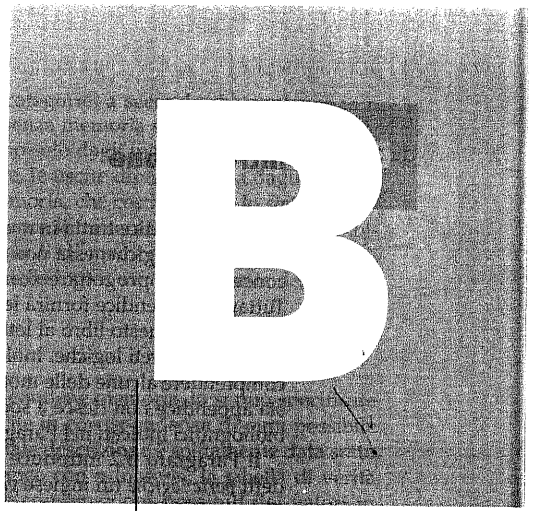
- `primo (n)`: restituisce 1 se n è primo, e 0 se non lo è;
- `main ()`: esegue un ciclo sui numeri interi, verificando se ciascuno di essi è primo; stampa i primi 100 numeri primi.

Controllare la correttezza del programma utilizzando SPIM.

A.10 [10] <§§A.6, A.9> Usando SPIM, scrivere e verificare un programma ricorsivo per risolvere il classico rompicapo matematico della Torre di Hanoi (essendo ricorsivo, il programma richiederà l'uso di stack frame). Il rompicapo consiste di tre pioli (1, 2 e 3) e n dischi (il numero n può variare, e valori tipici sono compresi fra 1 e 8). Il disco 1 è più piccolo del disco 2, che a sua volta è più piccolo del disco 3, e così via, fino al disco n che è il più largo. All'inizio tutti i dischi sono infilati nel piolo 1, con il disco n sul fondo, il disco $n - 1$ subito sopra, fino al disco 1 che si trova il alto. Lo scopo è spostare tutti i dischi nel piolo 2. È possibile spostare un solo disco per volta, cioè il disco più in alto di uno qualsiasi dei pioli al di sopra dei dischi di uno degli altri due pioli; inoltre non si può mettere un disco più grande sopra un disco più piccolo.

Il programma C riportato nel seguito può essere utile nella stesura del programma assembler.

```
/* sposta gli n dischi più piccoli dal piolo inizio al piolo
   fine usando il piolo extra */
void hanoi(int n, int inizio, int fine, int extra) {
  if(n !=0) {
    hanoi(n-1, inizio, extra, fine);
    print_string("Sposto il disco ");
    print_int(n);
    print_string("dal piolo ");
    print_int(inizio);
    print_string("al piolo ");
    print_int(fine);
    print_string(".\n");
    hanoi(n-1, extra, fine, inizio);
  }
}
main() {
  int n;
  print_string("Inserisci il numero di dischi> ");
  n = read_int();
  hanoi(n, 1, 2, 3);
  return 0;
}
```

A P P E N D I C E

Fondamenti sulla progettazione di reti logiche

Ho sempre amato questa parola: Booleano.

Claude Shannon
IEEE Spectrum, Aprile 1992
(La tesi di laurea di Shannon dimostrò che l'algebra inventata da George Boole nel diciannovesimo secolo poteva modellare il funzionamento degli interruttori elettrici)

B.1 Introduzione

Questa appendice tratta in modo sintetico i principi fondamentali relativi al progetto delle reti logiche; essa non mira a sostituire un corso di progettazione logica, né consentirà di progettare correttamente sistemi logici di una certa complessità. Tuttavia l'appendice fornirà le informazioni sufficienti a comprendere gli argomenti trattati in questo libro ai lettori che non abbiano sufficienti conoscenze sulla progettazione di reti logiche. Inoltre sarà un'utile introduzione per chi sia interessato a comprendere alcune delle motivazioni alla base del progetto dei calcolatori; se questa appendice non riesce a soddisfare le curiosità che ha fatto nascere, i riferimenti bibliografici indicati nel Paragrafo 3.9 indicano diverse fonti alternative.

Il Paragrafo B.2 introduce i blocchi costitutivi fondamentali delle reti logiche, detti *porte logiche* (in inglese *gate*); il Paragrafo B.3 usa tali blocchi nella costruzione di semplici sistemi logici di tipo *combinatorio*, ossia non dotati di memoria. Chi avesse già seguito dei corsi introduttivi sulla progettazione logica probabilmente avrà già acquisito le conoscenze contenute in questi primi due paragrafi. Il Paragrafo B.4 è una breve introduzione sull'argomento della temporizzazione, necessario per comprendere il funzionamento degli elementi di memoria; questi ultimi verranno introdotti nel Paragrafo B.5, dove si descriveranno sia le caratteristiche utili alla comprensione del materiale trattato nei Capitoli 5 e 6, sia le motivazioni alla base del progetto delle gerarchie di memorie del Capitolo 7. Il Paragrafo B.6 illustra la progettazione e l'utilizzo delle macchine a stati finiti, un tipo particolare di reti logiche sequenziali. Se si intende passare alla lettura dell'Appendice C, è necessario comprendere in dettaglio il materiale trattato nei paragrafi da B.2 a B.6; al fine della comprensione dei Capitoli 5 e 6 è invece sufficiente una lettura superficiale di questa appendice con l'esclusione del Paragrafo B.7. Quest'ultimo paragrafo è infatti dedicato a coloro che desiderino un maggior approfondimento sulle metodologie di temporizzazione e di utilizzo dei segnali di clock; esso descrive il principio di funzionamento della metodologia di temporizzazione sensibile ai fronti, introduce una metodologia di temporizzazione alternativa, ed infine tratta il problema della sincronizzazione di ingressi di tipo asincrono.

B.2 Porte logiche, tabelle di verità ed equazioni logiche

I componenti elettronici interni ai calcolatori odierni sono di tipo *digitale*: l'elettronica di tipo digitale lavora con due soli livelli di tensione: una tensione alta ed una tensione bassa. Tutti gli altri valori sono temporanei e compaiono solo durante le transizioni tra i due valori principali. Come già accennato nel Capitolo 3, questo è il motivo principale per cui gli elaboratori usano i numeri binari: il sistema di numerazione binario corrisponde infatti perfettamente al modello astratto del livello elettronico sottostante. Nelle diverse famiglie logiche i valori e le relazioni tra i due livelli di tensione sono diversi, per cui anziché parlare di livelli di tensione, si parlerà di segnali *veri* (dal punto di vista logico), o pari ad 1, o *affermati*, e di segnali *falsi* (dal punto di vista logico), o pari a 0, o *non affermati* o *negati*. Il valore 0 è il *complemento* o *inverso* del valore 1, e viceversa.

I circuiti logici sono classificati in due categorie, a seconda che contengano o meno degli elementi di memoria. I circuiti senza memoria sono detti *combinatori*: le uscite di un circuito combinatorio dipendono solo dagli ingressi presenti in quell'istante. Nei circuiti dotati di memoria, invece, le uscite dipendono sia dagli ingressi che dai valori presenti negli elementi di memoria, che costituiscono lo *stato* del circuito logico. In questo paragrafo e nel seguente si tratteranno solo i circuiti combinatori; dopo aver introdotto i diversi elementi di memoria, nel Paragrafo B.5 si descriveranno le tecniche di progettazione della logica *sequenziale*, ossia dei circuiti che contengono informazioni di stato.

Tabelle di verità

Poiché un circuito logico combinatorio non contiene memoria, lo si può specificare completamente definendo i valori delle uscite in corrispondenza di ogni possibile combinazione dei valori degli ingressi; una tale descrizione è solitamente data sotto forma di *tabella di verità*. Per un circuito logico con n ingressi, la tabella di verità contiene 2^n elementi, poiché tante sono le combinazioni possibili dei valori in ingresso; ciascuna riga della tabella specifica il valore di tutte le uscite in corrispondenza di quella particolare combinazione di ingressi.

Tabelle di verità

Esempio Si consideri una funzione logica con tre ingressi A, B e C e tre uscite D, E ed F. La funzione è definita come segue: D è vero se almeno uno degli ingressi è vero, E è vero se esattamente due ingressi sono veri, F è vero solo se tutti e tre gli ingressi sono veri. Si mostri la tabella di verità di questa funzione.

Soluzione La tabella di verità contiene $2^3 = 8$ elementi, ed è la seguente:

Ingressi			Uscite		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Le tabelle di verità possono descrivere completamente qualsiasi funzione logica combinatoria; purtroppo crescono velocemente di dimensioni e possono divenire difficilmente comprensibili. Nel caso di funzioni che valgono 0 per molte combinazioni di ingressi, se ne adotterà una versione semplificata, elencando solo gli elementi della tabella di verità corrispondenti ai valori non nulli delle uscite; un tale approccio è adottato sia nel Capitolo 5 che nell'Appendice C.

Algebra booleana

Un approccio alternativo consiste nell'esprimere la funzione logica mediante delle equazioni logiche: ciò è possibile utilizzando l'*algebra booleana* (che prende il nome da Boole, un matematico del diciannovesimo secolo). Nell'algebra booleana tutte le variabili assumono il valore 0 o 1 e, nelle formulazioni più diffuse, vi sono tre operatori:

- L'operatore OR è indicato da +, come nell'espressione $A + B$. Il risultato di un'operazione OR è pari a 1 se almeno una delle variabili vale 1. L'operazione OR è anche detta *somma logica*, poiché il risultato è 1 se almeno un operando è 1.
- L'operatore AND è indicato da ·, come nell'espressione $A \cdot B$. Il risultato di un'operazione AND è pari a 1 se tutti e due gli ingressi valgono 1. L'operazione AND è anche detta *prodotto logico*, poiché il risultato è 1 se entrambi gli operandi sono 1.
- L'operatore unario NOT è indicato come $\overline{A}$. Il risultato di un'operazione NOT è pari a 1 solo se l'ingresso è 0. Applicando l'operatore NOT ad un valore logico si ottiene l'inversione (o negazione) del valore (in altre parole, se l'ingresso era 0 l'uscita sarà 1, e viceversa).

Vi sono diverse proprietà dell'algebra booleana che risultano utili nel manipolare le equazioni logiche:

- Proprietà di identità: $A + 0 = A$ e $A \cdot 1 = A$.
- Proprietà di assorbimento: $A + 1 = 1$ e $A \cdot 0 = 0$.
- Proprietà dell'inverso: $A + \overline{A} = 1$ e $A \cdot \overline{A} = 0$.
- Proprietà commutativa: $A + B = B + A$ e $A \cdot B = B \cdot A$.
- Proprietà associativa: $A + (B + C) = (A + B) + C$ e $A \cdot (B \cdot C) = (A \cdot B) \cdot C$.
- Proprietà distributiva: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ e $A + (B \cdot C) = (A + B) \cdot (A + C)$.

Oltre alle proprietà precedenti, ve ne sono altre due utili, dette leggi di DeMorgan, che verranno presentate negli esercizi dal B.6 al B.9.

Ogni insieme di funzioni logiche può essere scritto come una serie di equazioni in cui le uscite compaiono nel primo membro di ciascuna equazione e nel secondo membro vi è una formula costruita a partire dalle variabili di ingresso e dai tre operatori appena visti.

Equazioni logiche

Esempio Si calcolino le equazioni logiche per la funzione logica descritta nell'esempio precedente.

Soluzione L'equazione che specifica D è la seguente:

$$D = A + B + C$$

F è altrettanto semplice:

$$F = A \cdot B \cdot C$$

L'uscita E è invece un po' più complessa. Conviene considerarla in due parti: che cosa deve essere vero affinché E sia vera (devono essere veri due dei tre ingressi) e che cosa non può essere vero (non possono essere veri tutti e tre gli ingressi). Si può quindi scrivere E come

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot (\overline{A \cdot B \cdot C})$$

Si può altresì derivare l'espressione di E considerando che deve assumere un valore vero se e solo se esattamente due degli ingressi sono veri. Si può quindi scrivere E come un OR dei tre possibili termini che hanno due ingressi veri ed uno falso:

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

La dimostrazione che queste due espressioni sono equivalenti sarà oggetto dell'Esercizio B.7.

Porte logiche

I circuiti logici sono costruiti a partire da *porte logiche* (o *gate*) che implementano le funzioni logiche primitive. Per esempio la porta logica AND implementa la funzione AND e la porta OR implementa la funzione OR. Poiché le funzioni AND e OR sono commutative ed associative, le porte AND e OR possono avere più ingressi, essendo l'uscita il risultato dell'operazione AND o OR su tutti gli ingressi. La funzione logica NOT è implementata mediante un invertitore che ha sempre un solo ingresso. La rappresentazione standard di questi tre blocchi logici elementari è riportata nella figura B.1.

Anziché disegnare esplicitamente gli invertitori, è uso comune aggiungere dei "pallini" agli ingressi o all'uscita delle porte logiche per indicare che il valore logico su tale linea di ingresso o uscita deve essere invertito. Per esempio la figura B.2 mostra il diagramma logico della funzione $\overline{\overline{A} + B}$, usando a sinistra degli invertitori espliciti ed a destra la convenzione dei "pallini" sugli ingressi e sulle uscite.

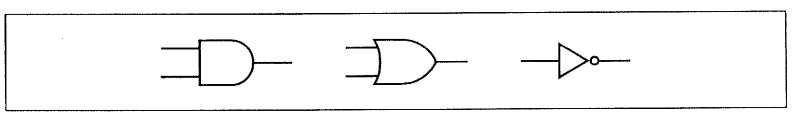


Figura B.1 Rappresentazione grafica standard della porta logica AND, della porta OR e dell'invertitore, da sinistra a destra. I segnali alla sinistra di ciascun simbolo sono gli ingressi, mentre l'uscita è rappresentata a destra. Le porte AND e OR hanno due ingressi, mentre l'invertitore ne ha uno solo.

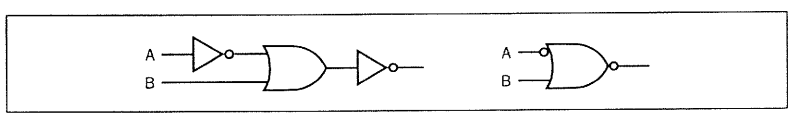


Figura B.2 Implementazione mediante porte logiche della funzione $\overline{\overline{A} + B}$ utilizzando invertitori espliciti (a sinistra) e "pallini" sugli ingressi e sulle uscite (a destra). Questa funzione logica può anche essere semplificata in $A \cdot \overline{B}$.

È possibile costruire una qualsiasi funzione logica mediante le porte AND, OR e l'inversione; molti degli esercizi forniscono l'opportunità di implementare alcune funzioni logiche frequentemente utilizzate mediante porte logiche. Nel prossimo paragrafo si vedrà come queste conoscenze possono essere messe a frutto per ricavare l'implementazione di una qualsiasi funzione logica.

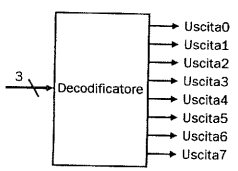
In effetti, tutte le funzioni logiche sono realizzabili a partire da un solo tipo di porta logica, a patto che questa comprenda un'inversione. Le due porte logiche invertenti più diffuse sono dette NOR e NAND e corrispondono, rispettivamente, alla porta OR con uscita invertita ed alla porta AND con uscita invertita. Le porte NOR e NAND sono dette *universali* in quanto si può costruire una qualsiasi funzione logica a partire da quel solo tipo di porta logica. Gli Esercizi B.3 e B.4 tratteranno la dimostrazione di questa proprietà.

B.3 Logica combinatoria

In questo paragrafo si analizzeranno alcuni circuiti logici fondamentali che verranno utilizzati frequentemente e si tratterà la progettazione della logica strutturata implementabile in modo automatizzato, mediante un programma di traduzione, a partire da un'equazione logica o da una tabella di verità. Si tratterà infine la nozione di matrice di elementi logici.

Decodificatori

Un circuito logico che viene frequentemente utilizzato nella costruzione di circuiti complessi è il *decodificatore* (o *decoder*). Il tipo più diffuso di decodificatore ha un ingresso di n bit e 2^n uscite, in cui viene affermata una sola uscita per ciascuna combinazione di ingressi. Il decodificatore traduce l'ingresso su n bit in un segnale che corrisponde al valore binario dell'ingresso; per far ciò le uscite sono solitamente numerate, ad esempio come Uscita0, Uscita1, ..., Uscita2^n-1: se il valore dell'ingresso è pari ad i, allora l'uscita Uscita$i$ sarà vera e tutte le altre uscite saranno false. La figura B.3 mostra un decodificatore a 3 bit e la corrispondente tabella della verità; tale



a. Un decodificatore a 3-bit

Ingressi			Uscite							
I2	I1	I0	Uscita 7	Uscita 6	Uscita 5	Uscita 4	Uscita 3	Uscita 2	Uscita 1	Uscita 0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Figura B.3 Un decodificatore a 3 bit ha 3 ingressi, detti I2, I1 e I0 e 2^3 uscite, da Uscita0 ad Uscita7. Assume un valore vero solo l'uscita corrispondente al valore binario dell'ingresso, come indicato dalla tabella di verità. L'etichetta 3 sul segnale di ingresso sta ad indicare che tale segnale ha ampiezza pari a 3 bit.

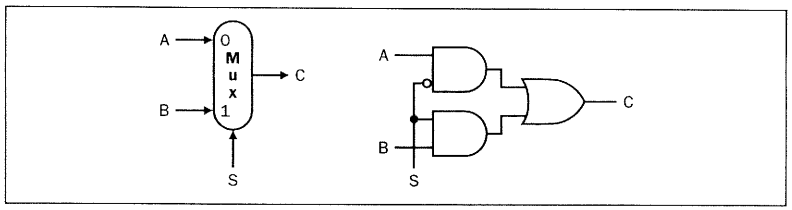


Figura B.4 Un multiplexer a due vie (a sinistra) e la sua implementazione mediante porte logiche (a destra). Il multiplexer ha due ingressi di dato (A e B), indicati dalle etichette 0 e 1, ed un ingresso di selezione (S), oltre ad un'uscita C.

elemento è detto anche *decodificatore da 3 a 8* poiché vi sono 3 ingressi ed 8 (2^3) uscite. Esiste anche un elemento logico detto *codificatore* (o *encoder*) che calcola la funzione inversa del decodificatore, ricevendo 2^n ingressi e producendo un'uscita di n bit.

Multiplexer

Una funzione logica particolarmente utilizzata nei Capitoli 4, 5 e 6 è quella del *multiplexer*; esso potrebbe più propriamente chiamarsi *selettore*, poiché la sua uscita è uguale ad uno degli ingressi, scelto mediante un segnale di controllo. Si consideri un multiplexer a due vie: come indicato nella parte sinistra della figura B.4, questo multiplexer ha tre ingressi, due di dato ed uno di selezione (o di controllo). Il valore dell'ingresso di selezione determina quale degli ingressi viene usato per determinare il valore dell'uscita. Si può rappresentare la funzione logica calcolata dal multiplexer a due vie come $C = (A \cdot \overline{S}) + (B \cdot S)$, che è rappresentata sotto forma di porte logiche nella parte destra della figura B.4.

Si possono creare dei multiplexer con un numero arbitrario di ingressi di dato. Quando vi sono due soli ingressi, vi è un solo segnale di selezione che identifica uno dei due ingressi se è vero (1) o l'altro se è falso (0). Se vi fossero n ingressi di dato, servirebbero $\lceil \log_2 n \rceil$ ingressi di selezione; in tal caso il multiplexer sarebbe composto da tre parti fondamentali:

1. Un decodificatore che genera n segnali, ciascuno corrispondente ad un diverso valore dell'ingresso di selezione.
2. Una schiera di n porte AND, ciascuna delle quali combina uno degli ingressi con uno dei segnali provenienti dal decodificatore.
3. Un'unica grande porta OR che raccoglie tutte le uscite delle porte AND.

Per associare gli ingressi ai valori dei segnali di selezione, si usano spesso etichette numeriche (0, 1, 2, 3, ..., $n - 1$) sugli ingressi di dato e si interpretano i valori sugli ingressi di selezione come un numero binario. Talvolta si farà uso di multiplexer in cui i segnali di selezione non sono codificati.

Logica a due livelli e PLA

Nel paragrafo precedente si è evidenziato che qualsiasi funzione logica può essere implementata ricorrendo alle sole funzioni AND, OR e NOT. In effetti risulta valida una proprietà ancora più restrittiva: ciascuna funzione logica può essere riportata ad una forma canonica, in cui per ciascun ingresso si ha una sola variabile, affermata o complementata, ed in cui sono presenti due soli livelli di porte logiche, una

porta AND ed una OR, più un'eventuale inversione dell'uscita finale. Tale rappresentazione è detta *rappresentazione a due livelli* e può prendere due forme, dette rispettivamente *prodotto di somme* e *somma di prodotti*. Una rappresentazione come somma di prodotti è una somma logica (OR) di prodotti (termini calcolati con l'operatore AND); un prodotto di somme è esattamente il contrario. Nell'esempio precedente si erano viste due equazioni per determinare l'uscita E:

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot (\overline{A \cdot B \cdot C})$$

e

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

La seconda equazione è nella forma di una somma di prodotti: ha due livelli di logica e le uniche operazioni di inversione sono applicate a singole variabili. Nella prima equazione vi sono invece tre livelli di logica.

Approfondimento: Si può anche scrivere E come prodotto di somme:

$$E = \overline{(\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{C} + B) \cdot (\overline{B} + \overline{C} + A)}$$

Per ricavare quest'ultima forma è necessario usare i *teoremi di DeMorgan*, trattati a partire dall'Esercizio B.6. L'Esercizio B.8 chiede di ricavare la rappresentazione sotto forma di prodotto di somme a partire dalla rappresentazione come somma di prodotti usando i teoremi di DeMorgan.

In questo testo verrà usata la forma più comune, cioè la somma di prodotti; è facile vedere che qualunque funzione logica può essere rappresentata come somma di prodotti costruendo tale rappresentazione a partire dalla tabella di verità della funzione. Ciascun elemento della tabella di verità per cui la funzione assume un valore vero corrisponde ad un termine prodotto: il termine prodotto consiste del prodotto logico di tutti gli ingressi o dei loro complementi, a seconda che l'elemento della tabella della verità abbia un 1 od uno 0 in corrispondenza di tale variabile. La funzione logica è la somma logica dei termini prodotto corrispondenti alle combinazioni per cui la funzione è vera. Tutto questo si chiarirà molto più semplicemente con un esempio.

Esempio

Somma di prodotti

Si calcoli la rappresentazione come somma di prodotti a partire dalla seguente tabella di verità:

Ingressi			Uscita
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	1	1	0
1	1	0	0
1	1	1	1

Soluzione

Vi sono quattro termini prodotto, poiché la funzione è vera (1) per quattro diverse combinazioni degli ingressi, in particolare:

$$\overline{A} \cdot \overline{B} \cdot C$$
$$\overline{A} \cdot B \cdot \overline{C}$$
$$A \cdot \overline{B} \cdot \overline{C}$$
$$A \cdot B \cdot C$$

Si può dunque scrivere la funzione che esprime il valore di D come somma di tali termini:

$$D = (\overline{A} \cdot \overline{B} \cdot C) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot B \cdot C)$$

Si noti che solo gli elementi della tabella di verità corrispondenti ai valori veri della funzione contribuiscono alla generazione di nuovi termini nell'equazione.

È possibile sfruttare la relazione vista tra tabelle di verità e rappresentazioni a due livelli per generare un'implementazione mediante porte logiche di qualsiasi insieme di funzioni logiche. Un insieme di funzioni logiche è rappresentabile da una tabella di verità con più colonne di uscita, come già visto in un precedente esempio, in cui ciascuna colonna rappresenta una diversa funzione logica che può essere costruita direttamente dalla tabella di verità.

La rappresentazione come somma di prodotti corrisponde ad una diffusa tecnologia di implementazione mediante logica strutturata detta *programmable logic array* (PLA, letteralmente *matrice logica programmabile*). Una PLA dispone di un insieme di ingressi e dei relativi valori complementati (che si possono ottenere per mezzo di invertitori) e di due stadi di logica. Il primo stadio è una matrice di porte AND che formano un insieme di termini prodotto (talvolta detti *minterm* o *mintermini*); ciascun termine prodotto combina le variabili di ingresso o i loro complementi. Il secondo stadio è una matrice di porte OR, ciascuna delle quali calcola la somma logica di un numero arbitrario di termini prodotto. La figura B.5 illustra la struttura fondamentale di una PLA.

Un dispositivo PLA è in grado di realizzare direttamente la tabella di verità di un insieme di funzioni logiche con più ingressi ed uscite. Poiché ciascun elemento della tabella di verità in cui essa è vera richiede un termine prodotto, vi sarà una riga corrispondente nella PLA; ciascuna uscita invece corrisponde in genere ad una

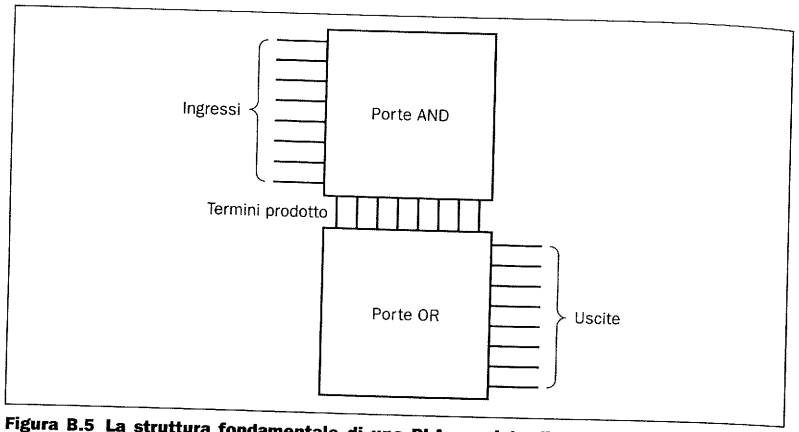


Figura B.5 La struttura fondamentale di una PLA consiste di una matrice di porte AND seguita da una matrice di porte OR. Ciascun elemento della matrice di porte AND è un termine prodotto che consiste di un certo numero di ingressi o ingressi invertiti; ciascun elemento della matrice di porte OR è un termine somma che consiste di un certo numero di termini prodotto.

riga di porte OR del secondo stadio. Il numero di porte OR corrisponde al numero di elementi della tabella di verità per cui ciascuna uscita è vera. La dimensione totale di una PLA come quella mostrata in figura B.5 è data dalla somma della dimensione della matrice delle porte AND (detta *piano AND*) con la dimensione della matrice delle porte OR (detta *piano OR*). Analizzando la figura B.5, si deduce che la dimensione della matrice di porte AND è pari al numero di ingressi moltiplicato per il numero di termini prodotto, mentre la dimensione della matrice di porte OR è pari al numero di uscite moltiplicato per il numero di termini prodotto.

Le PLA hanno due caratteristiche che le rendono utili per implementare efficacemente gli insiemi di funzioni logiche: innanzitutto è sufficiente tenere conto dei soli elementi della tabella di verità che producono un valore vero per almeno un'uscita; in secondo luogo ogni termine prodotto distinto compare una sola volta nella PLA, anche se viene usato in più uscite, come illustrato dal seguente esempio.

Dispositivi PLA

Esempio Si consideri l'insieme delle funzioni logiche definite nel primo esempio del Paragrafo B.2 e si mostri un'implementazione mediante PLA di tali funzioni.

Soluzione La tabella di verità precedentemente costruita è la seguente:

Ingressi			Uscite		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Essendovi sette termini prodotto distinti che hanno almeno un valore vero nella parte relativa alle uscite, vi saranno sette colonne nel piano AND, mentre il numero di righe del piano AND è pari a tre (il numero degli ingressi); il numero di righe del piano OR è anch'esso pari a tre (essendoci tre uscite). La figura B.6 mostra la PLA risultante, in cui i termini prodotto corrispondono agli elementi della tabella di verità a partire dall'alto.

Anziché disegnare esplicitamente le porte logiche, come nella figura B.6, i progettisti spesso indicano solo la posizione delle porte AND e delle porte OR: si usano dei puntini nell'intersezione di una linea di segnale relativa ad un termine prodotto con le linee corrispondenti agli ingressi o alle uscite ogniqualvolta è necessaria una porta di tipo AND o OR. La figura B.7 riporta la stessa PLA di figura B.6 usando la nuova convenzione grafica. Il contenuto di una PLA è definito nel momento in cui la PLA viene creata, anche se esistono delle strutture simili alle PLA, dette *PAL*, che possono essere programmate elettronicamente nel momento in cui il progettista decide di usarle.

Dispositivi ROM

Un secondo tipo di logica strutturata che può essere usata nella realizzazione di un insieme di funzioni logiche è la *memoria a sola lettura* (*read-only memory* o ROM).

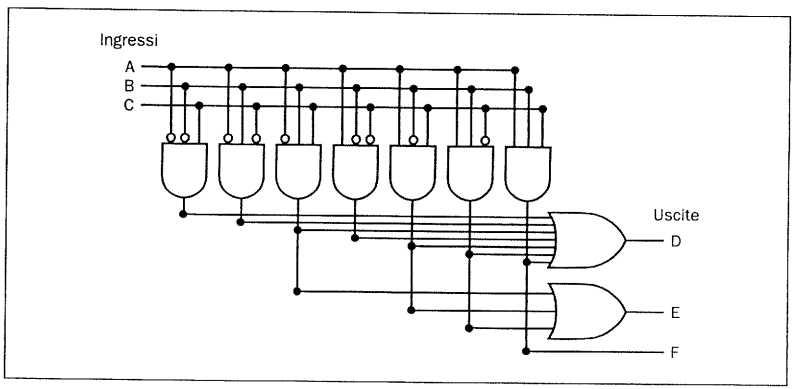


Figura B.6 La PLA che realizza la funzione logica dell'esempio.

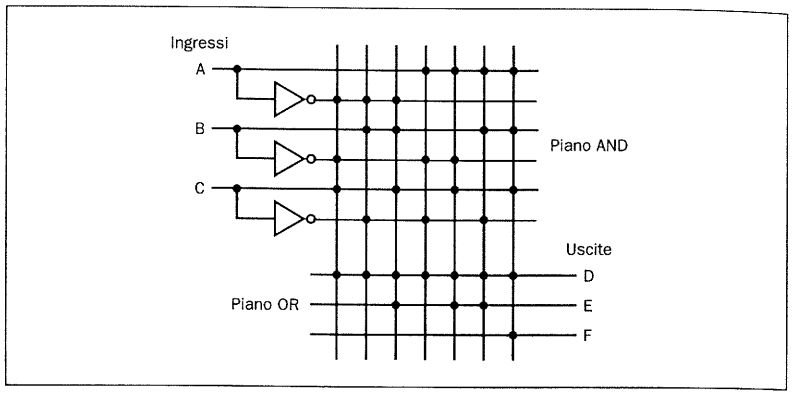


Figura B.7 Una PLA disegnata usando i puntini per indicare i termini prodotto ed i termini somma nella matrice. Anziché usare degli invertitori su ciascuna porta AND, solitamente gli ingressi attraversano il piano AND sia nella forma affermata che in quella complementata. Un puntino nel piano AND indica che l'ingresso (o il suo complemento) è presente nel termine prodotto corrispondente alla colonna. Un puntino nel piano OR indica che il termine prodotto compare nella relativa uscita.

Una ROM è detta "memoria" poiché dispone di un insieme di locazioni che possono essere lette; tuttavia il contenuto di tali locazioni è fisso, determinato al momento della creazione della ROM stessa. Vi sono anche *ROM programmabili* (dette PROM) che vengono programmate elettronicamente, non appena il progettista ne conosce il contenuto. Esistono inoltre delle PROM cancellabili, che richiedono però un processo di cancellazione molto lento mediante luce ultravioletta, per cui vengono usate come memorie di sola lettura, eccezion fatta per la fase di sviluppo e per quella di correzione degli errori.

I dispositivi ROM sono dotati di un insieme di linee di indirizzo in ingresso e di un insieme di uscite. Il numero di elementi indirizzabili nella ROM determina il numero di linee di indirizzo: se la ROM contiene 2^n elementi indirizzabili (l'*altezza* della ROM), vi sono n linee di ingresso. Il numero di bit presenti in ciascun elemento indirizzabile determina invece il numero di bit di uscita ed è talvolta detto *ampiezza* della ROM. Il numero totale di bit in una ROM è pari al prodotto tra altezza ed ampiezza; questi due parametri sono talvolta indicati complessivamente come la *forma* della ROM.

Una ROM può codificare un insieme di funzioni logiche direttamente a partire dalla tabella di verità: ad esempio, se vi sono n funzioni con m ingressi, è necessaria una ROM con m linee di indirizzo (e 2^m elementi) ed elementi ampi n bit ciascuno. Nella tabella di verità la parte relativa agli ingressi rappresenta gli indirizzi della ROM, mentre la parte relativa alle uscite corrisponde al suo contenuto. Se la tabella di verità è organizzata in modo tale che la sequenza delle combinazioni di ingresso crei una sequenza di numeri binari crescenti (come in tutte le tabelle di verità viste fin qui), la parte relativa alle uscite fornisce il contenuto della ROM nell'ordine corretto. Nell'esempio precedente vi erano tre ingressi e tre uscite, determinando quindi una ROM di $2^3 = 8$ elementi di 3 bit ciascuno. Il contenuto degli elementi della ROM, in ordine crescente di indirizzo, è dato direttamente dalla parte relativa alle uscite della tabella di verità riportata nell'esempio.

Vi è una stretta relazione tra ROM e PLA: le ROM sono dispositivi completamente decodificati, ossia contengono un'intera parola di uscita per tutte le possibili combinazioni di ingresso, mentre le PLA sono solo parzialmente decodificate. Ciò significa che una ROM conterrà sempre più elementi. Per la tabella di verità dell'esempio precedente, la ROM contiene gli elementi relativi a tutti gli otto possibili ingressi, mentre la PLA contiene solo i sette termini prodotto utili. Al crescere del numero di ingressi, il numero di elementi di una ROM cresce esponenzialmente, mentre per la maggior parte delle funzioni logiche il numero di termini prodotto cresce molto più lentamente (si vedano gli esempi riportati nell'Appendice C). Tale differenza rende in genere le PLA più efficienti ai fini della realizzazione delle funzioni logiche combinatorie. Le ROM hanno invece il vantaggio di poter realizzare qualsiasi funzione logica con un dato numero di ingressi ed uscite: è quindi possibile modificare più facilmente il contenuto della ROM se la funzione logica dovesse cambiare, poiché la dimensione della ROM resterà immutata.

Condizioni di indifferenza

Nell'implementazione della logica combinatoria nascono spesso delle situazioni in cui è indifferente il valore di qualche uscita, o perché qualche altra uscita ha assunto un valore particolare o perché nella determinazione delle uscite viene usato solo un sottoinsieme delle combinazioni di ingresso. Tali situazioni vengono denominate *condizioni di indifferenza* (in inglese *don't care*, letteralmente "non importa") e si rivelano molto importanti ai fini dell'ottimizzazione della realizzazione delle funzioni logiche.

Vi sono due tipi di condizioni di indifferenza: indifferenza in uscita e indifferenza in ingresso, entrambi rappresentabili su una tabella di verità. Le condizioni di *indifferenza in uscita* (*output don't care*) sorgono quando il valore di qualche uscita è indifferente in corrispondenza di qualche combinazione di ingresso; esse sono rappresentate con delle X nella parte relativa alle uscite della tabella di verità. Quando un'uscita è indifferente per qualche combinazione degli ingressi, il progettista o il programma di ottimizzazione logica è libero di fissare l'ingresso al valore vero o falso in corrispondenza di tale ingresso. Le condizioni di *indifferenza in ingresso* (*input don't care*) sorgono quando un'uscita dipende solo da alcuni ingressi, e sono anch'esse rappresentate nella tabella di verità con delle X, questa volta nella parte relativa agli ingressi.

Esempio

Condizioni di indifferenza

Si consideri una funzione logica con ingressi A, B e C definiti come segue:

- Se A o C sono veri, allora l'uscita D è vera, indipendentemente dal valore di B.
- Se A o B sono veri, allora l'uscita E è vera, indipendentemente dal valore di C.
- L'uscita F è vera se esattamente uno degli ingressi è vero, anche se non interessa il valore di F quando sia D che E sono veri.

Si mostri la tabella di verità completa di questa funzione e la tabella di verità con le condizioni di indifferenza. Quanti termini prodotto sono necessari, nei due casi, in un'implementazione mediante PLA?

Soluzione La tabella di verità completa, senza far uso di condizioni di indifferenza, è la seguente:

Ingressi			Uscite		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	1

Essa richiederebbe, senza ottimizzazioni, sette termini prodotto. La tabella di verità tenendo conto delle condizioni di indifferenza in uscita avrebbe il seguente aspetto:

Ingressi			Uscite		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

Essa si può ulteriormente semplificare come segue:

Ingressi			Uscite		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

La tabella di verità semplificata richiede una PLA con quattro soli minterm, oppure potrebbe essere implementata mediante porte logiche con due porte AND a due ingressi e tre porte OR (due con tre ingressi ed una con due ingressi). Si confrontino questi dati con la tabella di verità originaria, che aveva sette minterm e avrebbe richiesto quattro porte AND.

La minimizzazione delle funzioni logiche è fondamentale per ottenere delle implementazioni efficienti; uno strumento particolarmente utile per la minimizzazione manuale della logica sparsa sono le *mappe di Karnaugh*. Esse rappresentano

graficamente la tabella di verità, per cui i termini prodotto risultano più semplici da vedere e da combinare; tuttavia la minimizzazione manuale di funzioni complesse mediante mappe di Karnaugh è impossibile, per via della loro dimensione e della complessità derivante. Fortunatamente il processo di minimizzazione logica è di natura decisamente meccanica e può essere realizzato da appositi strumenti di progetto; nei processi di minimizzazione gli strumenti traggono vantaggio dalle condizioni di indifferenza, per cui è importante che queste vengano specificate. I riferimenti ai libri di testo elencati al termine dell'appendice forniscono approfondimenti sulla minimizzazione logica, sulle mappe di Karnaugh e sulla teoria alla base degli algoritmi di minimizzazione.

Matrici di elementi logici

Molte delle operazioni combinatorie che vengono realizzate sui dati devono elaborare delle intere parole di dato (32 bit): si deve quindi spesso costruire una matrice (o schiera) di elementi logici, che si può semplicemente rappresentare indicando che una data operazione viene applicata ad un intero insieme di ingressi. Prendendo ad esempio la funzione svolta dal multiplexer, all'interno di un calcolatore nella maggior parte dei casi occorrerà selezionare tra due interi *bus*. Un bus è definito come un insieme di linee di dato che vengono trattate dal punto di vista logico come un unico segnale (il termine *bus* è anche usato per indicare un gruppo di linee condivise con più sorgenti e più funzioni, soprattutto nel Capitolo 8, dove si trattano i bus di ingresso/uscita).

Ad esempio nel set di istruzioni MIPS il valore scritto nel registro che contiene il risultato di un'istruzione può provenire da due fonti diverse: si utilizzerà un multiplexer per scegliere quale dei due bus (di 32 bit ciascuno) verrà scritto nel registro risultato: occorrerà ripetere per 32 volte il multiplexer ad 1 bit visto in precedenza. Il fatto che un segnale sia un bus anziché un singolo bit si indica graficamente mediante un tratto più spesso; la maggior parte dei bus hanno ampiezza pari a 32 bit, ed in tal caso si omette l'etichetta indicante la dimensione. Quando si rappresenta un determinato componente collegandolo in ingresso e in uscita a dei bus, significa che il componente deve essere replicato il numero di volte opportuno in funzione dell'ampiezza del segnale di ingresso. La figura B.8 mostra la rappresentazione grafica di un multiplexer che seleziona tra due bus di 32 bit e come questo venga espanso in termini di multiplexer di ampiezza 1 bit. Talvolta è necessario costruire una matrice di elementi logici in cui gli ingressi di alcuni degli elementi siano le uscite degli elementi precedenti; ciò avviene ad esempio nel caso delle ALU a più bit. In tali casi è necessario indicare esplicitamente come costruire le matrici, poiché i singoli elementi non sono più indipendenti, a differenza del caso del multiplexer a 32 bit.

B.4 I segnali di temporizzazione (clock)

Prima di passare a trattare gli elementi di memoria e la logica di tipo sequenziale, risulta utile discutere brevemente i segnali di temporizzazione, o clock. Questo breve paragrafo introduce l'argomento ed approfondisce la discussione fatta all'inizio del Capitolo 5; maggiori dettagli sui segnali di clock e sulle metodologie di temporizzazione si trovano nel Paragrafo B.7.

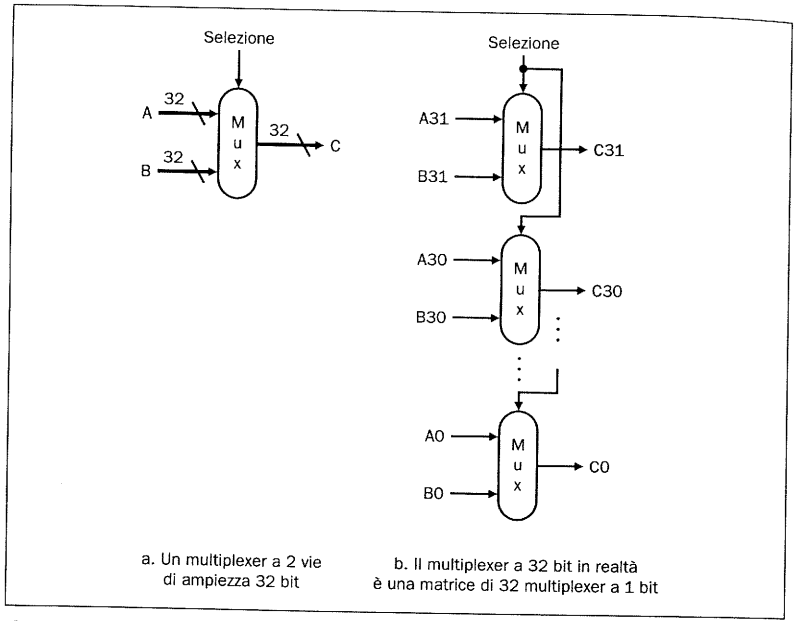


Figura B.8 Un singolo multiplexer viene ripetuto 32 volte per poter selezionare tra due ingressi di 32 bit ciascuno. Si noti che vi è un solo segnale di selezione per tutti i 32 multiplexer.

Nella logica di tipo sequenziale i segnali di *clock* sono necessari per determinare quando debbano essere aggiornati gli elementi che contengono informazioni di stato. Un segnale di clock non è altro che un segnale che evolve indipendentemente con un *periodo* (o *tempo di ciclo*) predeterminato e costante; la *frequenza di clock* è l'inverso del periodo. Come indicato in figura B.9, il *tempo di ciclo* o *periodo del clock* è diviso in due parti: la parte in cui il segnale di clock è alto e quella in cui è basso. In questo testo si utilizzeranno metodologie di temporizzazione esclusivamente di tipo *sensibile ai fronti* (*edge-sensitive*), in cui cioè tutte le variazioni di stato avvengono in corrispondenza di un fronte (*edge*) del segnale di clock. Tale metodologia è stata scelta perché più semplice da esporre, ma per alcune tecnologie potrebbe non essere quella più adatta.

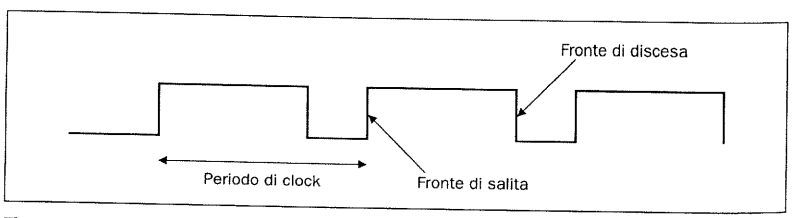


Figura B.9 Un segnale di clock oscilla tra i valori alto e basso. Il periodo del clock è il tempo necessario a compiere un ciclo completo. In un progetto sensibile ai fronti, il fronte attivo del clock, che causa le variazioni di stato, può essere o quello di salita o quello di discesa.

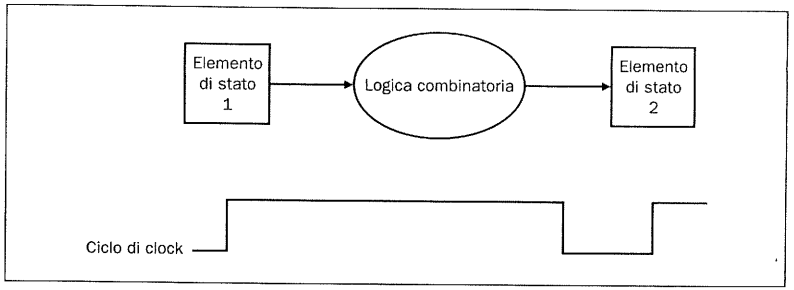


Figura B.10 Gli ingressi di un blocco di logica combinatoria provengono da un elemento di stato e le uscite sono scritte in un altro elemento di stato. Il fronte del clock determina quando viene aggiornato il contenuto degli elementi di stato.

Adottando una metodologia di temporizzazione sensibile ai fronti, uno dei due fronti (di salita o di discesa) è detto *attivo* e causa le variazioni di stato. Come si vedrà nel paragrafo seguente, gli elementi di stato nei progetti sensibili ai fronti sono progettati in modo tale che il loro contenuto può cambiare soltanto in corrispondenza del fronte attivo del clock. La scelta di quale fronte sia quello attivo è determinata dalla tecnologia implementativa e non ha effetti nella fase di progetto della logica.

Il vincolo principale in un sistema dotato di clock, detto anche *sistema sincrono*, è che i segnali scritti negli elementi di stato devono essere *validi* quando arriva il fronte attivo del clock. Un segnale si definisce valido se è stabile (ossia non sta variando) ed il suo valore non varierà fintanto che gli ingressi non cambieranno. Poiché i circuiti combinatori non possono contenere dei cicli di reazione, se gli ingressi di un blocco combinatorio non cambiano, prima o poi le sue uscite diverranno valide. La figura B.10 mostra la relazione tra gli elementi di stato ed i circuiti combinatori in un circuito logico sequenziale sincrono. Gli elementi di stato, le cui uscite cambiano solo in corrispondenza dei fronti del clock, forniscono ingressi validi ai blocchi di logica combinatoria. Per garantire che i valori scritti negli elementi di stato in corrispondenza del fronte attivo del clock siano validi, il clock deve avere un periodo abbastanza lungo da permettere a tutti i segnali della parte combinatoria di diventare stabili: questo vincolo determina il valore minimo della durata del periodo di clock. Nel seguito di questa appendice, così come nei Capitoli 5 e 6, non si indica esplicitamente il segnale di clock, poiché si ipotizza che tutti i segnali di stato vengano aggiornati in corrispondenza dello stesso fronte del clock. Alcuni elementi di stato verranno effettivamente aggiornati ad ogni fronte del clock, mentre altri dovranno essere scritti solo sotto certe condizioni (ad esempio nel caso di aggiornamento di un registro). In questi casi vi sarà un segnale di scrittura esplicito per tali elementi, il quale intercetterà il segnale di clock facendo sì che l'aggiornamento avvenga solo in corrispondenza dei fronti del clock nei quali esso è attivo. Si vedrà nel prossimo paragrafo come ciò è realizzato e come viene utilizzato.

Un ulteriore vantaggio della metodologia sensibile ai fronti consiste nella possibilità di avere elementi di stato usati sia come ingresso che come uscita dello stesso blocco di logica combinatoria, come mostrato dalla figura B.11. In pratica, in tali situazioni occorre fare in modo di evitare le condizioni critiche e occorre garantire che il periodo di clock sia abbastanza lungo; questo argomento verrà approfondito nel Paragrafo B.7.

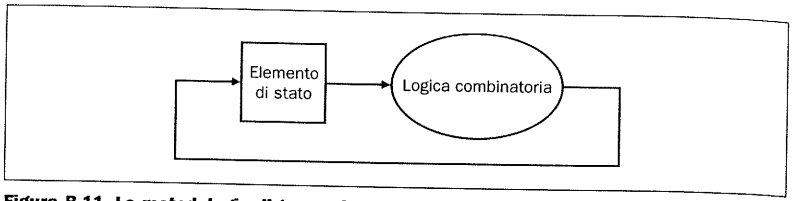


Figura B.11 La metodologia di temporizzazione sensibile ai fronti permette di leggere e scrivere lo stesso elemento di stato nello stesso ciclo di clock senza creare criticità che possano portare a valori indeterminati dei dati. Naturalmente il periodo di clock deve avere durata sufficiente a garantire che i valori di ingresso siano stabili al sopraggiungere del fronte attivo del clock.

Avendo discusso di come usare i segnali di clock per aggiornare gli elementi di stato, si passerà ora ad analizzare la realizzazione degli elementi di stato stessi.

B.5 Gli elementi di memoria

In questo paragrafo si discuteranno i principi fondamentali alla base del funzionamento degli elementi di memoria, a partire dai flip-flop e dai latch, per arrivare ai register file ed infine alle memorie. Tutti gli elementi di memoria memorizzano delle informazioni di stato: l'uscita di un elemento di memoria dipende sia dai suoi ingressi che dal valore memorizzato all'interno dell'elemento stesso; tutti i circuiti logici che contengono un elemento di memoria quindi contengono un'informazione di stato e sono di tipo sequenziale.

I tipi più semplici di elementi di memoria sono quelli *non temporizzati*, ossia non dotati di un ingresso di clock. Anche se in questo testo si useranno solo elementi di memoria temporizzati, i *latch* (letteralmente lucchetto) non temporizzati sono il tipo più semplice di elemento di memoria, per cui verranno analizzati per primi. La figura B.12 mostra un *latch S-R* (latch di tipo set-reset) costruito a partire da due porte NOR (porte OR con l'uscita invertita). Le uscite Q e $\bar{Q}$ rappresentano il valore dello stato memorizzato ed il suo complemento. Quando né S né R sono affermati, le porte NOR accoppiate agiscono come invertitori e memorizzano i valori precedenti di Q e $\bar{Q}$. Ad esempio se l'uscita Q è vera, l'invertitore inferiore produrrà un'uscita falsa (corrispondente a $\bar{Q}$), che a sua volta diviene ingresso per l'invertitore superiore, che fornisce un'uscita vera, corrispondente a Q, e così via. Se S viene affermato, Q verrà affermato e $\bar{Q}$ sarà negato, mentre se si afferma R l'uscita $\bar{Q}$ sarà affermata e Q negata. Se S ed R sono entrambi non affermati, gli ultimi valori di Q e $\bar{Q}$ continueranno ad essere memorizzati nella struttura grazie al doppio accoppiamento. Affermare contemporaneamente i due ingressi R e S può portare ad un funzionamento non corretto: a seconda del modo in cui S ed R vengono in un secondo momento riportati allo stato non affermato, il latch può oscillare oppure divenire metastabile (per una descrizione più approfondita si veda il Paragrafo B.7).

Questa struttura a doppio accoppiamento è alla base di elementi di memoria più complessi che permettono di memorizzare i dati. Tali elementi contengono porte logiche aggiuntive per memorizzare i valori dei segnali e far sì che il valore venga aggiornato solo in corrispondenza di un fronte di clock. Il paragrafo seguente esaminerà i dettagli costruttivi di questi elementi.

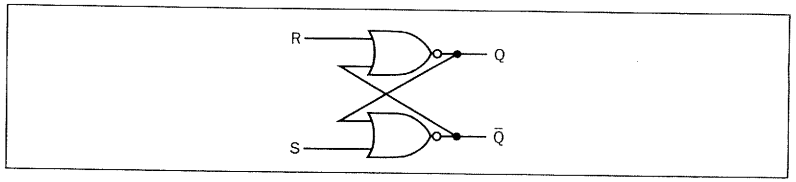


Figura B.12 Una coppia di porte NOR accoppiate può memorizzare un valore. Il valore memorizzato nell'uscita Q viene rigenerato invertendolo in modo da ottenere $\bar{Q}$ ed invertendo $\bar{Q}$ in modo da ottenere ancora Q. Se R oppure $\bar{Q}$ è affermato, Q diverrà non affermato, e viceversa.

Flip-flop e latch

I flip-flop ed i latch sono gli elementi di memoria più semplici: per entrambi, l'uscita è pari al valore dello stato memorizzato nell'elemento. Inoltre, a differenza del latch S-R appena descritto, tutti i latch ed i flip-flop che si utilizzeranno da qui in avanti saranno temporizzati, ossia avranno un ingresso di clock e la variazione di stato sarà attivata dal clock. La differenza tra flip-flop e latch è relativa all'istante del ciclo di clock che causa la variazione di stato. In un latch temporizzato, lo stato viene aggiornato ogniqualvolta cambiano gli ingressi ed il clock è affermato, mentre in un flip-flop lo stato può cambiare solo in corrispondenza di un fronte del clock. Poiché in tutto il testo si usa una metodologia di temporizzazione sensibile ai fronti, nella quale lo stato viene aggiornato solo in corrispondenza dei fronti del clock, sarà sufficiente usare esclusivamente dei flip-flop. I flip-flop sono spesso costruiti a partire da latch, per cui verrà inizialmente descritto il funzionamento di un semplice latch temporizzato e si estenderà in seguito la trattazione al funzionamento di un flip-flop costruito a partire da tale latch.

Nelle applicazioni legate ai calcolatori, la funzione di flip-flop e latch è di memorizzare un segnale. I *latch di tipo D* o i *flip-flop di tipo D* memorizzano nella memoria interna il valore del segnale di dato presente in ingresso. Anche se vi sono molti altri tipi di latch e flip-flop, il tipo D è l'unico elemento fondamentale usato in questo testo. Un latch di tipo D ha due ingressi e due uscite: gli ingressi sono il valore da memorizzare (chiamato D) ed il segnale di clock (chiamato C) che indica quando il latch deve leggere il valore sull'ingresso D e memorizzarlo. Le uscite sono semplicemente il valore dello stato interno (Q) ed il suo complemento ($\bar{Q}$). Quando l'ingresso di clock C è affermato, si dice che il latch è *aperto* ed il valore dell'uscita Q diviene il valore dell'ingresso D. Quando il clock C passa ad un valore non affermato, si dice che il latch è *chiuso* ed il valore dell'uscita Q rimane fissato all'ultimo valore che era stato memorizzato quanto il latch era aperto.

La figura B.13 mostra come sia possibile implementare un latch di tipo D con due porte logiche aggiunte alle porte NOR accoppiate; poiché quando il latch è aperto il valore di Q varia ogni volta che D varia, questa struttura è spesso detta *latch trasparente*. La figura B.14 illustra il funzionamento di questo latch di tipo D, assumendo che l'uscita Q sia inizialmente falsa e che D cambi per primo.

Come già detto, si utilizzeranno come blocchi fondamentali i flip-flop anziché i latch; i flip-flop non sono trasparenti: la loro uscita può cambiare *solo* in corrispondenza di un fronte del clock. Si può costruire un flip-flop in modo che venga attivato o sul fronte di salita (fronte positivo) o su quello di discesa (negativo) del clock; nei progetti di questo testo si possono usare indifferentemente i due tipi. La figura B.15 mostra la realizzazione di un flip-flop di tipo D sensibile al fronte di

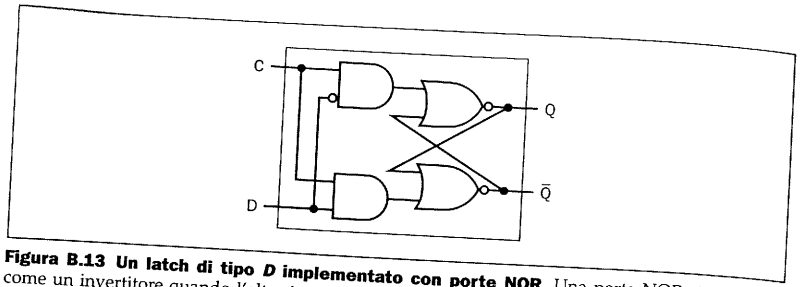


Figura B.13 Un latch di tipo *D* implementato con porte NOR. Una porta NOR si comporta come un invertitore quando l'altro ingresso vale 0, per cui le due porte NOR accoppiate si comportano in modo da memorizzare lo stato a meno che l'ingresso di clock *C* sia affermato, nel qual caso il valore dell'ingresso *D* sostituisce il valore di *Q* e viene quindi memorizzato. Il valore dell'ingresso *D* deve essere stabile quando il segnale di clock *C* passa dallo stato affermato a quello non affermato.

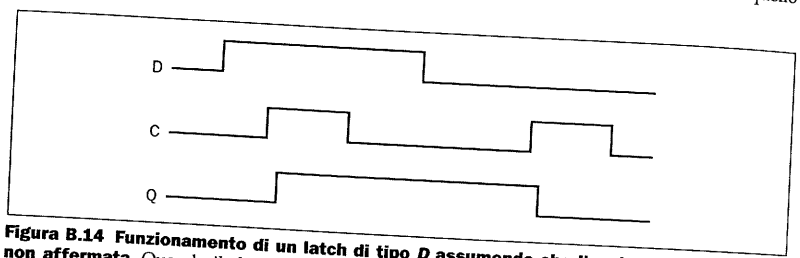


Figura B.14 Funzionamento di un latch di tipo *D* assumendo che l'uscita sia inizialmente non affermata. Quando il clock *C* è affermato, il latch è aperto e l'uscita *Q* assume immediatamente il valore dell'ingresso *D*.

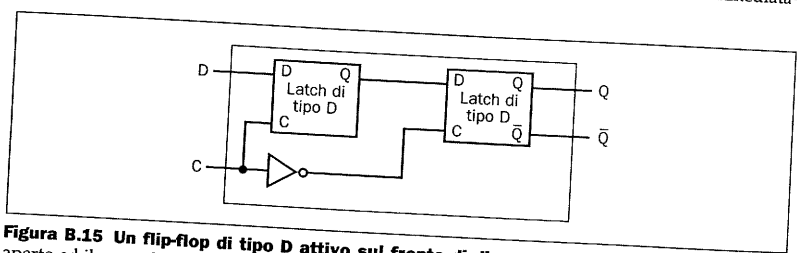


Figura B.15 Un flip-flop di tipo D attivo sul fronte di discesa. Il primo latch, detto *master*, è aperto ed il suo valore insegue l'ingresso *D* quando l'ingresso di clock *C* è affermato. Non appena l'ingresso di clock *C* scende, il primo latch viene chiuso, ma il secondo latch, detto *slave*, si apre ed assume il valore del suo ingresso, che proviene dall'uscita del latch master.

discesa a partire da due latch di tipo *D*. Nei flip-flop di tipo *D*, l'uscita viene memorizzata in corrispondenza del fronte del clock, come illustrato anche dalla figura B.16.

Poiché l'ingresso *D* è campionato ad ogni fronte del clock, deve rimanere valido in un opportuno intervallo di tempo immediatamente prima ed immediatamente dopo il fronte del clock. Il tempo minimo per cui l'ingresso deve rimanere valido prima del fronte del clock è detto *tempo di set-up* (letteralmente "tempo di preparazione"), mentre il tempo minimo per cui deve rimanere valido dopo il fronte del clock è detto *tempo di hold* (letteralmente "tempo di mantenimento"). In pratica gli ingressi di ogni flip-flop (e di ogni circuito costruito a partire da flip-flop) devono essere validi in una finestra temporale che inizia al tempo $t_{set\text{-}up}$ prima del fronte del

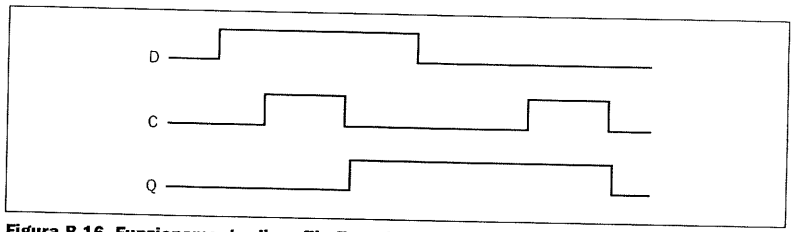


Figura B.16 Funzionamento di un flip-flop di tipo D attivo sul fronte di discesa, assumendo che l'uscita sia inizialmente non affermata. Quando l'ingresso di clock *C* da affermato diviene non affermato, l'uscita *Q* memorizza il valore dell'ingresso *D*. Si confronti questo comportamento con quello del latch temporizzato mostrato nella figura B.14: in un latch temporizzato il valore memorizzato e l'uscita *Q* variano ogniqualvolta *C* è alto, anziché solo quando *C* varia.

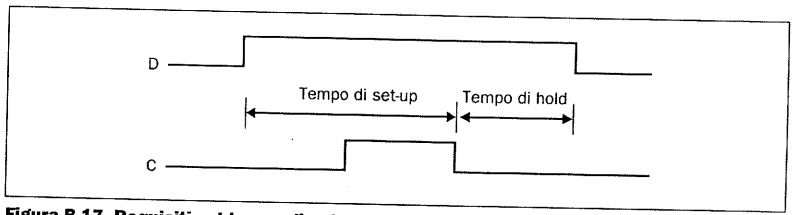


Figura B.17 Requisiti sul tempo di set-up e sul tempo di hold per un flip-flop di tipo D attivo sul fronte di discesa. L'ingresso deve rimanere stabile per un certo intervallo di tempo prima del fronte del clock e per un altro intervallo dopo il fronte. Il tempo minimo per cui il segnale deve rimanere stabile prima del fronte del clock è detto tempo di set-up, mentre il tempo minimo per cui il segnale deve rimanere stabile dopo il fronte del clock è detto tempo di hold. Se non si dovessero rispettare questi vincoli l'uscita del flip-flop potrebbe divenire imprevedibile, come descritto nel Paragrafo B.7. Il tempo di hold è solitamente pari a 0 oppure è molto breve e non è fonte di preoccupazioni.

clock e termina al tempo t_{hold} dopo il fronte del clock, come illustrato dalla figura B.17. Il Paragrafo B.7 approfondirà i concetti legati alla temporizzazione ed ai vincoli temporali.

È possibile usare una serie di flip-flop di tipo D per costruire un registro capace di memorizzare un dato su più bit, come un byte o una parola. Le unità di elaborazione dei Capitoli 5 e 6 usavano tali registri.

Register file

La struttura intorno alla quale è costruita l'unità di elaborazione è il *register file*: esso consta di un insieme di registri che possono essere letti o scritti fornendo il numero del registro a cui fare accesso. Un register file può essere realizzato mediante un decodificatore per ciascuna porta di lettura o scrittura ed una matrice di registri costruiti a partire da flip-flop di tipo D. Poiché la lettura di un registro non ne modifica lo stato, occorre solo fornire in ingresso il numero del registro ed in uscita si otterrà semplicemente il dato contenuto nel registro. Per una scrittura in un registro sono invece richiesti tre ingressi: il numero di registro, il dato da scrivere ed un segnale di clock che controlli l'operazione di scrittura. Nei Capitoli 5 e 6 si è usato un register file con due porte di lettura ed una di scrittura, disegnato come nella figura B.18. Le porte in lettura si possono implementare con due multiplexer, ciascuno dei quali di ampiezza pari al numero di bit nel register file. La figura B.19

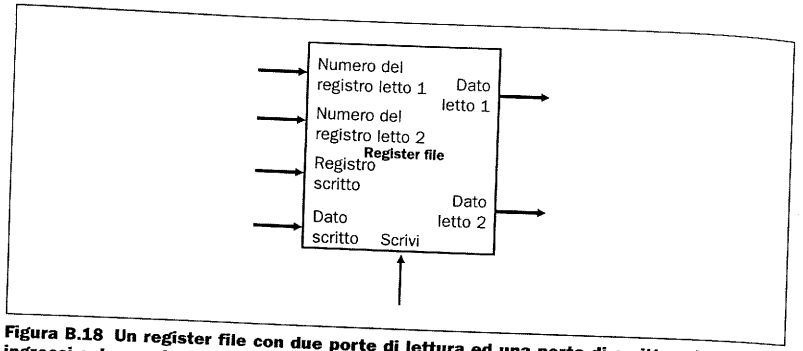


Figura B.18 Un register file con due porte di lettura ed una porta di scrittura ha quattro ingressi e due uscite. L'ingresso di controllo Scrivi è evidenziato in colore.

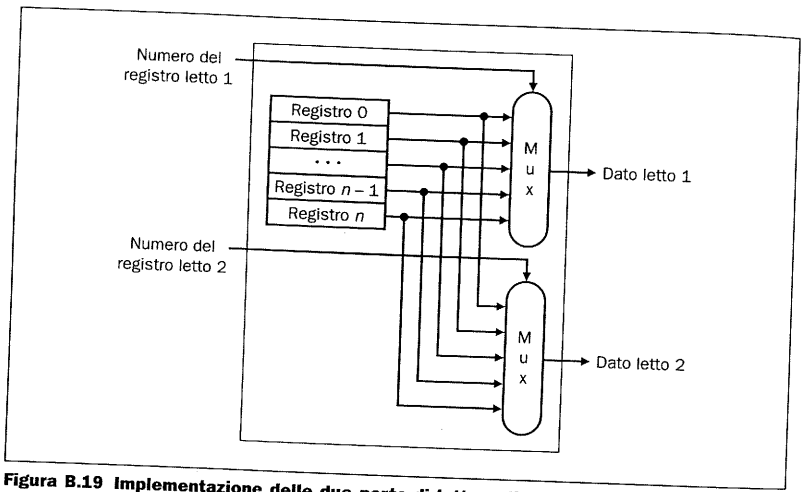


Figura B.19 Implementazione delle due porte di lettura di un register file con *n* registri mediante una coppia di multiplexer ad *n* vie di ampiezza 32 bit ciascuno. Il segnale con il numero del registro da leggere è usato come segnale di selezione del multiplexer. La figura B.20 indica come viene realizzata la porta di scrittura.

mostra l'implementazione delle due porte di lettura dei registri per un register file ampio 32 bit.

L'implementazione della porta di scrittura è lievemente più complessa poiché è permesso modificare solo il contenuto del registro designato: lo si può fare usando un decodificatore per generare un segnale in grado di determinare in quale registro scrivere. La figura B.20 mostra l'implementazione della porta di scrittura del register file. È importante ricordare che lo stato del flip-flop può cambiare solo in corrispondenza di un fronte del clock. Nei Capitoli 5 e 6 si erano indicati esplicitamente i segnali di scrittura dei registri, mentre si assumeva in modo implicito che il clock fosse connesso come mostrato in figura B.20.

Che cosa avviene se lo stesso registro viene letto e scritto nello stesso ciclo di clock? Poiché la scrittura del registro avviene in corrispondenza del fronte del

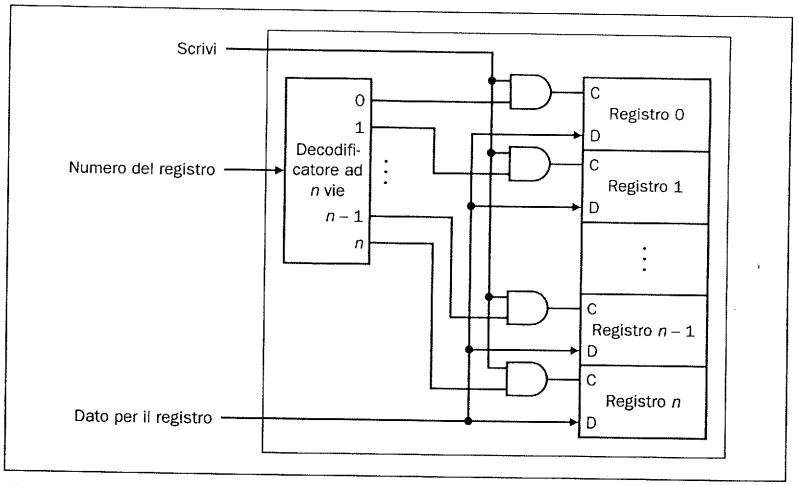


Figura B.20 La porta di scrittura del register file è implementata mediante un decodificatore che elabora il segnale di scrittura per generare l'ingresso *C* ai registri. I tre ingressi (il numero del registro, il dato ed il segnale di scrittura) hanno dei vincoli sui tempi di set-up e di hold per permettere il corretto caricamento del dato scritto nel register file.

clock, il registro sarà valido durante la sua lettura, come già visto nella figura B.10. Il valore letto sarà il valore scritto in un periodo di clock precedente. Se si volesse leggere il valore attualmente in corso di scrittura, sarebbe necessaria della logica aggiuntiva nel register file o al suo esterno. Il Capitolo 6 usa spesso della logica di questo tipo.

Memorie SRAM

I registri ed i register file sono i circuiti principali per costruire piccole memorie, ma quantità maggiori di memoria sono realizzabili solo ricorrendo alle *SRAM* (RAM, o memorie ad accesso casuale statiche) o alle *DRAM* (RAM dinamiche). Questo paragrafo tratta le SRAM, che in un certo senso sono più semplici, mentre il prossimo illustrerà le DRAM. Le SRAM sono semplicemente dei circuiti integrati che funzionano come matrici di memoria con (solitamente) una sola porta di accesso attraverso la quale effettuare operazioni di lettura o di scrittura. Le SRAM hanno lo stesso tempo di accesso per tutti i dati, anche se le caratteristiche dell'accesso in lettura e in scrittura spesso differiscono.

Ciascun chip di SRAM ha una specifica configurazione in termini di numero di locazioni indirizzabili e di ampiezza di ciascuna locazione. Ad esempio una SRAM 256K × 1 contiene 256K elementi, ciascuno dei quali ampio 1 bit; avrà quindi 18 linee di indirizzo (poiché $256K = 2^{18}$), una sola linea per il dato in uscita ed una sola linea di dato in ingresso. Una SRAM 32K × 8 ha lo stesso numero di bit complessivi, ma avrà 15 linee di indirizzo per indirizzare 32K elementi che contengono un dato di 8 bit ciascuno; vi saranno quindi otto uscite di dato ed otto ingressi di dato. Come per le ROM, il numero di locazioni indirizzabili è spesso chiamato *altezza* della memoria, mentre il numero di bit per locazione è detto *ampiezza*. Per diverse ragioni di tipo tecnologico le SRAM più recenti e veloci sono tipicamente disponi-

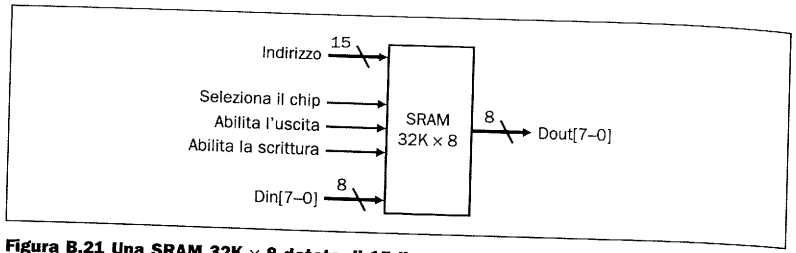


Figura B.21 Una SRAM 32K × 8 dotata di 15 linee di indirizzo (32K = 2^{15}) ed otto linee di dato in ingresso, tre linee di controllo ed otto linee di dato in uscita.

bili in configurazioni poco ampie: × 1 e × 2. La figura B.21 mostra i segnali di ingresso e di uscita per una SRAM 32K × 8.

Per attivare un accesso in lettura o scrittura, il segnale "Seleziona il chip" deve essere affermato. Per le letture occorre anche attivare il segnale "Abilita l'uscita" che controlla se il dato selezionato dalle linee di indirizzo è portato sui piedini di uscita o no. Il segnale "Abilita l'uscita" è utile per collegare più memorie allo stesso bus di uscita e poter determinare mediante tale segnale quale delle memorie debba pilotare il bus. Il tempo di accesso in lettura di una SRAM è solitamente specificato come ritardo tra l'istante di tempo in cui il segnale "Abilita l'uscita" diviene vero e le linee di indirizzo sono valide e l'istante di tempo in cui i dati compaiono sulle linee di uscita. I valori tipici per il tempo di accesso in lettura delle SRAM del 1997 sono compresi tra i 5 ns dei componenti CMOS più veloci ed i 25 ns dei componenti che, sebbene più lenti, sono spesso più economici ed hanno una densità maggiore. Le più grandi SRAM disponibili nel 1997 hanno oltre 4 milioni di bit di dato.

Per le operazioni di scrittura occorre fornire il dato da scrivere ed il relativo indirizzo, oltre ai segnali che causano l'esecuzione della scrittura. Quando i segnali "Abilita la scrittura" e "Seleziona il chip" sono entrambi veri, il dato presente sulle linee di dato in ingresso è scritto nella cella specificata dalle linee di indirizzo. Vi sono dei vincoli sul tempo di set-up e sul tempo di hold per le linee di indirizzo e di dato, esattamente come per i flip-flop di tipo D e per i latch; inoltre, il segnale "Abilita la scrittura" non è un fronte di clock bensì un impulso con un vincolo sulla sua durata minima. Il tempo necessario a completare un'operazione di scrittura è specificato dalla combinazione dei tempi di set-up, dei tempi di hold e dell'ampiezza dell'impulso sul segnale "Abilita la scrittura".

Le SRAM più grandi non si possono costruire con la stessa tecnica dei register file, poiché mentre in un register file un multiplexer a 32 vie può essere sensato, il multiplexer a 64K vie che sarebbe richiesto in una SRAM 64K × 1 sarebbe del tutto sproporzionato. Anziché utilizzare un multiplexer gigantesco le memorie più grandi sono realizzate mediante una linea di uscita condivisa, detta *linea di bit*, che può essere affermata da parte di molte celle di memoria della matrice di memoria; per permettere a più sorgenti di pilotare in singolo segnale si usano dei *buffer a tre stati*. Un buffer a tre stati ha due ingressi: un segnale di dato ed un segnale di abilitazione dell'uscita. L'unica uscita del buffer a tre stati ha lo stesso valore del segnale di ingresso se il segnale di abilitazione dell'uscita è affermato, altrimenti viene posta in uno stato di *alta impedenza* che permette ad un altro buffer a tre stati la cui uscita sia abilitata di determinare il valore dell'uscita condivisa. La figura B.22 mostra un gruppo di buffer a tre stati collegati in modo da realizzare la funzione di un multiplexer con ingresso non codificato; è estremamente importante che solo uno dei

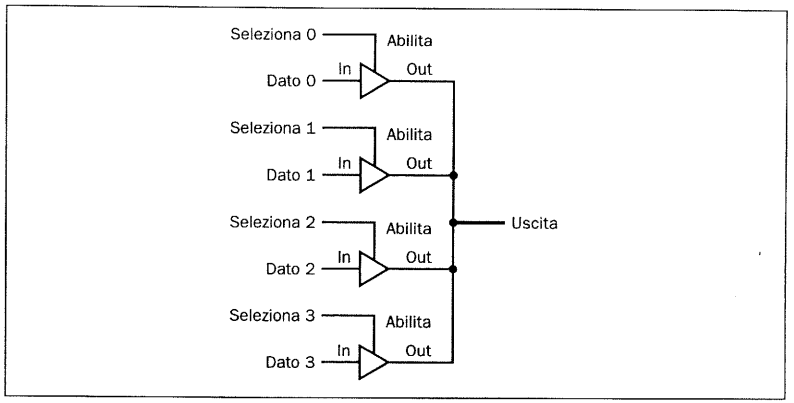


Figura B.22 Quattro buffer a tre stati connessi in modo da realizzare un multiplexer. Solo uno dei quattro ingressi di selezione può essere affermato. Un buffer a tre stati con segnale di abilitazione dell'uscita non affermato pone la propria uscita in alta impedenza in modo da permettere ad un altro buffer a tre stati, il cui segnale di abilitazione dell'uscita sia affermato, di pilotare la linea di uscita condivisa.

segnali di abilitazione dell'uscita dei tre multiplexer sia affermato, altrimenti diversi buffer potrebbero cercare di forzare valori diversi sulla linea di uscita. Utilizzando dei buffer a tre stati in ciascuna cella della SRAM, tutte le celle che corrispondono allo stesso segnale di uscita possono condividere la stessa linea. L'uso di un insieme distribuito di buffer a tre stati è un'implementazione più efficiente rispetto ad un grande multiplexer centralizzato. I buffer a tre stati sono integrati all'interno dei flip-flop che formano la cella elementare della SRAM. La figura B.23 mostra come costruire una piccola SRAM 4 × 2 utilizzando dei latch di tipo D dotati di un ingresso di abilitazione che controlla l'uscita a tre stati.

Il progetto di figura B.23 elimina la necessità di un multiplexer enorme, ma richiede in ogni caso un decodificatore molto grande ed un numero analogamente grande di linee di parola. Per esempio in una SRAM 16K × 8 servirebbe un decodificatore a 14 bit e 16K vie e sarebbero necessarie 16K linee di parola (che sono le linee usate per abilitare i singoli flip-flop)! Per aggirare questo problema le memorie più grandi sono organizzate come matrici rettangolari ed usano una tecnica di decodifica a due passi. La figura B.24 mostra come sarebbe possibile organizzare una SRAM 32K × 8 con una decodifica a due passi; come si vedrà, il processo di decodifica a due passi è notevolmente importante nella comprensione del funzionamento delle memorie DRAM.

Negli ultimi tempi si è assistito allo sviluppo sia di SRAM sincrone (SSRAM) che di DRAM sincrone (SDRAM): la funzionalità chiave fornita dalle RAM di tipo sincrono è di poter trasferire una sequenza di dati (detta *burst*) posta ad una serie di indirizzi consecutivi all'interno della stessa matrice o riga. Il "burst" è definito da un indirizzo di partenza, fornito nel solito modo, e da una lunghezza. La maggior velocità delle SRAM sincrone risiede nella capacità di trasferire i bit interni al burst senza dover fornire informazioni aggiuntive sugli indirizzi; si utilizza invece un segnale di clock per trasferire direttamente i bit successivi nel burst. L'eliminazione della necessità di specificare l'indirizzo degli accessi interni al burst migliora notevolmente la frequenza di trasferimento dei blocchi di dati: grazie a questa caratteri-

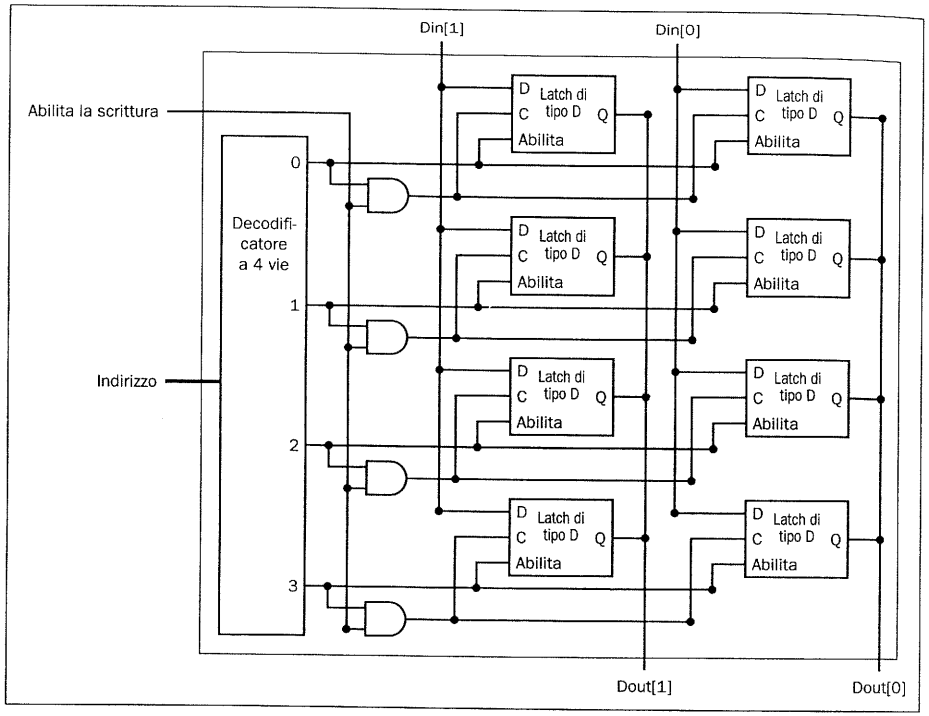


Figura B.23 La struttura fondamentale di una SRAM 4 × 2 consiste di un decodificatore che seleziona quale coppia di celle attivare. Le celle attivate usano un'uscita a tre stati collegata alle linee di bit verticali che forniscono il dato richiesto. L'indirizzo che seleziona la singola cella viene inviato ad una serie di linee di indirizzo verticali, dette linee di parola. Per semplicità i segnali di Abilita l'uscita e Seleziona il chip sono stati omessi, ma si sarebbero facilmente potuti aggiungere per mezzo di alcune porte AND.

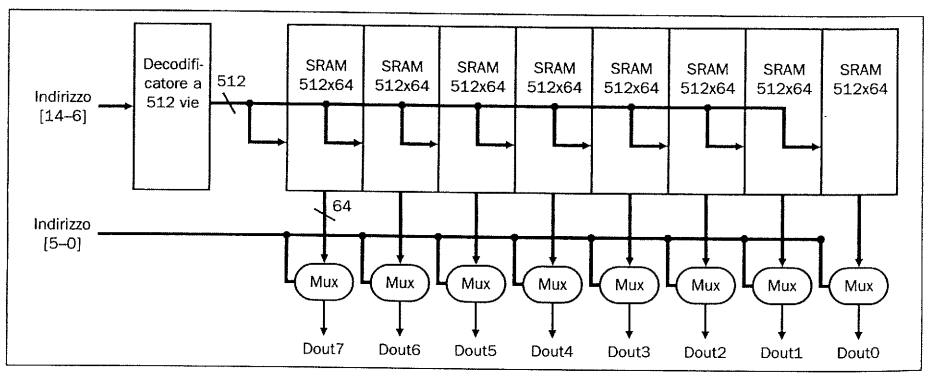


Figura B.24 Organizzazione tipica di una SRAM 32K × 8 come matrice di matrici 512 × 64. Il primo decodificatore genera gli indirizzi per otto matrici 512 × 64; un insieme di multiplexer viene poi usato per selezionare un bit da ciascuna matrice ampia 64 bit. Questo schema è molto più semplice da progettare rispetto ad una decodifica ad un solo livello, che richiederebbe un decodificatore enorme (da 15 a 32K) o un multiplexer gigantesco (da 32K ad 1).

stica le SRAM e le DRAM sincrone stanno rapidamente divenendo le memorie prescelte nella costruzione di sistemi basati su cache, che per loro natura trasferiscono i dati in blocchi.

Memorie DRAM

In una RAM statica (SRAM) il valore memorizzato in ciascuna cella è mantenuto da una coppia di porte invertenti e, fintantoché vi è tensione di alimentazione, il valore può rimanervi per un tempo indefinito. In una memoria RAM dinamica (DRAM), il valore presente nelle celle è invece memorizzato dalla carica contenuta in un condensatore. Si usa un solo transistor per accedere a questa carica memorizzata, sia per leggerne il valore, sia per sovrascriverlo con un nuovo valore di carica. Poiché le DRAM richiedono un solo transistor per bit di memoria, sono molto più dense ed hanno un minor costo per bit; per fare un confronto si consideri che le SRAM richiedono da quattro a sei transistor per bit. Nelle DRAM la carica è memorizzata in un condensatore, per cui non può permanere indefinitamente e deve essere periodicamente *rinfrescata*: questo è il motivo per cui la memoria è definita *dinamica*, anziché statica come nel caso delle SRAM. Per rinfrescare il valore di una cella, questa viene semplicemente letta e riscritta; la carica può poi permanere per diversi millisecondi, corrispondenti a circa un milione di periodi di clock. Oggigiorno chip singoli che fungono da controllori di memoria sono in grado di gestire la funzione del rinfresco in modo autonomo ed indipendente dal microprocessore. Se si dovessero leggere e riscrivere nella DRAM i singoli bit, nelle DRAM più grandi contenenti diversi megabyte si dovrebbero sempre eseguire operazioni di rinfresco, non lasciando tempo agli accessi normali. Fortunatamente anche le DRAM usano una struttura di decodifica a due livelli, per cui è possibile rinfrescare un'intera riga (che condivide la stessa linea di parola) con un ciclo di lettura immediatamente seguito da un ciclo di scrittura. Generalmente le operazioni di rinfresco consumano dall'1% al 2% dei cicli attivi della DRAM, lasciando il restante 98% o 99% dei cicli disponibili per letture e scritture di dati.

Approfondimento: Come può una DRAM leggere e scrivere il segnale memorizzato in una cella? Il transistor in ciascuna cella si comporta come un interruttore, anche detto *pass transistor*, che permette di accedere al valore memorizzato nel condensatore per le operazioni di lettura o di scrittura. La figura B.25 mostra l'aspetto delle celle a transistor singolo. Il pass transistor agisce come un interruttore: quando viene affermato il segnale sulla linea di parola l'interruttore è chiuso, ed il condensatore risulta connesso alla linea di bit. Se l'operazione richiesta è di scrittura, il valore da scrivere è posto sulla linea di bit: se il valore è 0, il condensatore verrà scaricato, mentre se è 1 verrà caricato. Le operazioni di lettura sono lieve-

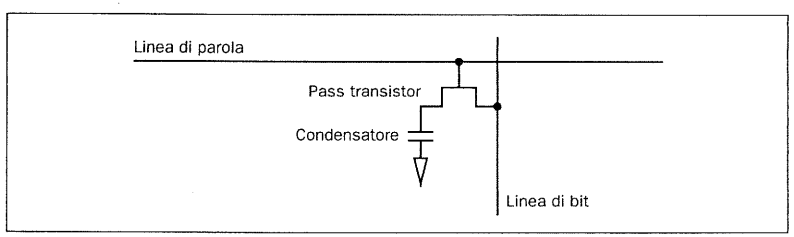


Figura B.25 Una cella di DRAM con un singolo transistor contiene un condensatore che memorizza il contenuto della cella ed un transistor per accedere alla cella stessa.

mente più complesse, poiché la DRAM deve essere in grado di rilevare una carica molto piccola memorizzata nel condensatore: prima di attivare la linea di parola la linea di bit viene caricata ad un valore di tensione a metà strada tra la tensione alta e quella bassa; in seguito all'attivazione della linea di parola, la carica del condensatore viene trasferita sulla linea di bit. Ciò causa una variazione della carica della linea di bit, che si sposta lievemente verso l'alto o verso il basso, e tale variazione è rilevata da un amplificatore (detto *sense amplifier*), in grado di rilevare piccole variazioni di tensione.

Le memorie DRAM usano un meccanismo di decodifica a due livelli come quello riportato in figura B.26, che si compone di un *accesso alla riga* seguito da un *accesso alla colonna*. Durante l'accesso alla riga si sceglie una riga tra quelle possibili e si attiva la linea di parola corrispondente; il contenuto di tutte le colonne della riga attiva è poi memorizzato in una serie di latch. Durante l'accesso alla colonna si selezionano poi i dati a partire dai latch delle colonne. Per ridurre il numero di piedini e per ridurre il costo del package, si usano le stesse linee di indirizzo per l'indirizzo di riga e per quello di colonna; per indicare alla DRAM che l'indirizzo fornito è relativo alle righe o alle colonne vi sono due segnali distinti, detti RAS (Row Access Strobe, comando di accesso alle righe) e CAS (Column Access Strobe, comando di accesso alle colonne). L'operazione di rinfresco è eseguita semplicemente leggendo i valori delle colonne nei relativi latch e riscrivendo lo stesso valore nelle celle: in questo modo si rinfresca un'intera riga in un solo ciclo. La tecnica di indirizzamento a due livelli, insieme alla circuiteria interna, fanno sì che il tempo di accesso alle DRAM sia molto più lungo (di un fattore compreso tra 5 e 10) rispetto al corrispondente tempo di accesso alle SRAM; nel 1997 i tempi di accesso tipici per le memorie DRAM si aggiravano tra i 60 ed i 110 ns. Il costo per bit decisamente inferiore rende le DRAM adatte ad essere usate come memoria principale, mentre il tempo di accesso ridotto delle SRAM fa sì che siano scelte per le cache.

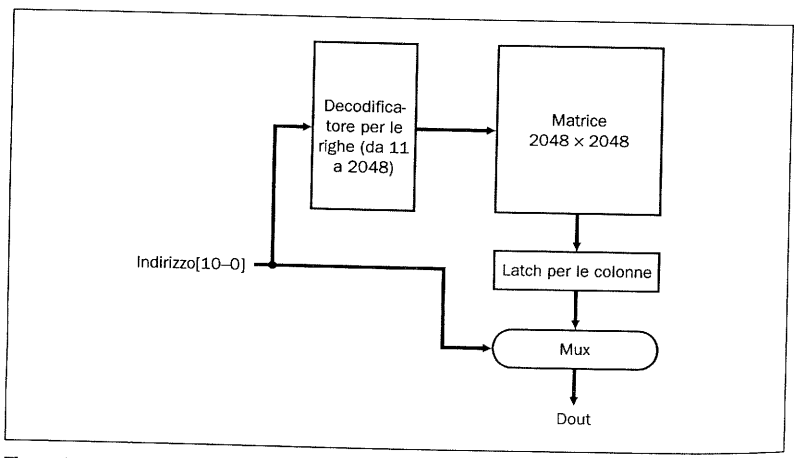


Figura B.26 Una memoria DRAM 4M × 1 costruita come matrice 2048 × 2048. L'accesso alle righe usa 11 bit per selezionare un'intera riga, il cui contenuto viene memorizzato in 2048 latch di 1 bit ciascuno. Il multiplexer seleziona un bit di uscita a partire dai 2048 latch. I segnali di controllo RAS e CAS determinano se i segnali di indirizzo siano inviati al decodificatore relativo alle righe o al multiplexer relativo alle colonne.

Si può osservare che una memoria DRAM 4M × 1, seleziona 2048 bit in ciascun accesso alle righe e subito dopo scarta 2047 di tali bit durante l'accesso alla colonna. I progettisti delle DRAM hanno cercato di usare la struttura interna della memoria per fornire una maggiore larghezza di banda in uscita: ciò è possibile permettendo di cambiare l'indirizzo di colonna senza cambiare l'indirizzo di riga, accedendo quindi agli altri bit presenti nei latch delle colonne. Le RAM dotate di *modalità di accesso a pagine* (*page-mode*) e di *modalità di accesso per colonne* (*static-column-mode*) permettono di accedere a più bit della stessa riga cambiando solo l'indirizzo di colonna (la differenza tra le due modalità risiede nel fatto che il segnale CAS debba essere riaffermato o meno). Le RAM con *modalità di accesso a nibble* (*nibble-mode*) generano internamente i tre indirizzi di colonna successivi, fornendo quindi 4 bit (detti *nibble*, letteralmente un "boccone") a ciascun accesso di riga. L'ultima versione delle memorie RAM che lavorano in modalità simile a quella a pagine è costituita dalle RAM EDO (Extended Data Out, uscita dati estesa); nel 1997 tali memorie sono divenute standard in molte applicazioni e sono in grado di fornire tempi di accesso che scendono fino a 25 ns. Nel 1997 si assiste anche alla diffusione delle memorie SDRAM: esse, come già detto, permettono accessi ancora più veloci ad una serie di bit della stessa riga eliminando la necessità di specificare l'indirizzo di colonna ed usando invece un segnale di clock per controllare il trasferimento di tutti i bit interni ad un burst. Come discusso nel Capitolo 7, tutti questi modi di funzionamento possono essere sfruttati per aumentare la larghezza di banda disponibile all'uscita della memoria principale in modo da soddisfare le necessità del processore e delle cache.

Correzione degli errori

Poiché è possibile che in memorie di grandi dimensioni i dati memorizzati si corrompano, la maggior parte dei sistemi di elaborazione usa qualche tipo di codice di rilevamento degli errori capace di rilevare gli eventuali dati corrotti. Un codice molto semplice che è frequentemente utilizzato è il *codice di parità*: secondo tale codice si conta il numero di bit pari ad 1 all'interno di una parola, e si dice che la parola ha parità dispari se il numero di 1 è dispari e parità pari in caso contrario. Quando si scrive una parola in memoria, viene scritto anche un bit che indica la parità (1 per la parità dispari, 0 per la parità pari); quando si legge una parola, si legge e si controlla il valore del bit di parità: se la parità della parola di memoria ed il bit di parità appena letto non coincidono, si è rilevato un errore. Una tecnica basata su un bit di parità può rilevare al massimo errori di un bit in ciascun dato: se vi fossero due bit errati, il codice di parità ad 1 bit non sarebbe in grado di rilevare l'errore, poiché la parità del dato iniziale e quella del dato con due errori sarebbero uguali (per la precisione, un codice basato su un bit di parità è in grado di rilevare errori su un numero dispari di bit; tuttavia, la probabilità che si verifichino tre errori è molto minore della probabilità di due errori, per cui in pratica un solo bit di parità è in grado di rilevare solo errori su un singolo bit). Per sua natura, un codice di parità non è in grado di indicare quale dei bit di dato sia errato.

I codici di parità ad un bit sono un esempio di codice per il rilevamento degli errori; vi sono anche dei *codici di correzione degli errori* (ECC, error-correcting codes) che sono in grado sia di rilevare che di permettere la correzione degli errori. Nel caso di memorie principali di grandi dimensioni, molti elaboratori usano dei codici che permettono di rilevare errori fino a 2 bit e di correggere un singolo bit errato. Tali codici richiedono di memorizzare diversi bit aggiuntivi per ciascun dato; ad

esempio i codici più utilizzati nelle memorie principali occupano 7 o 8 bit ogni 128 bit di dato.

Approfondimento: Un codice di parità di 1 bit è detto codice a *distanza* 2, poiché se si considerano insieme il dato ed il bit di parità non esiste alcuna modifica di un solo bit capace di generare un'altra combinazione valida di dato più bit di parità. Per esempio, se si modifica un bit nella parte di dato, la parità sarà sbagliata, e viceversa. Al contrario, se si modificano 2 bit (2 bit di dato qualsiasi oppure 1 bit di dato ed il bit di parità), l'informazione di parità sarà coerente con il valore del dato e l'errore non potrà essere rilevato. Vi è quindi una distanza pari a due tra ogni coppia di combinazioni lecite di dato e bit di parità.

Per rilevare più di un errore o per correggere degli errori, servono dei codici a *distanza 3*, che hanno la proprietà che ciascuna combinazione lecita di dato più bit di correzione dell'errore differisca di almeno 3 bit da tutte le altre combinazioni. Si supponga di disporre di un codice di questo tipo e si supponga che si verifichi un errore in un bit di dato: il tal caso la nuova combinazione di dato più codice avrà una distanza pari ad 1 bit rispetto ad una combinazione legale, perciò si può correggere l'errore. In presenza di due errori, si può riconoscere che vi è stato un errore, ma non lo si può correggere. Per comprendere meglio il funzionamento di tali codici conviene esaminare un esempio: la tabella seguente riporta le parole di dato ed i 3 bit del codice di correzione dell'errore per dati di 4 bit.

Dati	Codice	Dati	Codice
0000	000	1000	111
0001	011	1001	100
0010	101	1010	010
0011	110	1011	001
0100	110	1100	001
0101	101	1101	010
0110	011	1110	100
0111	000	1111	111

Si esamini ad esempio la parola di dato 0110, il cui codice di correzione dell'errore è 011: i quattro possibili errori di 1 bit sul dato sono: 1110, 0010, 0100 e 0111. Si esamini ora il dato che abbia lo stesso valore (011) del codice, ossia il dato 0001: se la logica di correzione dell'errore ricevesse uno dei quattro possibili valori errati, dovrebbe decidere se correggerlo come 0110 o come 0001. Si può notare che le quattro parole errate hanno un solo bit di differenza rispetto al dato 0110, mentre ne hanno due di differenza rispetto all'altra correzione possibile, 0001. Grazie a questa osservazione il meccanismo di correzione dell'errore può agevolmente scegliere la correzione 0110, poiché è molto più probabile un errore di un bit che di due. Per convincersi che il codice sarebbe stato in grado di rilevare gli errori doppi, si noti semplicemente che cambiando due bit di qualsiasi parola si ottiene una parola con un codice diverso; gli unici casi in cui i codici sono uguali sono in corrispondenza di differenze di 3 bit sui dati, ma se si tentasse di correggere un errore di 2 bit, si effettuerebbe una correzione errata, poiché la logica di correzione farebbe l'ipotesi errata che vi sia un solo bit errato. Se si volessero correggere gli errori di un bit e rilevare gli errori di 2 bit senza il rischio di correggerli erroneamente, sarebbe necessario un codice a distanza 4.

Sebbene nel capoverso precedente i dati siano stati trattati separatamente dai bit di codice, in realtà la logica di correzione degli errori tratta uniformemente la combinazione di dato e codice (di 7 bit nell'esempio), come una parola unica appartenente ad un codice più ampio. Ciò permette di gestire gli eventuali errori nei bit di codice nello stesso modo in cui si trattano i bit di dato.

Anche se nell'esempio precedente erano necessari $n - 1$ bit di codice per n bit di dato, il numero di bit richiesti cresce molto lentamente: ad esempio, nel caso di codici a distanza 3 servono 7 bit di codice per dati di 64 bit ed 8 bit di codice per dati di 128 bit. Questo tipo di codice è detto *codice di Hamming*, dal cognome di R. Hamming, che descrisse una metodologia per la creazione di tali codici.

B.6 Le macchine a stati finiti

Come si è visto in apertura di appendice, le reti logiche digitali possono essere classificate come combinatorie o sequenziali: i sistemi sequenziali contengono delle informazioni di stato memorizzate negli elementi di memoria interni al sistema. Il comportamento di un sistema sequenziale dipende sia dall'insieme di ingressi forniti al sistema che dal contenuto della memoria interna, che rappresenta lo stato del sistema. I sistemi sequenziali quindi non sono descrivibili mediante le tabelle di verità: occorre invece ricorrere a descrizioni basate su *macchine a stati finiti* (spesso dette solo *macchine a stati*). Una macchina a stati finiti è caratterizzata da un insieme di stati e da due funzioni dette *funzione di stato futuro* e *funzione di uscita*; l'insieme di stati corrisponde a tutti i possibili valori degli elementi di memoria interna, per cui se vi sono n bit di memoria vi saranno 2^n stati. La funzione di stato futuro è una funzione combinatoria che, dato il valore dell'ingresso e dello stato presente (o stato corrente), calcola il valore del prossimo stato assunto dal sistema; la funzione di uscita produce invece le uscite a partire dallo stato presente e dagli ingressi. La figura B.27 riporta in modo schematico i diversi elementi.

Le macchine a stati finiti trattate in questa appendice e nel Capitolo 5 sono di tipo *sincrono*, cioè lo stato cambia in corrispondenza dei cicli di clock, e si calcola un nuovo stato ad ogni nuovo ciclo di clock; in pratica gli elementi di stato sono aggiornati solo in corrispondenza dei fronti del clock. Tale metodologia è usata in tutto questo paragrafo e nei Capitoli 5 e 6, per cui non si indicherà esplicitamente il segnale di clock. Nel Capitoli 5 e 6 le macchine a stati finiti sono utilizzate per il controllo delle azioni del processore e dell'unità di elaborazione.

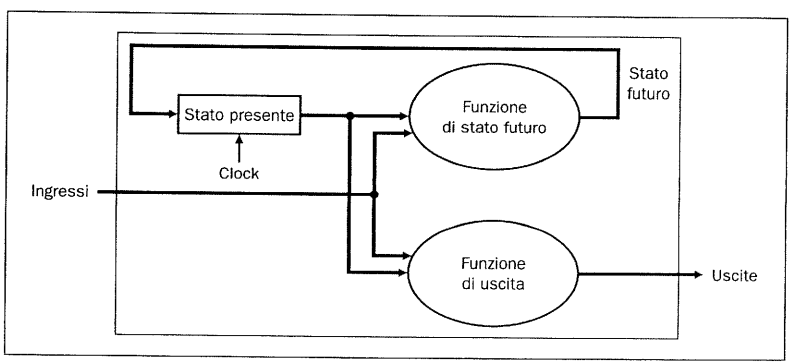


Figura B.27 Una macchina a stati consiste di elementi di memoria interni che memorizzano lo stato e di due funzioni combinatorie: la funzione di stato futuro e la funzione di uscita. Spesso si usa una versione più semplice della funzione di uscita, vincolandola a ricevere in ingresso solo il valore dello stato corrente; ciò non modifica le capacità del circuito sequenziale, ma ha influenza sulla sua struttura interna.

Per illustrare il funzionamento di una macchina a stati finiti e analizzarne le tecniche di progetto si utilizzerà un semplice esempio: il controllo di un semaforo (i Capitoli 5 e 6 contengono esempi più dettagliati dell'uso delle macchine a stati finiti nel controllo del processore). Quando si usa una macchina a stati finiti come controllore, la funzione di uscita viene spesso limitata in modo da dipendere solo dallo stato presente: macchine a stati finiti di tale tipo sono dette *macchine di Moore*, e saranno le sole che si utilizzeranno in questo testo. Se la funzione di uscita dipendesse sia dallo stato presente che dall'ingresso corrente, si direbbe che la macchina è una *macchina di Mealy*; i due tipi di macchine hanno capacità equivalenti, e si possono convertire l'una nell'altra seguendo un procedimento del tutto meccanico. Il vantaggio fondamentale delle macchine di Moore è di essere più veloci, mentre le macchine di Mealy possono rivelarsi più piccole, poiché possono richiedere un numero di stati minore rispetto alle corrispondenti macchine di Moore. La differenza è approfondita nel Capitolo 5.

L'esempio che verrà sviluppato è relativo al controllo di un semaforo posto all'incrocio di una strada in direzione nord-sud con una strada in direzione est-ovest; per semplicità si considereranno sono le luci verdi e rosse, mentre l'aggiunta del giallo è lasciata al lettore come semplice esercizio. Si vuole che le luci non cambino più velocemente di una volta ogni 30 secondi in entrambi i versi, per cui si userà un clock di 0,033 Hz in modo che la macchina cambi stato al più una volta ogni 30 secondi. Vi sono due segnali di uscita:

- *luceNS*: Quando si afferma questo segnale, viene accesa la luce verde sulla strada nord-sud; quando il segnale non è affermato, viene accesa la luce rossa sulla strada nord-sud.
- *luceEO*: Quando si afferma questo segnale, viene accesa la luce verde sulla strada est-ovest; quando il segnale non è affermato, viene accesa la luce rossa sulla strada est-ovest.

Vi sono inoltre due ingressi:

- *autoNS*: Indica che vi è un'auto in corrispondenza del rilevatore posto sotto l'asfalto di fronte al semaforo sulla strada nord-sud (sia in direzione nord che in direzione sud).
- *autoEO*: Indica che vi è un'auto in corrispondenza del rilevatore posto sotto l'asfalto di fronte al semaforo sulla strada est-ovest (sia in direzione est che in direzione ovest).

Il semaforo deve modificare la direzione nella quale viene presentato il verde solo se vi è un'auto in attesa su tale strada; altrimenti il semaforo deve continuare a mostrare il verde sulla stessa strada da cui proveniva l'ultima auto che è passata nell'incrocio.

Per implementare questo semplice semaforo sono necessari due stati:

- *verdeNS*: Il semaforo è verde nella direzione nord-sud.
- *verdeEO*: Il semaforo è verde nella direzione est-ovest.

È altresì necessario definire la funzione di stato futuro, che può essere specificata per mezzo di una tabella come la seguente.

Stato presente	Ingressi		Stato futuro
	autoNS	autoEO	
verdeNS	0	0	verdeNS
verdeNS	0	1	verdeEO
verdeNS	1	0	verdeNS
verdeNS	1	1	verdeEO
verdeEO	0	0	verdeEO
verdeEO	0	1	verdeEO
verdeEO	1	0	verdeNS
verdeEO	1	1	verdeNS

Si noti che la specifica non indicava cosa fare nel caso in cui vi fossero delle auto in entrambe le direzioni: in questo caso la funzione di stato futuro sopra riportata modifica lo stato in modo da garantire che un flusso continuo di auto in una direzione non possa bloccare le auto nell'altra direzione.

La macchina a stati finiti risulta completamente specificata indicando anche la funzione di uscita:

Stato presente	Uscite	
	luceNS	luceEO
verdeNS	1	0
verdeEO	0	1

Prima di passare all'implementazione, conviene esaminare una rappresentazione grafica, molto usata per rappresentare le macchine a stati finiti. In questa rappresentazione si usano i nodi per indicare gli stati; all'interno dei nodi si riporta l'elenco delle uscite affermate in tale stato. Si usano degli archi orientati per indicare la funzione di stato futuro, in cui le etichette sugli archi specificano le condizioni di ingresso mediante funzioni logiche. La rappresentazione grafica della macchina a stati finiti appena vista è riportata in figura B.28.

Una macchina a stati finiti può essere implementata mediante un registro per memorizzare lo stato presente ed un circuito logico combinatorio per calcolare le funzioni di stato futuro e di uscita; la figura B.29 mostra l'aspetto di un'ipotetica macchina a stati finiti con 4 bit di stato, corrispondenti a 16 stati. Per implementare

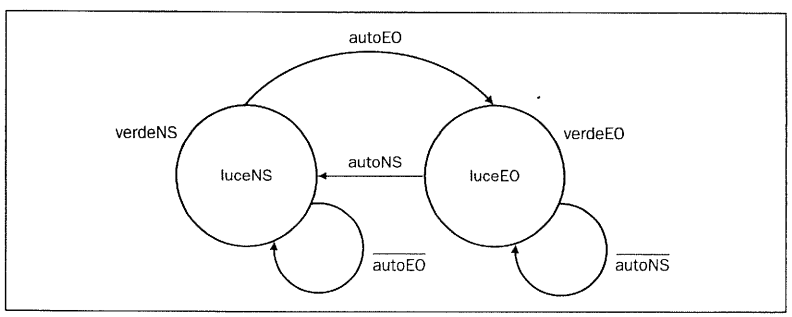


Figura B.28 Rappresentazione grafica del controllore di semaforo a due stati. Le funzioni logiche relative alle transizioni di stato sono state semplificate; ad esempio, la transizione da verdeNS a verdeEO si può ricavare dalla tabella degli stati futuri come ($\overline{\text{autoNS}}$ · autoEO) + autoNS · autoEO) equivalente ad autoEO.

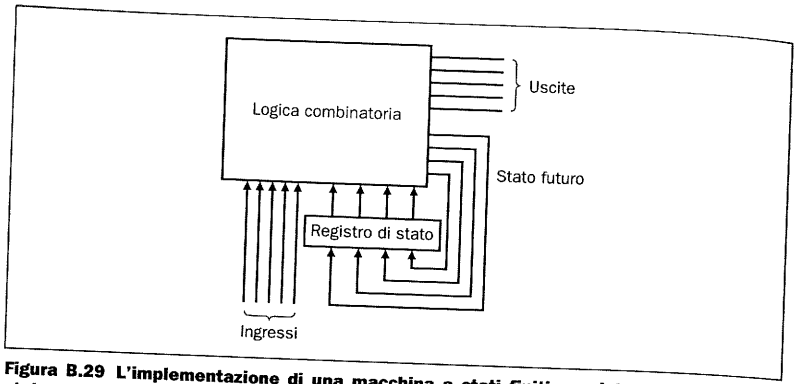


Figura B.29 L'implementazione di una macchina a stati finiti consiste di un registro di stato che contiene lo stato presente e di un circuito logico combinatorio che calcola le funzioni di stato futuro e di uscita. Le due funzioni sono spesso divise ed implementate da due circuiti logici separati, il che spesso fa risparmiare porte logiche.

in tal modo la macchina a stati finiti occorre innanzitutto assegnare dei valori numerici agli stati, mediante un processo che prende il nome di *codifica degli stati* o *assegnazione degli stati*. Per esempio si potrebbe assegnare lo stato 0 a verdeNS e lo stato 1 a verdeEO, ottenendo un registro di stato di un solo bit. La funzione di stato futuro diverrà quindi

$$\text{stato futuro} = (\overline{\text{stato presente}} \cdot \text{autoEO}) + (\text{stato presente} \cdot \overline{\text{autoNS}})$$

dove lo stato presente è il contenuto del registro di stato (0 o 1) e lo stato futuro è il valore dell'uscita della funzione di stato futuro, che verrà scritto nel registro di stato al termine del ciclo di clock. La funzione di uscita è altrettanto semplice:

$$\text{luceNS} = \overline{\text{stato presente}}$$

$$\text{luceEO} = \text{stato presente}$$

Il circuito logico combinatorio è spesso implementato ricorrendo a logica strutturata, ad esempio una PLA, che può essere costruita automaticamente a partire dalle tabelle delle funzioni di stato futuro e di uscita. Vi sono infatti programmi di progettazione assistita da calcolatore (CAD, computer-aided design) che partono da una rappresentazione grafica o testuale di una macchina a stati finiti e producono automaticamente un'implementazione ottimizzata. Nei Capitoli 5 e 6 le macchine a stati finiti sono utilizzate nelle unità di controllo dell'esecuzione del processore; l'Appendice C analizza in dettaglio l'implementazione di tali unità di controllo sia mediante PLA sia mediante ROM.

B.7 Metodologie di temporizzazione

In tutta l'appendice e nel resto del testo si utilizza una metodologia di temporizzazione sensibile ai fronti: essa ha il vantaggio di essere più semplice da illustrare e

comprendere rispetto ad una metodologia sensibile ai livelli. In questo paragrafo si illustrerà in maggior dettaglio questa metodologia di temporizzazione e si accennerà brevemente alla temporizzazione sensibile ai livelli; il paragrafo si concluderà con una breve discussione sui segnali asincroni e sui circuiti di sincronizzazione, un problema importante per i progettisti di circuiti digitali.

L'obiettivo di questo paragrafo è di introdurre i concetti più importanti sulle metodologie di temporizzazione; si faranno diverse importanti ipotesi semplificative, per cui i lettori interessati a comprendere meglio le metodologie di temporizzazione sono invitati a consultare i testi citati al termine di questa appendice.

Finora si è usata una metodologia di temporizzazione sensibile ai fronti poiché è più semplice da spiegare e richiede meno regole per funzionare correttamente. In particolare se si assume che tutti i segnali di clock arrivino nello stesso istante di tempo, è garantito che un sistema composto di registri sensibili ai fronti separati da circuiti logici combinatori funzioni correttamente, senza presentare condizioni critiche, a patto di usare un periodo di clock di durata sufficientemente lunga. Si verifica una *condizione critica* (o *corsa critica*, in inglese, *race*) quando il contenuto di un elemento di stato dipende dalla velocità relativa di diversi elementi logici. Nei progetti sensibili ai fronti il periodo di clock deve essere abbastanza lungo da contenere il ritardo sul cammino da un flip-flop ad un altro attraverso la logica combinatoria, oltre a garantire il requisito sul tempo di set-up. La figura B.30 mostra questo requisito per un sistema che usa flip-flop sensibili ai fronti; in tale sistema il periodo di clock deve essere almeno pari a

$$t_{\text{prop}} + t_{\text{combinatorio}} + t_{\text{set-up}}$$

usando per i tre ritardi il valore nel caso peggiore. Per semplicità si assume che il requisito sul tempo di hold sia soddisfatto; nella maggior parte dei progetti non è difficile soddisfare il requisito sul tempo di hold, poiché il tempo di propagazione (t_{prop}) è sempre maggiore del tempo di hold dei flip-flop.

Nei progetti sensibili ai fronti occorre tenere conto di una complicazione aggiuntiva: lo sfasamento del clock (detto *clock skew*), definito come la differenza tra gli istanti di tempo assoluti nei quali due elementi di stato ricevono il fronte del clock. Lo sfasamento del clock nasce per via dei diversi cammini seguiti dal segnale di

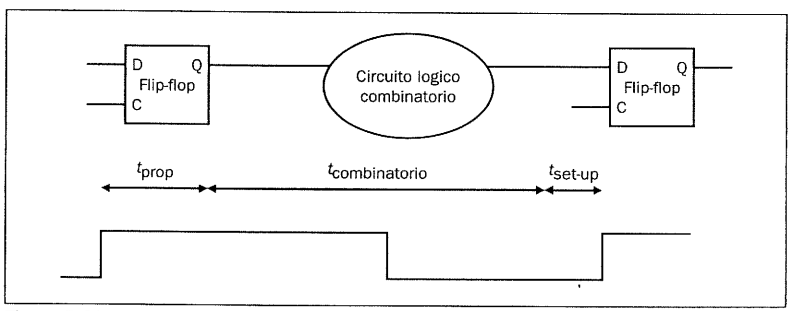


Figura B.30 Nei progetti sensibili ai fronti il clock deve essere abbastanza lento da permettere ai segnali di rimanere validi per tutto il tempo di set-up prima che arrivi il fronte successivo del clock. Il tempo richiesto ad un flip-flop per propagare sull'uscita il valore del proprio ingresso è t_{prop}; il segnale impiega un tempo $t_{\text{combinatorio}}$ per viaggiare attraverso la logica combinatoria e deve essere valido almeno $t_{\text{set-up}}$ prima del successivo fronte di clock.

clock per raggiungere gli elementi di stato, che possono introdurre un ritardo diverso. Se lo sfasamento è abbastanza grande può essere possibile che un elemento di stato cambi e causi una variazione sull'ingresso di un altro flip-flop prima che il secondo flip-flop riceva il fronte del clock; tale problema è illustrato dalla figura B.31, in cui si ignorano il tempo di set-up ed il ritardo di propagazione del flip-flop. Per evitare un funzionamento errato occorre allungare il periodo di clock di una quantità pari al massimo sfasamento del clock; il periodo di clock dovrà quindi essere maggiore di

$$t_{prop} + t_{combinatorio} + t_{set\text{-}up} + t_{skew}$$

Imponendo questo vincolo al periodo di clock, i due clock possono anche giungere nell'ordine sbagliato agli elementi di memoria, con il secondo clock in anticipo di t_{skew} sul primo, ed il circuito si comporterà correttamente. I progettisti cercano di ridurre i problemi di sfasamento mediante un attento instradamento del segnale di clock per minimizzare la differenza tra gli istanti di arrivo. In aggiunta a ciò, i progettisti più attenti forniscono un margine di sicurezza scegliendo un periodo di clock lievemente più lungo del valore minimo; in tal modo si tiene conto delle variazioni nelle caratteristiche dei componenti e nella tensione di alimentazione. Poiché lo sfasamento del clock può avere effetti sul vincolo sul tempo di hold, è importante ridurlo il più possibile.

I progetti sensibili ai fronti hanno due difetti: richiedono della circuiteria aggiuntiva e possono risultare più lenti. Un semplice confronto del flip-flop di tipo D con il latch sensibile ai livelli usato nella costruzione dei flip-flop indica che i progetti sensibili ai fronti richiedono una quantità maggiore di porte logiche. In alternativa si potrebbe usare una *temporizzazione sensibile ai livelli*: poiché la variazione di stato nella metodologia sensibile ai livelli non è istantanea, questa tecnica si rivela un po' più complessa e richiede una maggiore attenzione per garantirne il corretto funzionamento.

Temporizzazione sensibile ai livelli

In una metodologia di temporizzazione sensibile ai livelli le variazioni di stato avvengono in corrispondenza dei livelli alti o di quelli bassi, ma non sono istantanee come nella metodologia sensibile ai fronti; per via di questa non istantaneità si possono verificare delle condizioni critiche. Per garantire che un progetto sensibile ai livelli funzioni correttamente con clock sufficientemente lenti, i progettisti usano una *temporizzazione a due fasi*: si tratta di una tecnica che fa uso di due segnali di clock non sovrapposti. Poiché i due clock, spesso chiamati ϕ_1 e ϕ_2, non si sovrap-

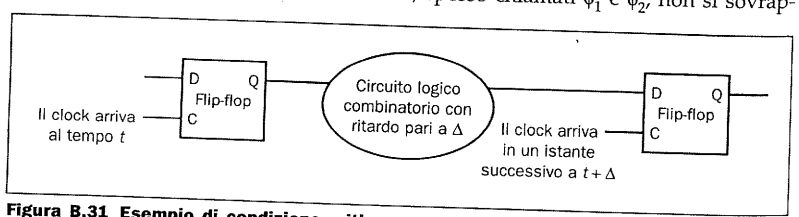


Figura B.31 Esempio di condizione critica causata dallo sfasamento (skew) del clock, che porta ad un funzionamento erroneo. A causa della differenza tra gli istanti di tempo in cui i due flip-flop vedono il fronte del clock, il segnale memorizzato nel primo flip-flop potrebbe propagarsi velocemente e modificare l'ingresso del secondo flip-flop prima dell'arrivo del clock.

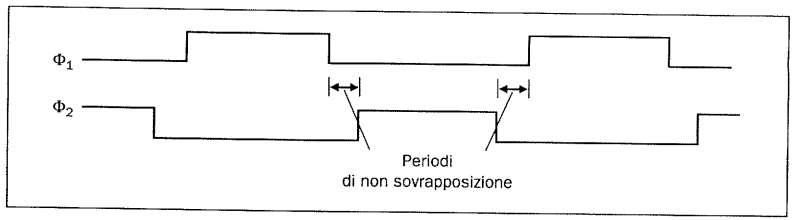


Figura B.32 Una tecnica di temporizzazione a due fasi in cui sono indicati i cicli dei due clock ed i periodi di non sovrapposizione.

pongono, in un qualsiasi istante al più uno dei due segnali sarà alto, come mostrato in figura B.32. Si possono usare questi due segnali di clock per costruire un sistema basato su latch sensibili ai livelli che sia privo di condizioni critiche, esattamente come lo erano i progetti sensibili ai fronti.

Un modo semplice di progettare un sistema di questo tipo è di alternare l'uso di latch che vengono aperti dal clock ϕ_1 con quelli aperti da ϕ_2; poiché i due clock non sono mai affermati insieme, non possono verificarsi delle condizioni critiche. Se l'ingresso di un blocco combinatorio proviene da un clock ϕ_1, l'uscita sarà inviata ad un latch pilotato da ϕ_2, che è aperto solo durante la parte attiva di ϕ_2 nella quale il latch sull'ingresso è chiuso e fornisce quindi in uscita un valore valido. La figura B.33 illustra il funzionamento di un sistema con temporizzazione a due fasi e latch alternati. Come per i progetti sensibili ai livelli, occorre tenere conto dello sfasamento dei clock, in particolare tra fasi diverse; aumentando la durata del periodo di non sovrapposizione tra le due fasi si può ridurre la possibilità di errori. Il sistema dunque funziona correttamente se ciascuna fase ha una durata sufficiente e se vi è un intervallo di non sovrapposizione sufficientemente ampio tra le fasi.

Ingressi asincroni e circuiti di sincronizzazione

Con l'uso di un clock unico o di un clock a due fasi si possono eliminare le condizioni critiche se si evitano i problemi di sfasamento del clock; sfortunatamente è molto difficile far funzionare un intero calcolatore con un solo clock ed allo stesso tempo ridurre lo sfasamento del clock. Mentre la CPU potrebbe usare un solo clock, probabilmente ciascun dispositivo di I/O userà il proprio; nel Capitolo 8 si è visto come un dispositivo asincrono possa comunicare con la CPU mediante una serie di passi di handshaking. Per tradurre l'ingresso asincrono in un segnale sincrono che

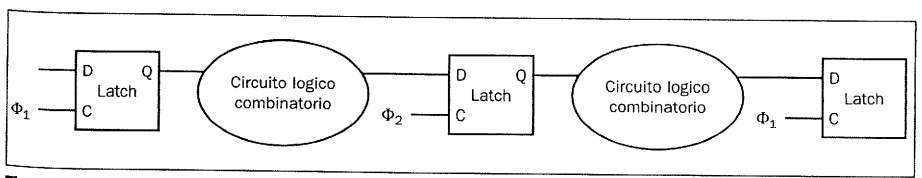


Figura B.33 Una tecnica di temporizzazione a due fasi con latch alternati in cui il sistema lavora sulle due fasi dei clock. L'uscita di un latch è stabile sulla fase contraria a quella del suo ingresso C, per cui il primo circuito combinatorio ha un ingresso stabile durante ϕ_2 e la sua uscita viene catturata da ϕ_2. Analogamente il secondo circuito combinatorio (a destra) lavora in modo complementare, con ingressi stabili durante ϕ_1. I ritardi dei circuiti combinatori determinano quindi il tempo minimo per cui i relativi clock devono rimanere affermati, mentre la durata del periodo di non sovrapposizione è determinata dal massimo sfasamento del clock e dal ritardo minimo dei diversi circuiti combinatori.

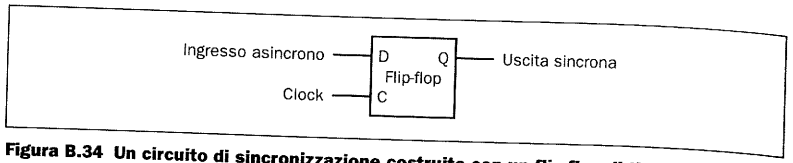


Figura B.34 Un circuito di sincronizzazione costruito con un flip-flop di tipo D viene usato per campionare un segnale asincrono producendo un'uscita sincrona con il segnale di clock. Questo "sincronizzatore" *non* funziona correttamente!

possa essere usato per determinare le variazioni di stato di un sistema è necessario un *circuito di sincronizzazione*, i cui ingressi sono il segnale asincrono ed un segnale di clock e la cui uscita è un segnale sincronizzato con il clock fornito in ingresso.

In un primo tentativo di costruzione di un circuito di sincronizzazione si potrebbe far uso di un flip-flop di tipo D sensibile ai fronti il cui ingresso *D* sia collegato al segnale asincrono, come indicato in figura B.34. Poiché si comunica mediante un protocollo di handshaking (come riportato nel Capitolo 8), non importa se si rivela lo stato affermato del segnale asincrono in un ciclo di clock o nel successivo, poiché il segnale verrà mantenuto affermato fino a quando non giungerà la conferma della sua ricezione. Si può quindi pensare che questa semplice struttura sia sufficiente a campionare con precisione il segnale, il che corrisponde a verità eccezion fatta per un piccolo problema.

Il problema consiste in una condizione detta *metastabilità*; si ipotizzi che il segnale asincrono stia variando tra il livello alto e quello basso mentre arriva il segnale di clock: è ovviamente impossibile sapere se il segnale verrà memorizzato come alto o basso, ma questo fatto non costituisce un problema. Sfortunatamente la situazione reale è peggiore: quando il segnale che viene campionato non è stabile durante i necessari periodi di set-up e di hold è possibile che il flip-flop si porti in uno stato *metastabile*. In tale stato, l'uscita non avrà un valore corretto (alto o basso), ma sarà nella regione indeterminata tra i due valori; inoltre non vi è garanzia che il flip-flop esca da questo stato in un un intervallo finito di tempo. Alcuni dei circuiti che leggono il valore presente sull'uscita del flip-flop lo interpreteranno come uno 0, altri come un 1: questa situazione è detta *errore di sincronizzazione*. In un sistema completamente sincrono gli errori di sincronizzazione si possono evitare garantendo il rispetto dei tempi di set-up e di hold per tutti i flip-flop o latch, ma ciò è impossibile quando il segnale in ingresso è di natura asincrona. Inoltre l'unica soluzione possibile è attendere abbastanza a lungo prima di leggere l'uscita del flip-flop per garantire che la sua uscita sia stabile, e che abbia abbandonato lo stato metastabile, se mai vi è entrata. Qual è il periodo di tempo necessario? Si può dimostrare che la probabilità che un flip-flop sia in uno stato metastabile decresce esponenzialmente, per cui dopo poco tempo la probabilità che il flip-flop sia nello stato metastabile diviene estremamente bassa; tuttavia tale probabilità non raggiunge mai il valore 0! I progettisti aspettano un periodo di tempo tale da rendere abbastanza piccola la probabilità di un errore di sincronizzazione, e tali errori si possono verificare a distanza di anni o migliaia di anni. Nella maggior parte dei flip-flop un'attesa di un tempo pari ad alcune volte il tempo di set-up rende abbastanza piccola la probabilità di errori di sincronizzazione. Se il periodo di clock è più lungo del periodo di potenziale metastabilità (condizione abbastanza probabile), un circuito di sincronizzazione sicuro si può costruire con due flip-flop di tipo D, come indicato in figura B.35. I lettori interessati ad un maggior approfondimento relativo a questi problemi sono invitati a consultare i riferimenti bibliografici.

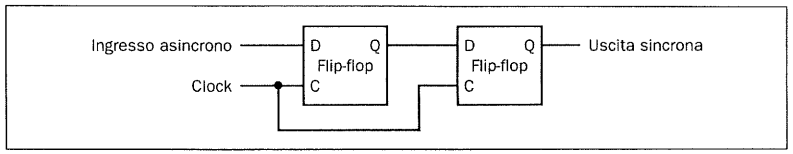


Figura B.35 Questo circuito di sincronizzazione funziona correttamente se il periodo di metastabilità che si vuole evitare è inferiore al periodo di clock. Sebbene l'uscita del primo flip-flop possa essere metastabile, essa non viene letta da alcun elemento logico prima del secondo ciclo di clock, quando il secondo flip-flop campiona il segnale, che in quell'istante non dovrebbe più essere nello stato metastabile.

B.8 Note conclusive

Questa appendice ha introdotto le nozioni fondamentali sulla progettazione logica; con la conoscenza di queste nozioni è possibile affrontare il materiale presentato nei Capitoli 5 e 6, i quali utilizzano ampiamente i concetti qui discussi.

Letture consigliate

Katz, R. H. [1993]. *Modern Logic Design*, Benjamin/Cummings, Redwood City, CA.
Un testo sulla progettazione di circuiti digitali.

McCluskey, E. J. [1986]. *Logic Design Principles*, Prentice Hall, Englewood Cliffs, NJ.
Contiene discussioni approfondite sulle condizioni critiche, sui principi di ottimizzazione e sulla collaudabilità.

Mead, C., and L. Conway [1980]. *Introduction to VLSI Systems*, Addison-Wesley, New York.
Tratta il progetto di sistemi VLSI usando la tecnologia nMOS.

Proser, F. P., and D. E. Winkel [1987]. *The Art of Digital Design*, second edition, Prentice Hall, Englewood Cliffs, NJ.
Un testo sulla progettazione di circuiti digitali.

Wakerly, J. F. [1990]. *Digital Design: Principles and Practices*, Prentice Hall, Englewood Cliffs, NJ.
Un testo sulla progettazione di circuiti digitali.

B.9 Parole chiave

Questa appendice ha introdotto un certo numero di termini importanti che potrebbero risultare nuovi; tali termini chiave sono elencati di seguito; il lettore che abbia solo sfogliato l'appendice può usare i termini chiave come un utile riscontro. Tutti i termini sono definiti nel Glossario al termine del libro.

bus
decodificatore
elemento di stato
errore di sincronizzazione
flip-flop
flip-flop di tipo D
funzione di stato futuro
ingresso di selezione o di controllo
latch
logica combinatoria
logica sequenziale
macchina a stati finiti
matrice logica programmabile (programmable logic array, PLA)
memoria a sola lettura (read-only memory, ROM)
memoria statica ad accesso casuale (static random access memory, SRAM)

metastabilità
metodologia di temporizzazione
minterm, mintermine o termine prodotto
modalità a pagine
porta logica
porta NAND
porta NOR
register file
ROM programmabile (PROM)
segnale affermato
segnale non affermato (negato)
sfasamento (o skew) del clock
sistema sincrono
somma di prodotti
tempo di hold
tempo di propagazione
tempo di set-up
temporizzazione sensibile ai fronti
temporizzazione sensibile ai livelli

B.10 Esercizi

B.1 [10] <§B.2> Si dimostri che ci sono 2^n elementi nella tabella di verità di una funzione con n ingressi.

B.2 [10] <§B.2> Una funzione logica usata in diverse applicazioni (ad esempio nei sommatori o per calcolare i valori di parità) è l'*OR esclusivo*. L'uscita di una funzione OR esclusivo a due ingressi è vera se esattamente uno degli ingressi è vero. Si indichi la tabella di verità per la funzione OR esclusivo a due ingressi e la si implementi usando porte AND, OR ed invertitori.

B.3 [15] <§B.2> Si dimostri che la porta NOR è universale indicando come si possano costruire le funzioni AND, OR e NOT usando porte NOR a due ingressi.

B.4 [15] <§B.2> Si dimostri che la porta NAND è universale indicando come si possano costruire le funzioni AND, OR e NOT usando porte NAND a due ingressi.

B.5 [15] <§§B.2, B.3> Si dimostri che anche un multiplexer a due ingressi è universale indicando come si possano costruire le funzioni AND, OR e NOT usando multiplexer.

In maggior dettaglio

I teoremi di DeMorgan

Oltre alle leggi fondamentali discusse nel Paragrafo B.2, vi sono due importanti teoremi, detti teoremi di DeMorgan:

$$\overline{A+B} = \overline{A}\cdot\overline{B} \quad \text{e} \quad \overline{A\cdot B} = \overline{A}+\overline{B}$$

B.6 [10] <§B.2> Si dimostrino i teoremi di DeMorgan mediante una tabella di verità come quella qui riportata.

A	B	$\overline{A}$	$\overline{B}$	$\overline{A+B}$	$\overline{A}\cdot\overline{B}$	$\overline{A\cdot B}$	$\overline{A}+\overline{B}$
0	0						
0	1						
1	0						
1	1						

B.7 [15] <§B.2> Si dimostri che le due equazioni di E nell'esempio del Paragrafo B.2 sono equivalenti facendo uso dei teoremi di DeMorgan e delle proprietà riportate nel Paragafo B.2.

B.8 [15] <§§B.2-B.3> Si ricavi la rappresentazione come prodotto di somme per la funzione E riportata nel Paragrafo B.3 a partire dalla rappresentazione come somma di prodotti. Si dovranno utilizzare i teoremi di DeMorgan.

B.9 [30] <§§B.2-B.3> Si enunci un algoritmo per costruire la rappresentazione come somma di prodotti di un'equazione logica qualsiasi costruita con operazioni AND, OR e NOT. L'algoritmo deve essere ricorsivo e non deve costruire la tabella di verità durante il processo di traduzione.

B.10 [5] <§§4.2, B.2, B.3> Si assuma che X sia composto da 3 bit, x2, x1 e x0. Si scrivano quattro funzioni logiche che siano vere se e solo se:

- X contiene un solo 1;
- X contiene un numero pari di 1;
- X, quando interpretato come numero binario senza segno, è minore di 3;
- X, quando interpretato come numero con segno (in complemento a due), è minore di -1.

B.11 [5] <§§4.2, B.2, B.3> {Es. B.10} Si implementino le quattro funzioni descritte nell'Esercizio B.10 usando una PLA.

B.12 [5] <§§4.2, B.2, B.3> Si assuma che X sia composto di 3 bit, x2, x1 e x0 e che Y sia composto di 3 bit, y2, y1 e y0. Si scrivano delle funzioni logiche che siano vere se e solo se:

- X < Y, dove X ed Y sono interpretati come numeri binari senza segno;
- X < Y, dove X ed Y sono interpretati come numeri con segno (in complemento a due);
- X = Y.

B.13 [5] <§§B.2, B.3> Si calcoli la tabella di verità di un multiplexer (ingressi A, B e S; uscita C) facendo uso delle condizioni di indifferenza per semplificare la tabella ove possibile.

B.14 [5] <§§B.2, B.3> Si implementi una rete di commutazione che abbia due ingressi di dato (A e B), due uscite di dato (C e D) ed un ingresso di controllo (S). Se S è pari ad 1, la rete lavora in modalità di attraversamento diretto, in cui C è uguale ad A e D uguale a B. Se S è pari a 0, la rete lavora in modalità incrociata, in cui C è uguale a B e D uguale ad A.

B.15 [10] <§§B.2, B.5> Si costruisca la tabella di verità di una funzione di parità pari con quattro ingressi (si veda il Paragrafo B.6 per la descrizione delle funzioni di parità).

B.16 [10] <§§B.2, B.5> Si implementi la funzione di parità pari con quattro ingressi mediante porte AND e OR con la convenzione dei "pallini" su ingressi ed uscite.

B.17 [10] <§§B.2, B.3, B.5> Si implementi la funzione di parità pari con quattro ingressi mediante PLA.

B.18 [5] <§B.5> Quale dei due seguenti frammenti di codice rappresenta più fedelmente un latch di tipo D? Quale descrive meglio un flip-flop di tipo D?

```
repeat
      while (clock==basso) do
```

```
            {} /* niente */
        Q = D ;
        while (clock==alto) do
            {} /* niente */
  until (manca_la_corrente)
```

oppure

```
  repeat
        while (clock==alto) do
            Q = D ;
  until (finisce_la_batteria)
```

B.19 [5] <§B.5> Molto spesso ci si aspetta che, dato un diagramma temporale che contiene la descrizione dei cambiamenti sull'ingresso di dato *D* e sull'ingresso di clock *C* (come quelli delle figure B.14 e B.17), vi siano delle differenze nelle forme d'onda dell'uscita *Q* per un latch di tipo D e per un flip-flop di tipo D. Si descrivano con una o due frasi le circostanze (ad esempio la caratteristica degli ingressi) per cui non esiste alcuna differenza tra le due forme d'onda in uscita.

B.20 [5] <§B.5> La figura B.19 illustra l'implementazione del register file per l'unità di elaborazione del MIPS. Si finga di dover costruire un nuovo register file, in cui vi siano solo due registri e solo una porta di lettura, e che ciascun registro abbia solo 2 bit di dato. Si ridisegni la figura B.19 in modo che ciascun collegamento nel diagramma corrisponda ad un solo bit di dato (a differenza del diagramma di figura B.19, dove alcune connessioni hanno ampiezza pari a 5 bit ed altre pari a 32 bit). Di ridisegnino i registri usando flip-flop di tipo D. Non occorre indicare l'implementazione dei flip-flop di tipo D o dei multiplexer.

B.21 [10] <§B.6> Un amico vi chiede di costruire un "occhio elettronico" da integrare in un antifurto fittizio. Il dispositivo consiste di tre luci allineate, controllate dalle uscite Sinistra, Centro e Destra, le quali, se affermate, indicano che la luce corrispondente deve essere accesa. In ogni istante è accesa una sola luce, che si "sposta" da sinistra a destra e viceversa, spaventando così i ladri che credono che il dispositivo stia controllando la loro attività. Si disegni la rappresentazione grafica della macchina a stati finiti usata per specificare l'occhio elettronico. Si noti che la frequenza di movimento dell'occhio è controllata dalla velocità del clock (che non deve essere troppo elevata) e che in pratica non ci sono ingressi.

B.22 [10] <§B.6> {Es. B.21} Si assegnino dei valori numerici agli stati della macchina a stati finiti costruita nell'Esercizio B.21 e si scriva un insieme di equazioni logiche per le diverse uscite, compresi i bit di stato futuro.

B.23 [15] <§§B.2, B.5> Si costruisca un contatore a 3 bit usando tre flip-flop di tipo D ed un gruppo di porte logiche. Gli ingressi sono costituiti da un segnale che riporta il contatore a 0, detto *reset*, e da un segnale che incrementa il contatore, detto *inc*; l'uscita deve indicare il valore raggiunto dal contatore. Quando il contatore raggiunge il valore 7 e viene incrementato deve ricominciare dall'inizio e tornare a 0.

B.24 [20] <§§B.3, B.5> Un *codice Gray* è una sequenza di numeri binari con la proprietà che tra due elementi consecutivi della sequenza può cambiare al più un bit; ad esempio un codice Gray su 3 bit è il seguente: 000, 001, 011, 010, 110, 111, 101 e 100. Usando tre flip-flop di tipo D ed una PLA si costruisca un contatore di 3 bit

secondo il codice Gray che abbia due ingressi: *reset*, che riporta il contatore a 000, e *inc*, che fa avanzare il contatore al prossimo elemento nella sequenza. Si noti che il codice è ciclico, per cui il valore successivo a 100 è nuovamente 000.

B.25 [25] <§§B.2, B.6> Si vuole aggiungere la luce gialla all'esempio del semaforo del Paragrafo B.6. Per far ciò si porta la frequenza di clock a 0,25 Hz (un periodo di clock corrispondente a 4 secondi), pari alla durata della luce gialla; per evitare che le luci rosse e verdi passino troppo velocemente, si aggiunge un *timer* di 30 secondi. Tale timer ha un solo ingresso, detto *ResetTimer*, che fa ripartire il timer, ed una sola uscita, detta *FineTimer*, che indica che è passato l'intervallo di 30 secondi. Occorre altresì ridefinire i segnali di uscita aggiungendo il giallo: si possono definire due uscite per ciascun semaforo: verde e giallo. Se è affermata l'uscita verdeNS, la luce verde è accesa; se è affermata l'uscita gialloNS, la luce gialla è accesa; se sono entrambe non affermate, la luce rossa è accesa. *Non* si affermino contemporaneamente i segnali del verde e del giallo, perché potrebbe confondere i guidatori americani, anche se gli europei capirebbero il significato del segnale. Si disegni la rappresentazione grafica della macchina a stati finiti del nuovo controllore; si scelgano dei nomi degli stati che siano *diversi* da quelli delle uscite.

B.26 [15] <§B.6> {Es. B.25} Si scrivano le tabelle delle funzioni di stato futuro e di uscita del controllore di semaforo descritto nell'Esercizio B.25.

B.27 [15] <§§B.2, B.6> {Es. B.25} Si assegnino dei valori numerici agli stati dell'esempio del semaforo dell'Esercizio B.25 e si usino le tabelle dell'Esercizio B.26 per scrivere le equazioni logiche delle uscite, comprese le uscite di stato futuro.

B.28 [15] <§§B.3, B.6> {Es. B.25} Si implementino le equazioni logiche dell'Esercizio B.27 mediante PLA.

C

APPENDICE

Realizzazione dell'unità di controllo

Un formato particolare come questo è strettamente dipendente dall'architettura dell'hardware e dal set di istruzioni per cui è stato progettato. Il formato deve corrispondere ad un adeguato compromesso tra la dimensione della ROM, la decodifica delle uscite della ROM, la dimensione del circuito e la velocità di esecuzione dell'elaboratore.

Jim McKevit et al.
8086 design report, 1977

C.1 Introduzione

I circuiti di controllo normalmente sono composti di due parti: una parte combinatoria priva di informazioni di stato ed un'unità di controllo sequenziale che gestisce la sequenzializzazione delle operazioni ed ha il compito di svolgere le funzioni di controllo di più alto livello in un progetto multiciclo. Le unità di controllo combinatorie sono utilizzate spesso per gestire una parte del processo di decodifica e di controllo: il controllo della ALU visto nel Capitolo 5 ne è un esempio. Anche una realizzazione a singolo ciclo come quella del Capitolo 5 può usare un controllore combinatorio, dal momento che non richiede stati multipli. Il Paragrafo C.2 analizza l'implementazione di queste due unità combinatorie partendo dalle tabelle di verità del Capitolo 5.

Poiché le unità di controllo sequenziali sono più grandi e spesso più complesse, esiste un numero maggiore di tecniche per realizzarli. L'utilità di queste tecniche dipende dalla complessità del processo di controllo, da caratteristiche quali il numero medio di stati raggiungibili da ciascuno stato e dalla tecnologia di implementazione.

Il modo più immediato per realizzare una funzione di controllo sequenziale è mediante un blocco di logica che riceve in ingresso lo stato corrente ed il campo codice operativo dell'Instruction Register e produce in uscita i segnali di controllo dell'unità di elaborazione ed il valore dello stato futuro. La rappresentazione di partenza può essere sia un diagramma a stati finiti sia un microprogramma; in quest'ultimo caso ciascuna microistruzione rappresenta uno stato. In una implementazione basata su un controllore a stati finiti la funzione di stato futuro verrà calcolata attraverso della logica apposita. Il Paragrafo C.3 descrive la costruzione di tale logica tramite ROM o PLA.

Una modalità di realizzazione alternativa consiste nel calcolare la funzione di stato futuro utilizzando un contatore che incrementa lo stato corrente per determinare lo stato futuro. Quando lo stato futuro non è il successivo in ordine sequenziale, viene attivato un ulteriore circuito per la determinazione dello stato. Il Paragrafo C.4 illustra questo tipo di implementazione e mostra come possa essere utilizzata per realizzare il controllore a stati finiti descritto nel Capitolo 5.

Nel Paragrafo C.5 si mostrerà come passare da una rappresentazione in forma di microprogramma alla logica di controllo.

C.2 Realizzazione di unità di controllo combinatorie

In questo paragrafo si mostrerà come l'unità di controllo per la ALU e l'unità di controllo principale per il progetto a singolo clock siano tradotte in una descrizione circuitale a livello di porte logiche. Con i moderni sistemi CAD tale processo è completamente automatico; gli esempi illustrano come un sistema CAD sfrutti la struttura della funzione di controllo, compresa la presenza di termini indifferenti (*don't care*).

ALUOp		Campo funct						Operazione
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Figura C.1 La tabella di verità per i tre bit di controllo della ALU (denominati Operazione) in funzione di ALUOp e del campo codice di funzione. La tabella coincide con quella mostrata in figura 5.15.

Funzione di controllo della ALU e porte logiche

La figura C.1 mostra la tabella della verità per la funzione di controllo della ALU sviluppata nel Paragrafo 5.3. Un blocco logico che implementa la funzione di controllo della ALU possiederà tre uscite distinte (denominate Operazione2, Operazione1 e Operazione0), ciascuna corrispondente ad uno dei tre bit di controllo della ALU nell'ultima colonna della figura C.1. La funzione logica svolta da ciascuna uscita è costruita sommando tutte le combinazioni della tabella della verità in corrispondenza delle quali quella particolare uscita vale 1. Per esempio il bit di ordine inferiore per il controllo della ALU (Operazione0) assume il valore 1 in corrispondenza delle ultime due righe della tabella di verità della figura C.1. Di conseguenza la tabella di verità per il segnale Operazione0 comprenderà queste due righe.

La figura C.2 mostra le tabelle di verità di ciascuno dei tre bit di controllo della ALU; si è tratto vantaggio dalla struttura comune delle tre tabelle di verità per aumentare il numero di *don't care*; ad esempio le cinque righe della tabella di verità della figura C.1 in corrispondenza delle quali Operazione1 vale 1 sono state ridotte a due soltanto nella figura C.2. Un programma di minimizzazione logica potrà utilizzare i termini *don't care* per ridurre il numero di porte logiche ed il numero di ingressi per ciascuna porta logica partendo da queste tabelle di verità.

Dalla tabella di verità semplificata della figura C.2 si può generare la logica riportata in figura C.3, denominata *blocco di controllo della ALU*. Questo processo è immediato e può essere eseguito utilizzando un programma di supporto al progetto su calcolatore (Computer-aided Design o CAD). Un esempio di come le porte logiche possano essere derivate dalle tabelle di verità è fornito nella didascalia della figura C.3.

La logica di controllo della ALU è semplice poiché vi sono tre sole uscite, e solo alcune delle possibili configurazioni di ingresso devono venire riconosciute. Se fosse necessario trasformare un gran numero di codici funzionali della ALU in segnali di controllo per la ALU stessa, il semplice metodo visto sinora non sarebbe efficiente. Si potrebbe usare invece un decodificatore, una memoria o una schiera di porte logiche. Queste tecniche sono descritte nell'Appendice B e di esse si vedranno alcuni esempi quando si tratterà dell'implementazione del controllore multiciclo nel Paragrafo C.3.

Approfondimento: In generale la rappresentazione di una funzione logica tramite equazione logica o tramite tabella di verità sono equivalenti (tale punto viene discusso in maggiore det-

ALUOp		Campo codice funzione					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
X	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

a. Tabella di verità per Operazione2 = 1 (questa tabella corrisponde al bit a sinistra nel campo Operazione nella figura C.1)

ALUOp		Campo codice funzione					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

b. Tabella di verità per Operazione1 = 1

ALUOp		Campo codice funzione					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X

c. Tabella di verità per Operazione0 = 1

Figura C.2 Le tabelle di verità per le tre linee di controllo della ALU. Sono riportate soltanto le righe in corrispondenza delle quali l'uscita assume il valore 1. I bit in ciascun campo sono numerati da destra a sinistra iniziando da 0; di conseguenza F5 è il bit più significativo del campo funzione e F0 è il bit meno significativo. Analogamente i nomi dei segnali corrispondenti al codice su tre bit dell'operazione in ingresso della ALU sono denominati Operazione2, Operazione1 e Operazione0 (l'ultimo corrisponde al bit meno significativo). La tabella di verità mostrata sopra riporta quindi le combinazioni di ingresso per le quali il segnale di controllo della ALU dovrebbe essere 010, 001, 110 o 111 (le combinazioni 011, 100 e 101 non sono utilizzate). I bit ALUOp sono denominati ALUOp1 e ALUOp0. I tre valori di uscita dipendono dal campo di 2 bit ALUOp e, quando tale campo assume il valore 10, dal codice funzione di 6 bit che compare nell'istruzione. Allo stesso modo quando il campo ALUOp non è uguale a 10 non è significativo il valore del codice funzione (che è quindi rappresentato con una X). Si veda l'Appendice B per maggiori approfondimenti sui valori *don't care*.

taglio nell'Appendice B). Tuttavia è possibile che quando una tabella di verità specifica soltanto le righe corrispondenti ad uscite non nulle questa non descriva completamente la funzione logica. Una tabella di verità completa indica tutti gli elementi compresi i valori *don't care*: per esempio la codifica 11 per ALUOp corrisponde sempre ad un valore *don't care* sull'uscita. Di conseguenza una tabella di verità completa includerebbe il valore XXX nella parte relativa alle uscite per tutte le configurazioni con il valore 11 nel campo ALUOp. Tali configurazioni con il valore *don't care* permettono di sostituire i valori 10 e 01 per il campo ALUOp rispettivamente con 1X e X1. Sia l'identificazione dei termini *don't care* sia la minimizzazione della logica corrispondente sono operazioni complesse e soggette ad errori, ed è quindi preferibile che vengano eseguite da un programma.

Funzione di controllo complessiva e porte logiche

La realizzazione della funzione di controllo complessiva attraverso un insieme non strutturato di porte logiche, secondo l'approccio seguito per la funzione di controllo della ALU, in questo caso è possibile poiché la funzione di controllo non è né complessa né di grandi dimensioni, come si può vedere dalla tabella di verità della figura C.4. Se tuttavia fosse stato utilizzato un numero maggiore dei 64 codici operativi previsti e ci fossero state molte più linee di controllo, il numero delle porte logiche sarebbe stato molto maggiore e ciascuna porta logica avrebbe avuto molti più ingressi.

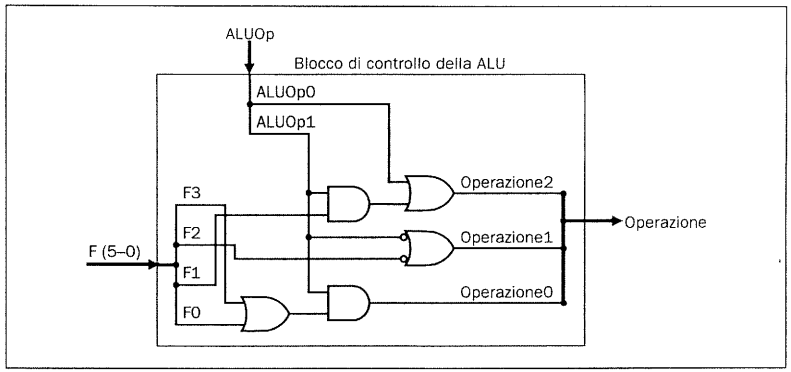


Figura C.3 Il blocco di controllo della ALU genera i tre bit di controllo della ALU sulla base del codice funzione e dei bit ALUOp. La logica riportata è stata generata direttamente a partire dalla tabella di verità della figura C.2. Solo 4 dei 6 bit nel codice funzione sono effettivamente necessari come ingressi in quanto i due bit più significativi hanno sempre il valore *don't care*. Si esamini la relazione tra la logica riportata e la tabella di verità della figura C.2. Si consideri l'uscita Operazione2 che viene prodotta dalle due righe della tabella di verità di Operazione2: la seconda riga corrisponde ad un'operazione AND tra due termini (F1 = 1 e ALUOp1 = 1); la porta AND a due ingressi in alto corrisponde a questo termine. L'altro termine che fa sì che Operazione2 assuma il valore vero corrisponde semplicemente a ALUOp0; questi due termini sono combinati tramite una porta OR la cui uscita è Operazione2. Le uscite Operazione0 e Operazione1 sono dedotte dalla tabella di verità in modo analogo.

Controllo	Nome del segnale	formato-R	lw	sw	beq
Ingressi	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Uscite	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figura C.4 La funzione di controllo dell'implementazione elementare a singolo ciclo è completamente specificata attraverso la tabella di verità. La tabella coincide con quella mostrata nella figura 5.27.

Poiché qualsiasi funzione può essere calcolata attraverso due livelli di logica, si può seguire un'altra strada per la sua realizzazione, basata su una struttura logica regolare a due livelli. La figura C.5 mostra un'implementazione di questo tipo, che usa una schiera di porte AND seguita da una schiera di porte OR. Tale struttura è chiamata *matrice logica programmabile* (*PLA, programmable logic array*). La PLA rap-

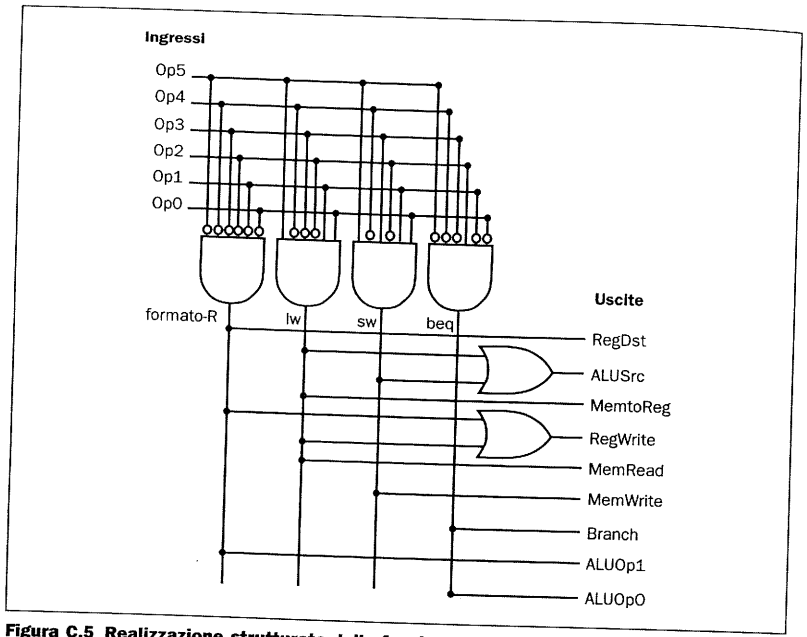


Figura C.5 Realizzazione strutturata della funzione di controllo descritta dalla tabella di verità della figura C.4. La struttura, denominata matrice logica programmabile (PLA o programmable logic array) usa una schiera di porte AND seguita da una schiera di porte OR. Gli ingressi delle porte AND sono gli ingressi della funzione ed i loro negati (i pallini indicano l'inversione di un segnale). Gli ingressi delle porte OR sono le uscite delle porte AND (o, nel caso limite, gli ingressi della funzione ed i loro negati). Le uscite delle porte OR corrispondono alle uscite della funzione.

presenta uno dei modi più comuni per realizzare una funzione di controllo. Si ritornerà sull'uso di elementi logici strutturati per la realizzazione della funzione di controllo quando, nel prossimo paragrafo, si implementerà il controllore a stati finiti.

C.3 Realizzazione dell'unità di controllo sotto forma di macchina a stati finiti

Per realizzare l'unità di controllo sotto forma di macchina a stati finiti si deve innanzitutto assegnare un numero a ciascuno dei 10 stati; a ciascuno stato può essere assegnato un numero qualsiasi, anche se qui per semplicità si userà una numerazione sequenziale, come si è fatto nel Capitolo 5 (la figura C.6 è una copia del diagramma a stati finiti tratto dalla figura 5.42, qui riprodotto per comodità del lettore). Con 10 stati sono necessari 4 bit per codificare il numero dello stato: si denomineranno S3, S2, S1 e S0 i bit di stato; il numero dello stato corrente verrà memorizzato nel registro di stato, come mostrato nella figura C.7. Se gli stati sono codificati in modo sequenziale, lo stato *i* è codificato usando i bit di stato nella configurazione corrispondente al numero binario *i*: per esempio lo stato 6 è codificato come 0110_{due}, ossia S3 = 0, S2 = 1, S1 = 1, S0 = 0, e può anche essere scritto come

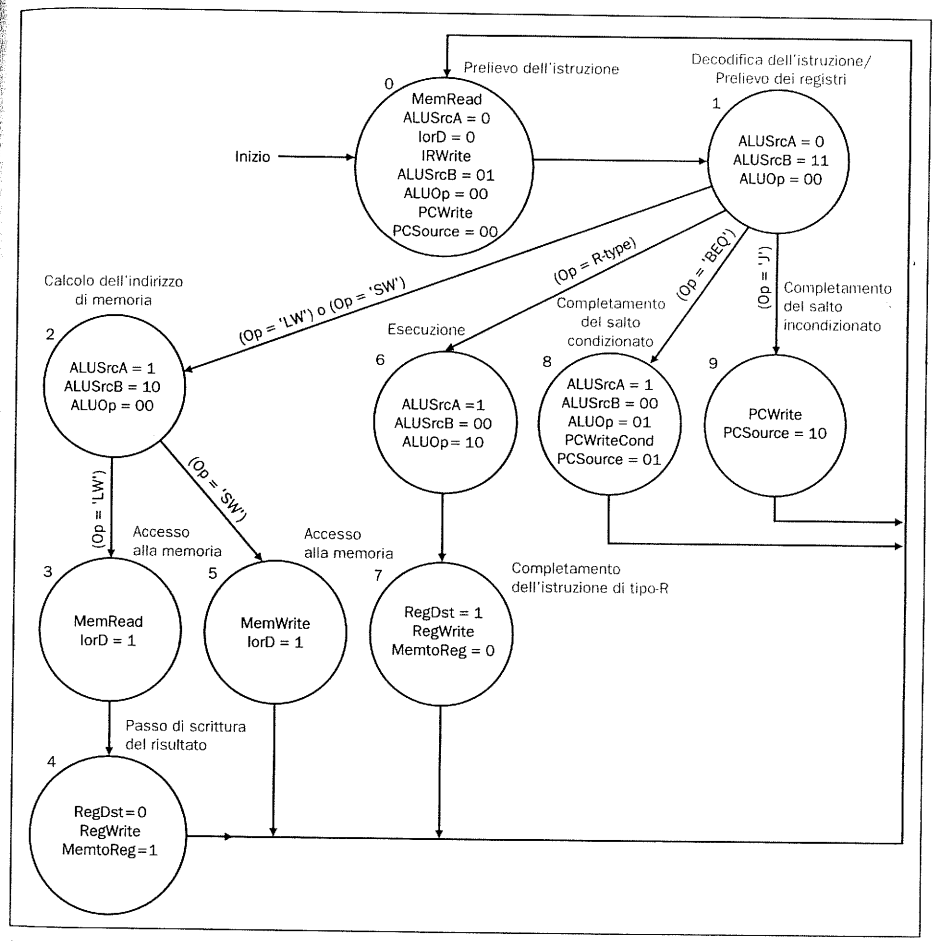


Figura C.6 Il diagramma a stati finiti sviluppato nel Capitolo 5; è identico a quello riportato in figura 5.42.

$$\overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0}$$

L'unità di controllo possiede delle uscite che specificano lo stato futuro: esse sono scritte nel registro di stato in corrispondenza del fronte del clock e diventano il nuovo stato all'inizio del ciclo di clock successivo al fronte di clock attivo. Tali uscite sono denominate SF3, SF2, SF1, SF0. Una volta che si è determinato il numero degli ingressi, degli stati e delle uscite si sa quale sarà la struttura complessiva dell'unità di controllo, secondo lo schema riportato in figura C.7.

Il blocco di figura C.7 etichettato "logica di controllo" è di tipo combinatorio: lo si può immaginare come una grande tabella che fornisce il valore delle uscite in

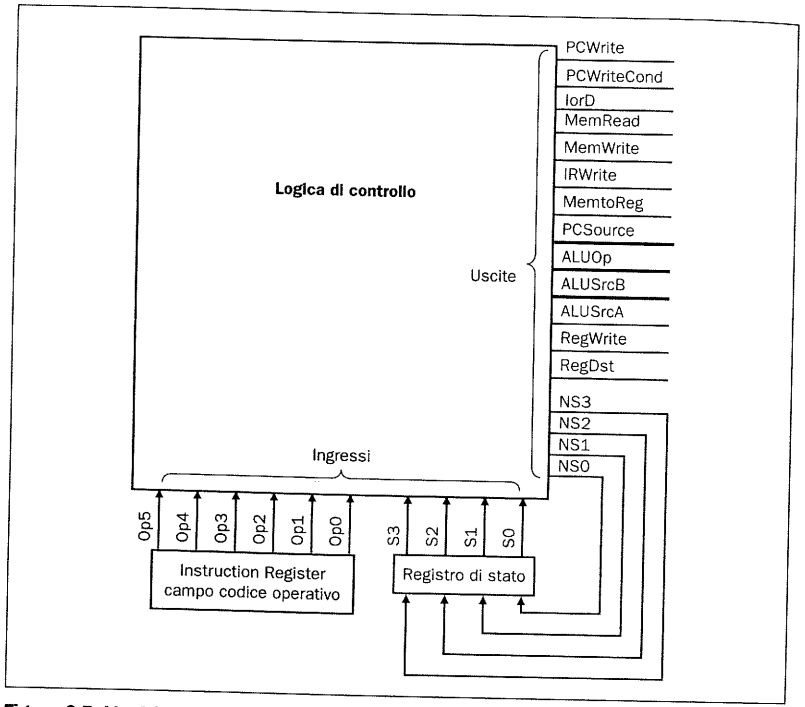


Figura C.7 L'unità di controllo del MIPS consiste di un blocco di logica di controllo e di un registro per la memorizzazione dello stato. Il registro di stato viene scritto in corrispondenza del fronte attivo del clock e rimane stabile durante il ciclo di clock.

funzione degli ingressi. La logica contenuta in questo blocco implementa le due diverse parti della macchina a stati finiti; una parte è la logica che determina i valori delle uscite di controllo dell'unità di elaborazione che dipendono solo dai bit di stato, l'altra parte del controllo implementa la funzione di stato futuro; queste equazioni determinano i valori dei bit di stato futuro in funzione dei bit dello stato corrente e degli altri ingressi (il codice operativo su 6 bit).

La figura C.8 mostra le equazioni logiche: nella parte superiore compaiono le uscite, in quella inferiore la funzione di stato futuro; i valori nella tabella sono stati dedotti dal diagramma a stati della figura C.6. Tutte le volte che in uno stato viene attivata una linea di controllo, lo stato stesso compare nella seconda colonna della tabella. Analogamente gli elementi relativi allo stato futuro compaiono ogni volta che uno stato è il successore di un altro.

Nella Figura C.8 si usa l'abbreviazione stato*N* per indicare lo stato corrente *N*; stato*N* è quindi sostituito con il termine che codifica lo stato numero *N*. Si usa StatoFuturo*N* per indicare il valore delle uscite di stato futuro per *N*; tali uscite sono realizzate usando le uscite di stato futuro (SF). Quando StatoFuturo*N* è attivo i bit SF[3-0] assumono il valore corrispondente alla versione binaria del valore *N*. Ovviamente, dal momento che un certo bit di stato futuro viene attivato in un certo numero di stati futuri, l'equazione per ciascun bit di stato corrisponderà all'OR dei

Uscita	Stato corrente	Operazione
PCWrite	stato0 + stato9	
PCWriteCond	stato8	
IorD	stato3 + stato5	
MemRead	stato0 + stato3	
MemWrite	stato5	
IRWrite	stato0	
MemtoReg	stato4	
PCSource1	stato9	
PCSource0	stato8	
ALUOp1	stato6	
ALUOp0	stato8	
ALUSrcB1	stato1 + stato2	
ALUSrcB0	stato0 + stato1	
ALUSrcA	stato2 + stato6 + stato8	
RegWrite	stato4 + stato7	
RegDst	stato7	
StatoFuturo0	stato4 + stato5 + stato7 + stato8+ stato9	
StatoFuturo1	stato0	
StatoFuturo2	stato1	(Op='lw')+(Op='sw')
StatoFuturo3	stato2	(Op='lw')
StatoFuturo4	stato3	
StatoFuturo5	stato2	(Op='sw')
StatoFuturo6	stato1	(Op='tipo-R')
StatoFuturo7	stato6	
StatoFuturo8	stato1	(Op='beq')
StatoFuturo9	stato1	(Op='jmp')

Figura C.8 Le equazioni logiche per l'unità di controllo in forma sintetica. Si ricordi che nelle equazioni logiche il segno "+" sta per OR. Gli ingressi di stato e le informazioni relative alle uscite di stato futuro devono essere espanse usando la codifica degli stati. Tutte le caselle vuote corrispondono a valori *don't care*.

termini che attivano tale segnale. Analogamente, quando si usa un termine come (Op = 'lw'), questo corrisponde all'AND degli ingressi di codice operativo che specificano la codifica del codice operativo `lw` su 6 bit, come si è fatto per la semplice unità di controllo vista nel precedente paragrafo di questo capitolo. La traduzione delle informazioni della figura C.8 in equazioni logiche per le uscite è immediata.

Esempio

Equazioni logiche per le uscite di stato futuro

Si calcoli l'equazione logica per il bit di stato futuro di ordine inferiore, SF0.

Soluzione

Il bit di stato futuro SF0 deve essere attivo ogni volta che lo stato futuro è caratterizzato dalla condizione SF0 = 1 nella codifica dello stato. Questa condizione è vera per StatoFuturo1, StatoFuturo3, StatoFuturo5, StatoFuturo7 e StatoFuturo9. Le righe relative a questi stati nella figura C.8 forniscono le condizioni per le quali questi valori di stato futuro devono essere attivi; le equazioni relative a

ciascuno degli stati futuri sono riportate sotto. La prima equazione afferma che lo stato futuro è 1 se lo stato corrente è 0; lo stato corrente è 0 se ciascuno dei bit di ingresso di stato vale 0, come indica il termine prodotto a destra.

$$\text{StatoFuturo1} = \text{Stato0} = \overline{S3}\cdot\overline{S2}\cdot\overline{S1}\cdot\overline{S0}$$

$$\text{StatoFuturo3} = \text{Stato2}\cdot(\text{Op[5-0]} = \text{'lw'}) = \overline{S3}\cdot\overline{S2}\cdot S1\cdot\overline{S0}\cdot Op5\cdot\overline{Op4}\cdot\overline{Op3}\cdot\overline{Op2}\cdot Op1\cdot Op0$$

$$\text{StatoFuturo5} = \text{Stato2}\cdot(\text{Op[5-0]} = \text{'sw'}) = \overline{S3}\cdot\overline{S2}\cdot S1\cdot\overline{S0}\cdot Op5\cdot\overline{Op4}\cdot Op3\cdot\overline{Op2}\cdot Op1\cdot Op0$$

$$\text{StatoFuturo7} = \text{Stato6} = \overline{S3}\cdot S2\cdot S1\cdot\overline{S0}$$

$$\text{StatoFuturo9} = \text{Stato1}\cdot(\text{Op[5-0]} = \text{'jmp'}) = \overline{S3}\cdot\overline{S2}\cdot\overline{S1}\cdot S0\cdot\overline{Op5}\cdot\overline{Op4}\cdot\overline{Op3}\cdot\overline{Op2}\cdot Op1\cdot\overline{Op0}$$

SF0 è la somma logica di tutti questi termini.

Come si è visto, la funzione di controllo può essere espressa sotto forma di un'equazione logica per ciascuna uscita. Questo insieme di equazioni logiche può essere implementato in due modi: utilizzando una tabella di verità completa o utilizzando una struttura logica a due livelli che permette una codifica sparsa della tabella di verità. Prima di vedere queste implementazioni, è bene osservare la tabella di verità per la funzione di controllo completa.

Per semplicità conviene spezzare la funzione di controllo definita nella figura C.8 in due parti: le uscite di stato futuro, che possono dipendere da tutti gli ingressi, e le uscite corrispondenti ai segnali di controllo, che dipendono solo dai bit di stato corrente. La figura C.9 mostra le tabelle di verità per tutti i segnali di controllo richiesti dall'unità di elaborazione. Dal momento che questi segnali dipendono in realtà solo dai bit di stato (e non dal codice operativo), ciascuna delle righe nelle tabelle di figura C.9 rappresenta 64 ($=2^6$) combinazioni, in corrispondenza delle quali i 6 bit denominati Op assumono tutti i possibili valori; questo significa che i bit Op costituiscono dei bit *don't care* nella determinazione delle uscite di controllo dell'unità di elaborazione. La figura C.10 mostra la tabella di verità per i bit di stato futuro SF[3-0], che dipendono sia dai bit di ingresso di stato che dai bit dell'istruzione che specificano il codice operativo.

Approfondimento: Vi sono molte possibilità per semplificare la funzione di controllo se si osservano le similitudini tra due o più segnali di controllo e si usa la semantica dell'implementazione. Per esempio i segnali PCWriteCond, PCSource0 e ALUOp0 sono tutti affermati in corrispondenza di uno stato, lo stato 8. Questi tre segnali di controllo possono quindi essere sostituiti da un unico segnale.

Implementazione tramite ROM

Probabilmente il modo più semplice per implementare la funzione di controllo è codificare le tabelle di verità in una memoria a sola lettura (ROM). Il numero delle righe nella memoria che implementa le tabelle di verità delle figure C.9 e C.10 è

s3	s2	s1	s0
0	0	0	0
1	0	0	1

a. Tabella di verità per PCWrite

s3	s2	s1	s0
1	0	0	0

b. Tabella di verità per PCWriteCond

s3	s2	s1	s0
0	0	1	1
0	1	0	1

c. Tabella di verità per IorD

s3	s2	s1	s0
0	0	0	0
0	0	1	1

d. Tabella di verità per MemRead

s3	s2	s1	s0
0	1	0	1

e. Tabella di verità per MemWrite

s3	s2	s1	s0
0	0	0	0

f. Tabella di verità per IRWrite

s3	s2	s1	s0
0	1	0	0

g. Tabella di verità per MemtoReg

s3	s2	s1	s0
1	0	0	1

h. Tabella di verità per PCSource1

s3	s2	s1	s0
1	0	0	0

i. Tabella di verità per PCSource0

s3	s2	s1	s0
0	1	1	0

j. Tabella di verità per ALUOp1

s3	s2	s1	s0
1	0	0	0

k. Tabella di verità per ALUOp0

s3	s2	s1	s0
0	0	0	1
0	0	1	0

l. Tabella di verità per ALUSrcB1

s3	s2	s1	s0
0	0	0	0
0	0	0	1

m. Tabella di verità per ALUSrcB0

s3	s2	s1	s0
0	0	1	0
0	1	1	0
1	0	0	0

n. Tabella di verità per ALUSrcA

s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Tabella di verità per RegWrite

s3	s2	s1	s0
0	0	0	0

p. Tabella di verità per RegDst

Figura C.9 Sono riportate le tabelle di verità per i 16 segnali di controllo dell'unità di elaborazione che dipendono esclusivamente dai bit di ingresso di stato. Ciascuna riga di ciascuna tabella corrisponde a 64 combinazioni, una per ciascuno dei possibile valori dei 6 bit Op. Si noti che alcune delle uscite sono attive in condizioni pressoché identiche; per esempio PCWriteCond, PCSource0 e ALUOp0 sono attivi soltanto nello stato 8 (si vedano le tabelle (b), (j) e (l)): i tre segnali possono quindi essere sostituiti da un unico segnale. Vi sono altre possibilità per ridurre la logica necessaria per implementare la funzione di controllo sfruttando ulteriori somiglianze esistenti nelle tabelle di verità.

pari a tutti i possibili valori degli ingressi (i 6 bit di codice operativo più i 4 bit di stato), ossia $2^{\#\text{ingressi}} = 2^{10} = 1024$. Gli ingressi dell'unità di controllo diventano le linee di indirizzo della ROM, che implementa il blocco della logica di controllo mostrato nella figura C.7. L'ampiezza di ciascuna riga (o parola di memoria) è pari a 20 bit, dal momento che vi sono 16 uscite di controllo per l'unità di elaborazione e 4 bit di stato futuro. Questo significa che la dimensione totale della ROM è pari a $2^{10} \times 20 = 20$ Kbit.

I valori dei bit di ciascuna parola della ROM dipendono da quali uscite sono attive in quella parola. Per prima cosa si devono ordinare i bit all'interno degli

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1

a. Tabella di verità per l'uscita SF3, che è attiva quando lo stato futuro è 8 o 9. Il segnale viene attivato quando lo stato corrente è 1.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	0	1	1
X	X	X	X	X	X	0	1	1	0

b. Tabella di verità per l'uscita SF2, che è attiva quando lo stato futuro è 4, 5, 6 o 7. Questo succede solo se lo stato corrente è 1, 2, 3 o 6.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	0	0
1	0	1	0	1	1	0	0	0	1
1	0	0	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0

c. Tabella di verità per l'uscita SF1, che è attiva quando lo stato futuro è 2, 3, 6 o 7. Lo stato futuro è 2, 3, 6 o 7 solo se lo stato corrente è 1, 2 o 6.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
X	X	X	X	X	X	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0
0	0	0	0	1	0	0	0	0	1

d. Tabella di verità per l'uscita SF0, che è attiva quando lo stato futuro è 1, 3, 5, 7 o 9. Questo succede solo se lo stato corrente è 0, 1, 2 o 6.

Figura C.10 Tabelle di verità per i 4 bit di uscita di stato futuro SF[3-0]. Le uscite di stato futuro dipendono dai valori di Op[5-0], corrispondenti al codice operativo, e dallo stato corrente, dato da S[3-0]. Gli elementi con valore X sono i termini *don't care*; ciascuna riga con un termine *don't care* corrisponde a due combinazioni, una con l'ingresso a 0 ed una con l'ingresso a 1. Una riga con *n don't care* corrisponde quindi in realtà a 2^n combinazioni nella tabella di verità.

ingressi di controllo (corrispondenti all'indirizzo) e all'interno delle parole di uscita (corrispondenti al contenuto). I bit verranno numerati utilizzando l'ordine che appare nella figura C.7, in cui i bit di stato futuro sono i bit di ordine inferiore della *parola* di controllo e i bit di ingresso dello stato corrente sono i bit di ordine inferiore dell'*indirizzo*. Questo significa che l'uscita PCWrite sarà il bit di ordine superiore (bit 19) di ciascuna parola di memoria, e SF0 sarà il bit di ordine inferiore. Il bit di indirizzo di ordine più elevato sarà dato da Op5, che è il bit di ordine superiore all'interno dell'istruzione, mentre il bit di indirizzo di ordine inferiore sarà dato da S0.

Si può definire il contenuto della ROM costruendo l'intera tabella di verità in una forma in cui ciascuna riga corrisponde ad una delle 2^n combinazioni di ingresso, ed un insieme di colonne indica quali uscite sono attive in corrispondenza di tale combinazione di ingresso. Non vi è qui lo spazio per mostrare tutte le 1024 righe della tabella di verità; tuttavia, se si separano le uscite di controllo dell'unità di elaborazione da quelle di stato futuro si ottiene una tabella di dimensioni accettabili, poiché le uscite di controllo dell'unità di elaborazione dipendono soltanto dallo stato corrente. La tabella di verità delle uscite di controllo per l'unità di elaborazione è riportata nella figura C.11; sono state inserite solo le codifiche degli ingressi di stato utilizzati (vale a dire i valori da 0 a 9, corrispondenti ai 10 stati della macchina a stati finiti).

La tabella di verità della figura C.11 fornisce direttamente il contenuto dei 16 bit più significativi di ciascuna parola della ROM. Il campo di 4 bit di ingresso fornisce i 4 bit di indirizzo di ordine inferiore, e la colonna fornisce il contenuto della parola posta a tale indirizzo.

Se si fosse mostrata la tabella di verità completa per i bit di controllo dell'unità di elaborazione considerando come ingressi sia il numero dello stato sia i bit del codice operativo, gli ingressi del codice operativo sarebbero stati tutti *don't care*. Quando si costruisce la ROM, non è possibile avere dei *don't care*, dal momento che gli indirizzi della ROM devono essere completamente specificati. Di conseguenza nella ROM compariranno più volte le stesse combinazioni di valori per le uscite di controllo dell'unità di elaborazione, poiché questa parte della ROM è la stessa se i bit di stato sono identici, indipendentemente dal valore degli ingressi di codice operativo.

Uscite	Valore degli ingressi (S[3-0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	1	0
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

Figura C.11 Tabella di verità per le 16 uscite di controllo dell'unità di elaborazione, che dipendono solo dagli ingressi di stato. I valori sono determinati sulla base della figura C.9. Benché vi siano 16 possibili valori del campo di stato di 4 bit, solo 10 di questi sono utilizzati e mostrati qui. I 10 possibili valori sono mostrati in alto; ciascuna colonna mostra il valore delle uscite di controllo dell'unità di elaborazione per il valore di ingresso di stato che compare in cima alla colonna. Per esempio quando gli ingressi di stato valgono 0011 (stato 3) le uscite di controllo attive sono IorD o MemRead.

Esempio

Contenuto della ROM che realizza l'unità di controllo

In corrispondenza di quali indirizzi della ROM il bit corrispondente a PCWrite, ossia il bit di ordine superiore nella parola di controllo, vale 1?

Soluzione

PCWrite è affermato negli stati 0 e 9; questi corrispondono agli indirizzi che hanno i 4 bit meno significativi pari a 0000 oppure 1001. Il bit sarà 1 all'interno delle parole di memoria indipendentemente dal valore degli ingressi Op[5-0], così che gli indirizzi in corrispondenza dei quali il bit è 1 sono 0000000000, 0000001001, 0000010000, 0000011001, ..., 1111110000, 1111111001. Questo può essere generalizzato nella forma XXXXXX0000 o XXXXXX1001, dove XXXXXX rappresenta una qualsiasi combinazione di bit e corrisponde al codice operativo su 6 bit, dal quale non dipende questa uscita.

L'intero contenuto della ROM verrà mostrato diviso in due parti in maniera da rendere più semplice la sua rappresentazione; la figura C.12 mostra i 16 bit più significativi della parola di controllo, ripresi direttamente dalla figura C.11. Queste uscite di controllo dell'unità di elaborazione dipendono soltanto dagli ingressi di stato, e questo insieme di parole è duplicato per 64 volte nella ROM completa, come descritto in precedenza. Le righe corrispondenti ai valori di ingresso da 1010 a 1111 non sono utilizzate, e quindi il loro contenuto non viene considerato.

La figura C.13 mostra i 4 bit meno significativi della parola di controllo, corrispondenti alle uscite di stato futuro. L'ultima colonna della tabella della figura C.13 corrisponde a tutti i possibili valori del codice operativo che non corrispondono ai codici operativi specificati. Nel caso dello stato 0 lo stato futuro è sempre lo stato 1, poiché l'istruzione è ancora in corso di caricamento; dopo lo stato 1 il campo codice operativo deve essere valido. Nella tabella questo viene rappresentato attraverso le righe che riportano l'indicazione "illegale"; si parlerà di come gestire questi codici operativi illegali nel Paragrafo 5.6.

La rappresentazione basata su due tabelle separate non è soltanto un modo più compatto per mostrare il contenuto della ROM, ma è anche un modo più efficace

4 bit meno significativi dell'indirizzo	Bit 19-4 della parola
0000	1001010000001000
0001	0000000000011000
0010	0000000000010100
0011	0011000000000000
0100	0000001000000010
0101	0010100000000000
0110	0000000001000100
0111	0000000000000011
1000	0100000010100100
1001	1000000100000000

Figura C.12 Il contenuto dei 16 bit più significativi della ROM dipende solo dagli ingressi di stato. Questi valori sono gli stessi della Figura C.11, semplicemente ruotati di 90°. Questo insieme di parole di controllo è duplicato 64 volte per ogni possibile valore dei 6 bit più significativi dell'indirizzo.

	Op [5-0]					
Stato corrente	000000	000010	000100	100011	101011	Ogni altro valore
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	illegale
0010	XXXX	XXXX	XXXX	0011	0101	illegale
0011	0100	0100	0100	0100	0100	illegale
0100	0000	0000	0000	0000	0000	illegale
0101	0000	0000	0000	0000	0000	illegale
0110	0111	0111	0111	0111	0111	illegale
0111	0000	0000	0000	0000	0000	illegale
1000	0000	0000	0000	0000	0000	illegale
1001	0000	0000	0000	0000	0000	illegale

Figura C.13 Questa tabella contiene i 4 bit meno significativi della parola di controllo (corrispondenti alle uscite SF), che dipendono sia dagli ingressi di stato (S[3-0]) sia dal codice operativo (Op[5-0]), corrispondente al codice operativo dell'istruzione. Questi valori possono essere dedotti dalla figura C.10. Il nome del codice operativo è riportato nell'intestazione al di sotto della codifica. Per ciascun elemento sono riportati i 4 bit della parola di controllo il cui indirizzo è dato dai bit di stato corrente e dai bit Op. Per esempio quando i bit di ingresso di stato sono 0000 l'uscita è sempre 0001, indipendentemente dagli altri ingressi; quando lo stato è 2, lo stato futuro è *don't care* in corrispondenza di 3 delle possibili configurazioni di ingresso, 3 in corrispondenza dell'ingresso pari a lw e 5 in corrispondenza dell'ingresso pari a sw. Insieme alle informazioni della figura C.12 questa tabella fornisce il contenuto della ROM dell'unità di controllo. Per esempio la parola all'indirizzo 1000110001 è ottenuta determinando i 16 bit più significativi nella tabella della figura C.12 utilizzando i soli bit di ingresso di stato (0001) e concatenando i 4 bit inferiori trovati utilizzando l'intero indirizzo (0001 per identificare la riga e 100011 per identificare la colonna). L'elemento deducibile dalla figura C.12 è **00000000000110000**, mentre l'elemento corretto nella tabella riportata in alto è 0010. Di conseguenza la parola di controllo all'indirizzo 10000110001 è **00000000000110000**0010. La colonna etichettata "Ogni altro valore" va considerata solo quando i bit Op non corrispondono a nessuno dei codici operativi indicati.

per implementarla. La maggior parte delle uscite (16 dei 20 bit) dipendono soltanto da 4 dei 10 ingressi. Il numero totale di bit che si ottiene allorché il controllo è implementato attraverso due ROM separate è pari a $2^4 \times 16 + 2^{10} \times 4 = 256 + 4096 =$ 4,3 Kbit, che è pari a circa un quinto della dimensione di una ROM singola, che richiede $2^{10} \times 20 = 20$ Kbit. Vi è ovviamente un costo aggiuntivo associato ad ogni blocco di logica strutturata, ma in questo caso il costo aggiuntivo dovuto ad una ROM in più è trascurabile rispetto al vantaggio ottenuto suddividendo la ROM.

La codifica della funzione di controllo tramite ROM, anche se semplice, non è ottimizzata, anche quando viene suddivisa in due parti. I valori degli ingressi dell'Instruction Register, ad esempio, spesso non sono necessari per determinare lo stato futuro; la ROM dello stato futuro contiene di conseguenza molti elementi che sono duplicati o *don't care*. Si consideri il caso in cui il sistema si trovi nello stato 0: nella ROM sono previste 2^6 parole (dal momento che il codice operativo può assumere un qualsiasi valore), e tutte queste parole hanno lo stesso contenuto (vale a dire, la parola di controllo 0001). La ragione per cui una parte così significativa della ROM è sprecata sta nel fatto che la ROM implementa la tabella di verità completa e fornisce la possibilità di avere un'uscita diversa per ciascuna combinazione degli ingressi; nella realtà la maggior parte delle combinazioni di ingresso non si verificano oppure sono ridondanti.

Implementazione tramite PLA

È possibile ridurre la quantità di memoria di controllo necessaria a patto di inserire un circuito di decodifica degli indirizzi di controllo più complesso, in grado di codificare soltanto le combinazioni di ingresso richieste. La struttura logica utilizzata più di frequente a questo scopo è quella rappresentata dalle matrici logiche programmabili (PLA), già trattate in precedenza ed illustrate nella figura C.5. In una PLA ciascuna uscita corrisponde all'OR logico di uno o più mintermini. Un *mintermine*, anche chiamato *termine prodotto*, è semplicemente un AND logico di uno o più ingressi. Gli ingressi possono essere pensati come indirizzi per l'accesso alla PLA, mentre i mintermini selezionano quelli utili tra i possibili indirizzi. Un mintermine corrisponde ad una singola riga in una tabella di verità come quelle nella figura C.9, compresi gli eventuali termini *don't care*; ciascuna uscita corrisponde ad un OR di questi mintermini, il che corrisponde esattamente ad una tabella di verità completa. Tuttavia, diversamente dalla ROM, in questo caso sono necessarie solo le righe della tabella di verità che producono un'uscita affermata, e si richiede soltanto una copia di ciascun mintermine, anche se il mintermine contiene dei *don't care*. La figura C.14 mostra la PLA che realizza la funzione di controllo.

Come si può vedere nella PLA della figura C.14, vi sono 17 mintermini unici, 10 dei quali dipendono solo dallo stato corrente ed altri 7 che dipendono da una combinazione del campo Op e dei bit di stato corrente. La dimensione totale della PLA è proporzionale al numero (ingressi × termini prodotto) + (uscite × termini prodotto), come si può vedere dalla figura. Questo significa che la dimensione totale della PLA mostrata nella figura C.14 è proporzionale a (10 × 17) + (20 × 17) = 460. A titolo di confronto, la dimensione della ROM singola è proporzionale a 20 Kbit, e anche la ROM composta da due parti ha un totale di 4,3 Kbit. Dal momento che la dimensione di una cella della PLA è solo leggermente maggiore di quella di un bit della ROM, una PLA rappresenta un'implementazione molto più efficiente dell'unità di controllo considerata.

Ovviamente, come si è suddivisa la ROM in due parti, così è possibile fare lo stesso per la PLA: una con quattro ingressi e 10 mintermini che genera le 16 uscite di controllo, ed un'altra con 10 ingressi e 7 mintermini che genera le 4 uscite di stato futuro. La prima PLA avrà una dimensione proporzionale a (4 × 10) + (10 × 16) = 200, mentre la seconda avrà una dimensione proporzionale a (10 × 7) + (4 × 7) = 98. Questo significa una dimensione complessiva pari a 298 celle di PLA, corrispondente a circa il 55% della dimensione della PLA singola. Queste due PLA sono considerevolmente più piccole rispetto ad una implementazione basata su due ROM. Per maggiori dettagli sulle PLA e la loro implementazione, così come per i riferimenti a libri sul progetto di circuiti logici si veda l'Appendice B.

C.4 Realizzazione della funzione di stato futuro tramite un sequenzializzatore

Si analizzi con attenzione l'unità di controllo costruita nel paragrafo precedente. Se si esaminano le ROM delle figure C.12 e C.13 che implementano la funzione di controllo, si può osservare che una parte consistente del circuito è utilizzata per specificare la funzione di stato futuro. Infatti nell'implementazione basata su due ROM separate, 4.096 dei 4.368 bit (pari al 94%) corrispondono alla funzione di stato futuro. Inoltre si immagini come sarebbe la logica di controllo se il set di istruzioni

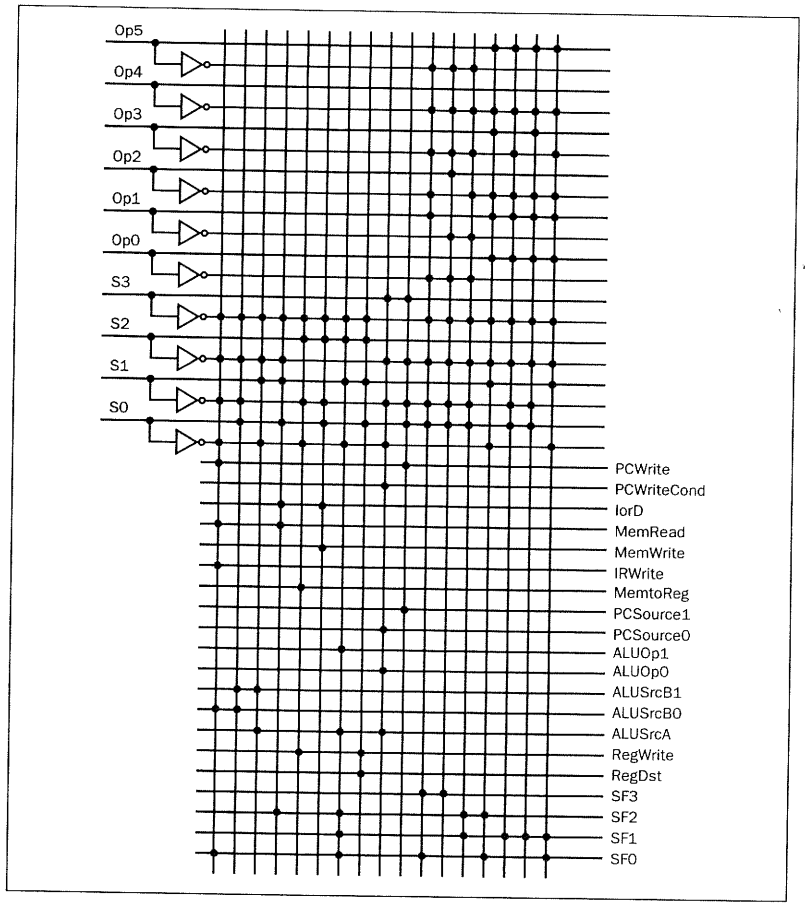


Figura C.14 Questa PLA realizza la logica per la funzione di controllo dell'implementazione multiciclo. Gli ingressi della logica di controllo compaiono sulla sinistra e le uscite sulla destra. La metà superiore della figura costituisce il piano AND che provvede al calcolo dei mintermini; i mintermini sono trasferiti al piano OR sulle linee verticali. Ciascun punto colorato corrisponde ad un segnale che costituisce il mintermine corrispondente a quella linea. I termini somma sono calcolati a partire da questi mintermini, e ciascun punto grigio rappresenta la presenza del mintermine intersezione all'interno di quel termine somma. Ciascuna uscita consiste di un singolo termine somma.

comprendesse un numero maggiore di tipi diversi di istruzioni, e di queste alcune richiedessero un numero elevato di cicli di clock per essere eseguite: vi sarebbero molti più stati nella macchina a stati finiti. In corrispondenza di alcuni stati potrebbe essere possibile passare ad un numero elevato di stati diversi a seconda del tipo di istruzione, come succede per lo stato 1 della macchina a stati finiti della figura C.6. Tuttavia in corrispondenza di molti stati si procede in maniera sequenziale, come succede per gli stati 3 e 4 nella figura C.6.

Ad esempio se si includesse l'aritmetica in virgola mobile si verificherebbero sequenze di molti stati che implementano un'istruzione multiciclo in virgola mobile. In alternativa si consideri come potrebbe apparire la logica di controllo per una macchina che prevedesse operandi multipli in memoria per ciascuna istruzione: essa richiederebbe molti più stati per il caricamento di tali operandi in memoria. Il risultato di questa situazione sarebbe una logica di controllo dominata dalla codifica della funzione di stato futuro. Inoltre gran parte della logica sarebbe destinata alle sequenze di stati caratterizzate dall'essere percorribili secondo un unico cammino, come per gli stati dal 2 al 4 nella figura C.6. Con un numero maggiore di istruzioni questo comportamento sequenziale sarebbe molto più frequente di quanto non accada con il semplice sottoinsieme considerato.

Per codificare in maniera efficiente queste funzioni di controllo più complesse si può usare un'unità di controllo che comprende un contatore per fornire in maniera sequenziale lo stato futuro; questo contatore è spesso in grado di eliminare la necessità della codifica esplicita della funzione di stato futuro all'interno dell'unità di controllo. Come è mostrato nella figura C.15, per incrementare lo stato si usa un sommatore, in sostanza trasformando il registro di stato in un contatore; lo stato

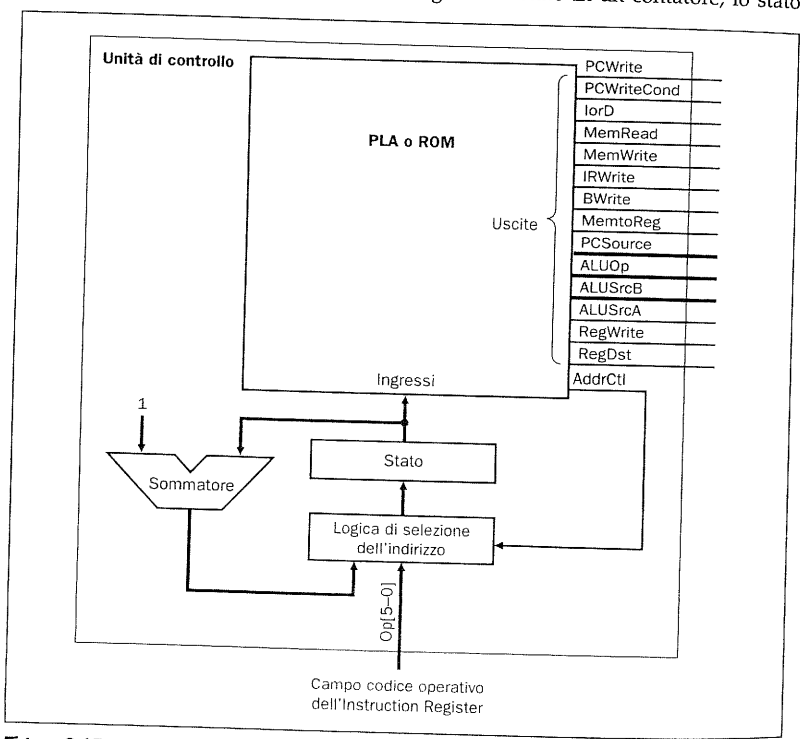


Figura C.15 L'unità di controllo basata su un contatore per il calcolo dello stato futuro. In questa unità di controllo lo stato futuro è calcolato usando un contatore (almeno in alcuni stati). Si confronti con la figura C.7, nella quale per ogni stato lo stato futuro è codificato nella logica di controllo. In questa unità di controllo i segnali etichettati *AddrCtl* controllano il modo in cui è determinato lo stato futuro.

dopo l'incremento corrisponde sempre allo stato con il numero successivo. Tuttavia la macchina a stati finiti a volte esegue dei "salti". Ad esempio nello stato 1 della macchina a stati finiti della figura C.6 vi sono quattro possibili stati futuri dei quali solo uno è lo stato successivo in ordine sequenziale. Di conseguenza è necessario essere in grado di scegliere tra lo stato dopo l'incremento e un diverso stato futuro sulla base degli ingressi provenienti dall'Instruction Register e dello stato corrente. Ciascuna parola di controllo includerà delle linee di controllo che determinano come viene scelto lo stato futuro.

È facile implementare la parte della parola di controllo che pilota il segnale di uscita dal momento che, se si usano gli stessi numeri per gli stati, questa parte della parola di controllo è esattamente identica al contenuto della ROM della figura C.12. Tuttavia il modo per selezionare lo stato futuro differisce dalla funzione di stato futuro adottata nella macchina a stati finiti.

Se si usa un contatore esplicito per fornire lo stato futuro secondo la sequenza predeterminata, la logica dell'unità di controllo richiede soltanto che si specifichi come selezionare lo stato quando non è quello sequenzialmente successivo. Vi sono due modi per fare questo. Il primo lo si è già visto: l'unità di controllo codifica esplicitamente la funzione di stato futuro; la differenza sta soltanto nel fatto che l'unità di controllo richiede che le linee di stato futuro vengano forzate quando lo stato futuro prescelto non corrisponde allo stato indicato dal contatore. Se il numero di stati è grande e la funzione di stato futuro necessaria è praticamente vuota, questa scelta può risultare sbagliata, poiché l'unità di controllo risultante conterrà molto spazio vuoto o ridondante. Un approccio alternativo consiste nell'utilizzare della logica esterna per specificare lo stato futuro quando il contatore non indica lo stato corretto. Molte unità di controllo, in particolare quelle che implementano set di istruzioni di grandi dimensioni, usano questo approccio. Ci si concentrerà ora proprio su come specificare dall'esterno le informazioni di controllo.

Anche se lo stato futuro, quando non è in sequenza, proviene da una tabella esterna, l'unità di controllo deve specificare quando questo deve avvenire e come determinare lo stato futuro. Vi sono due tipi di "salti" che si devono implementare nella logica di selezione dell'indirizzo: innanzitutto si deve poter saltare ad uno fra un certo numero di stati sulla base della parte relativa al codice operativo all'interno dell'Instruction Register. Questa operazione, denominata *smistamento* (o dispatch), viene normalmente implementata tramite un insieme apposito di ROM o PLA che fanno parte della logica di selezione dell'indirizzo. Un insieme ulteriore di uscite di controllo, che sono state chiamate AddrCtl, indica quando si deve eseguire un'operazione di smistamento. Se si osserva il diagramma a stati finiti della figura C.6 si vede che vi sono due stati nei quali si deve eseguire un salto sulla base del valore di parte del codice operativo. Di conseguenza è necessario disporre di due piccole tabelle di smistamento. In alternativa si potrebbe anche usare una singola tabella di smistamento ed usare i bit di controllo come bit di indirizzo che selezionano la parte di tabella di smistamento dalla quale estrarre l'indirizzo.

Il secondo tipo di salto che deve essere implementato consiste nel salto all'indietro allo stato 0, tramite il quale si inizia l'esecuzione della istruzione MIPS successiva. Vi sono quindi quattro possibili modi per scegliere lo stato futuro (tre tipi di salto, oltre all'incremento del numero dello stato corrente), che possono essere codificati in 2 bit. Si assuma che la codifica sia la seguente:

Valore di AddrCtl	Azione
0	Forza lo stato a 0
1	Esegui lo smistamento con la ROM 1
2	Esegui lo smistamento con la ROM 2
3	Usa lo stato incrementato

Se si usa questa codifica la logica di selezione dell'indirizzo per l'unità di controllo considerata può essere implementata come mostrato in figura C.16.

Per completare l'unità di controllo è sufficiente specificare il contenuto delle ROM di smistamento, nonché i valori delle linee di controllo degli indirizzi per ciascuno stato. La parte di parola di controllo destinata al controllo dell'unità di elaborazione è già stata specificata attraverso il contenuto della ROM di figura C.12 (o delle corrispondenti parti della PLA della figura C.14). Il contatore di stato futuro e le ROM di smistamento svolgono il ruolo della parte di unità di controllo che calcolava lo stato futuro, e che era mostrata nella figura C.13. Si sta implementando soltanto una parte del set di istruzioni, e quindi le ROM di smistamento saranno in buona parte vuote. La figura C.17 mostra gli elementi che devono assumere un valore specificato relativamente a tale sottoinsieme; il Paragrafo 5.6 discute cosa fare con gli elementi delle ROM di smistamento che non corrispondono ad alcuna istruzione.

È ora possibile determinare il valore delle linee di selezione dell'indirizzo (AddrCtl) in ciascuna parola di controllo; la tabella in figura C.18 mostra il valore che deve assumere il segnale di controllo dell'indirizzo in corrispondenza di ciascuno stato. Questa informazione verrà utilizzata per specificare il contenuto del campo AddrCtl nella parola di controllo associata ad uno stato.

Il contenuto dell'intera ROM di controllo è mostrato nella figura C.19. La memoria complessiva richiesta per la funzione di controllo è relativamente piccola: vi

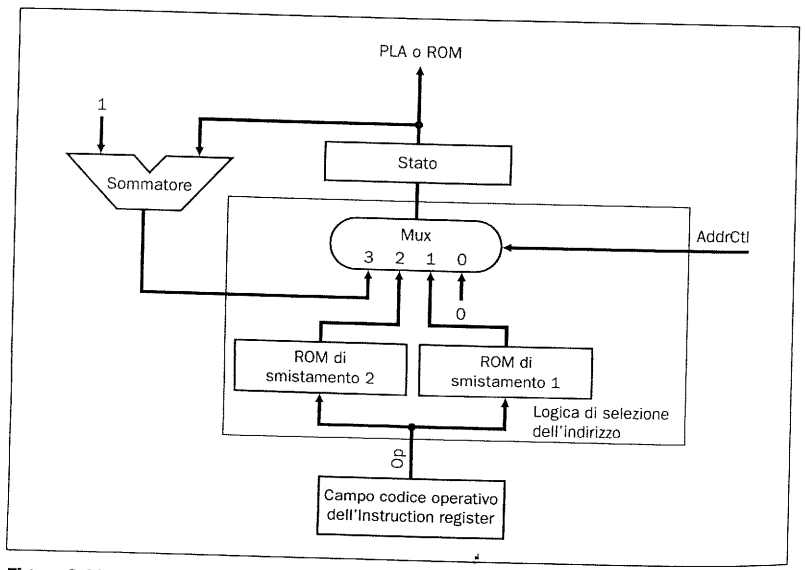


Figura C.16 La logica di selezione dell'indirizzo per l'unità di controllo di figura C.15.

ROM di smistamento 1		
Op	Nome del codice operativo	Valore
000000	Formato-R	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

ROM di smistamento 2		
Op	Nome del codice operativo	Valore
100011	lw	0011
101011	sw	0101

Figura C.17 Le ROM di smistamento contengono ciascuna 2^6 = 64 righe di ampiezza pari a 4 bit, poiché questo è il numero di bit nella codifica dello stato. Questa figura mostra solo le righe della ROM che risultano di interesse rispetto al sottoinsieme trattato. La prima colonna in ciascuna tabella indica il valore di Op, corrispondente all'indirizzo usato per accedere alla ROM di smistamento. La seconda colonna mostra il nome simbolico del codice operativo. La terza colonna indica il valore presente nella ROM a tale indirizzo.

Numero dello stato	Azione di controllo dell'indirizzo	Valore di AddrCtl
0	Usa lo stato incrementato	3
1	Usa la ROM di smistamento 1	1
2	Usa la ROM di smistamento 2	2
3	Usa lo stato incrementato	3
4	Sostituisci il numero dello stato con 0	0
5	Sostituisci il numero dello stato con 0	0
6	Usa lo stato incrementato	3
7	Sostituisci il numero dello stato con 0	0
8	Sostituisci il numero dello stato con 0	0
9	Sostituisci il numero dello stato con 0	0

Figura C.18 I valori delle linee di controllo dell'indirizzo sono determinati all'interno della parola di controllo corrispondente a ciascuno stato.

Numero dello stato	Bit 17-2 della parola di controllo	Bit 1-0 della parola di controllo
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000000	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

Figura C.19 Il contenuto della memoria di controllo in un'implementazione basata su un contatore esplicito. La prima colonna mostra lo stato mentre la seconda mostra i bit di controllo dell'unità di elaborazione; l'ultima colonna mostra i bit di controllo dell'indirizzo in ciascuna parola di controllo. I bit 17-2 sono identici a quelli della figura C.12.

sono dieci parole di controllo, ciascuna di ampiezza pari a 18 bit, per un totale di 180 bit. Inoltre le due tabelle di smistamento hanno ampiezza pari a 4 bit e contengono 64 righe, per un totale di ulteriori 512 bit; il numero complessivo, pari a 692 bit, è inferiore a quello dell'implementazione basata su due ROM, nella quale la funzione di stato futuro era codificata nelle ROM (per un totale di 4,3 Kbit).

Ovviamente le tabelle di smistamento sono poco dense e possono essere implementate in maniera più efficiente con due PLA di piccole dimensioni; anche la ROM di controllo può essere implementata tramite una PLA.

Ottimizzazione della logica di controllo

Si può ulteriormente ridurre la quantità di logica presente nell'unità di controllo facendo ricorso a due diverse tecniche. La prima è la *minimizzazione logica* che utilizza la struttura delle equazioni logiche (tenendo anche conto dei termini *don't care*) per ridurre la quantità di logica necessaria. Il successo di questo processo dipende dal numero di righe esistenti nella tabella di verità e da come queste righe sono tra loro correlate. Nel sottoinsieme che si sta considerando, ad esempio, solo i codici operativi `lw` e `sw` prevedono un valore attivo per il segnale Op5, e si possono quindi sostituire le due righe della tabella di verità che verificano se l'ingresso è `lw` o `sw` con un unico controllo su tale bit; analogamente è possibile eliminare diversi bit usati per accedere alle ROM di smistamento dal momento che un unico bit può essere utilizzato per individuare `lw` o `sw` nella prima ROM di smistamento. Ovviamente se lo spazio dei codici operativi fosse meno sparso le possibilità di realizzare una ottimizzazione di questo tipo sarebbero molto più difficili da localizzare. Tuttavia nella scelta dei codici operativi il progettista può far nascere delle ulteriori occasioni di ottimizzazione scegliendo dei codici operativi correlati per le istruzioni che probabilmente condivideranno gli stessi stati nella funzione di controllo.

Si può eseguire un tipo diverso di ottimizzazione assegnando in maniera opportuna i numeri degli stati in una realizzazione tramite macchina a stati finiti o microcodice. Questa ottimizzazione, denominata *assegnazione degli stati*, mira a scegliere i numeri degli stati in maniera che le equazioni logiche risultanti contengano più ridondanza e possano quindi essere semplificate. Si consideri il caso di una macchina a stati finiti nella quale la funzione di stato futuro sia codificata cosicché gli stati possano essere assegnati arbitrariamente. Si noti ad esempio che in una macchina a stati finiti il segnale RegWrite è attivo solo negli stati 4 e 7; se si codificano tali stati come 8 e 9, anziché come 4 e 7, è possibile riscrivere l'equazione che descrive RegWrite semplicemente come un controllo sul bit S3 (che vale 1 soltanto per gli stati 8 e 9). Questa rinumerazione permette di combinare le righe delle due tabelle di verità nella parte (o) della figura C.9 e di sostituirle con una sola riga, eliminando così un termine nell'unità di controllo. Ovviamente si dovranno rinumerare i vecchi stati 8 e 9, eventualmente come 4 e 7.

La stessa ottimizzazione può essere applicata in un'implementazione che usa un contatore di programma esplicito, anche se in tal caso vi sono ulteriori restrizioni. Poiché il numero dello stato futuro viene spesso calcolato incrementando il numero dello stato corrente, non è possibile assegnare arbitrariamente i numeri agli stati. Tuttavia se si mantengono inalterati gli stati nei quali lo stato incrementato è utilizzato nello stesso ordine come stato futuro, è possibile riassegnare in blocco i numeri di stati consecutivi. In un'implementazione basata su un contatore esplicito di stato futuro, l'assegnazione degli stati può permettere di semplificare il contenuto delle ROM di smistamento.

Se si torna all'unità di controllo della figura C.15, nella sua parte destra essa ha una struttura notevolmente simile a quella di un calcolatore. La ROM o la PLA possono essere viste come delle memorie che forniscono istruzioni all'unità di elaborazione, mentre lo stato può essere visto come l'indirizzo dell'istruzione. Di qui deriva l'origine del nome *microcodice* o *controllo microprogrammato*: le parole di controllo

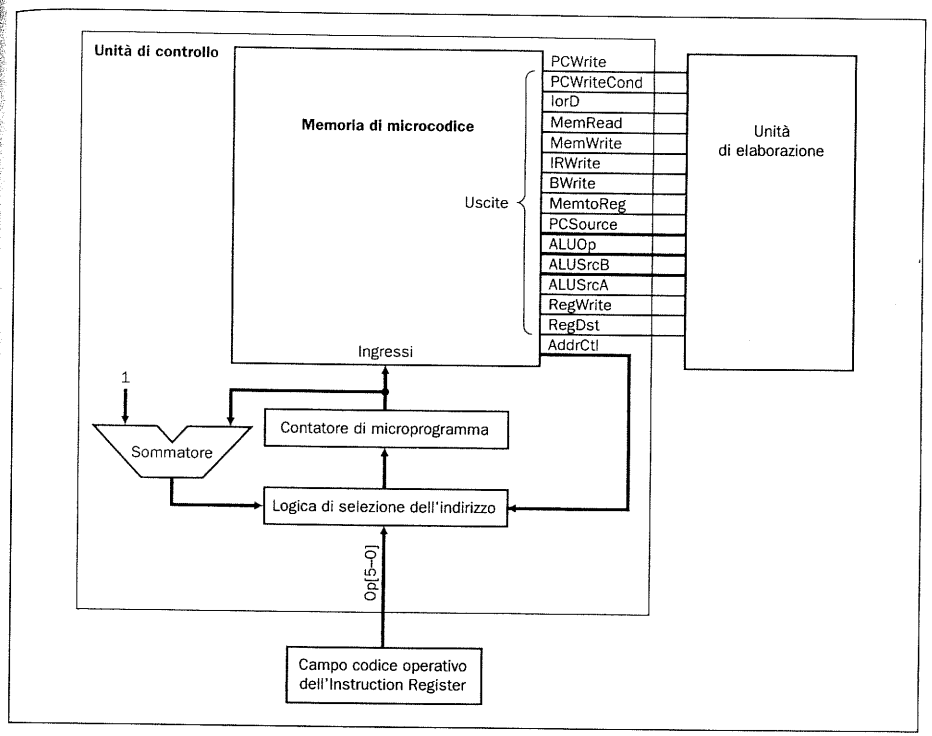


Figura C.20 L'unità di controllo come microcodice. L'uso della parola "micro" serve a distinguere tra il contatore di programma nell'unità di elaborazione e il contatore di microprogramma, e tra la memoria di microcodice e la memoria delle istruzioni.

possono essere considerate come *microistruzioni* che controllano l'unità di elaborazione ed il registro di stato viene chiamato *contatore di microprogramma*. La figura C.20 presenta l'unità di controllo come *microcodice*; il prossimo paragrafo descrive come si passa da un microprogramma al microcodice.

C.5 Tradurre in hardware un microprogramma

Per tradurre il microprogramma del Paragrafo 5.5 in un circuito reale si deve specificare la traduzione di ciascun campo nei corrispondenti segnali di controllo. È possibile implementare il microprogramma sia sotto forma di unità di controllo a stati finiti, sia come microcodice basato su un sequenzializzatore esplicito. Se si sceglie la soluzione basata su macchina a stati finiti è necessario costruire la funzione di stato futuro a partire dal microprogramma; una volta che è nota questa funzione si possono mettere in corrispondenza le righe delle tabelle di verità con le uscite di stato futuro. In questo paragrafo si vedrà come tradurre il microprogramma assumendo che lo stato futuro sia specificato attraverso un sequenzializzatore; a partire

dalle tabelle di verità che verranno costruite sarà immediato dedurre la funzione di stato futuro per una macchina a stati finiti.

Assumendo di disporre di un sequenzializzatore esplicito, è necessario eseguire due operazioni ulteriori per tradurre il microprogramma: assegnare gli indirizzi alle microistruzioni e definire il contenuto delle ROM di smistamento. Questo processo coincide sostanzialmente con quello di traduzione di un programma scritto in linguaggio assembler nelle istruzioni macchina: si traducono i campi del linguaggio assembler o le istruzioni del microcodice, e si trasformano in indirizzi le etichette associate alle istruzioni.

La figura C.21 mostra i vari valori per ogni campo della microistruzione destinato a controllare l'unità di elaborazione e come questi campi siano codificati in segnali di controllo. Se un campo che corrisponde ad un segnale che influenza un modulo dotato di stato (come una memoria, un registro di memoria, il registro destinazione della ALU, il Program Counter) è vuoto, allora non si deve attivare nessun segnale di controllo. Se un campo corrispondente ad un segnale di controllo di un multiplexer o ad un segnale di controllo dell'operazione svolta dalla ALU (ad esempio ALUOp, Src1, Src2) è vuoto, l'uscita non è utilizzata, e quindi i relativi segnali possono assumere il valore *don't care*.

Il campo di sequenzializzazione può assumere quattro valori: Preleva (significa andare allo stato di caricamento di un'istruzione), Smista 1, Smista 2 e Seq. Questi quattro valori sono codificati in maniera da far assumere ai 2 bit di controllo dell'indirizzo i valori previsti nella figura C.18: Preleva = 0, Smista 1 = 1, Smista 2 = 2, Seq = 3. Infine si deve specificare il contenuto delle tabelle di smistamento per correlare le informazioni di smistamento del campo di sequenzializzazione con le etichette simboliche del microprogramma. Si useranno le stesse tabelle di smistamento utilizzate in precedenza nella figura C.17.

Per generare le microistruzioni un assemblatore di microcodice utilizzerà la codifica del campo di sequenzializzazione, il contenuto delle tabelle simboliche di smistamento della figura C.22, le specifiche della figura C.21 ed il microprogramma della figura 5.46. Poiché il microprogramma è una rappresentazione astratta della funzione di controllo, vi è un buon livello di flessibilità nel modo in cui esso può essere tradotto. Ad esempio l'indirizzo assegnato a molte microistruzioni può essere scelto arbitrariamente; le uniche restrizioni sono quelle imposte dal fatto che certe microistruzioni devono presentarsi in ordine sequenziale (così che incrementando il registro di stato si genera l'indirizzo dell'istruzione successiva). L'assemblatore di microcodice può quindi ridurre la complessità della funzione di controllo tramite un'oculata assegnazione delle microistruzioni.

Organizzare il controllo per ridurre la logica

Nel caso di un elaboratore caratterizzato da una funzione di controllo complessa l'ammontare di logica all'interno dell'unità di controllo può essere cospicuo. La ROM o la PLA di controllo possono essere molto costose. Anche se la semplice implementazione vista prevede una microistruzione di soli 18 bit (se si assume che vi sia un sequenzializzatore esplicito) si sono costruiti elaboratori aventi microistruzioni di ampiezza pari a centinaia di bit. È chiaro che un progettista desidererebbe ridurre il numero delle microistruzioni e la loro ampiezza.

L'approccio ideale alla riduzione della memoria di controllo consiste innanzitutto nello scrivere il microprogramma completo in una notazione simbolica e poi nel misurare quante linee di controllo sono forzate all'interno di ciascuna microistruzione. Tramite queste misure è possibile identificare dei bit di controllo che possono essere codificati in un campo più piccolo. Per esempio se non più di una linea all'interno di un sottoinsieme di 8 è forzata ad ogni istante in una stessa microistruzione, allora questo sottoinsieme di linee di controllo può essere codificato in un campo di 3 bit ($\log_2 8 = 3$). Questa modifica permette di risparmiare 5 bit in ciascuna microi-

Nome del campo	Valore	Segnali attivi	Commenti
Controllo della ALU	Add	ALUOp=00	Forza la ALU ad eseguire la somma.
	Subt	ALUOp=01	Forza la ALU ad eseguire la sottrazione; si implementa così anche il confronto per i salti.
	Codice func	ALUOp=10	Usa il codice funzione dell'istruzione per determinare il controllo della ALU.
SRC1	PC	ALUSrcA=0	Usa il PC come primo ingresso della ALU.
	A	ALUSrcA=1	Il registro A è il primo ingresso della ALU.
SRC2	B	ALUSrcB=00	Il registro B è il secondo ingresso della ALU.
	4	ALUSrcB=01	Si usa 4 come secondo ingresso della ALU.
	Extend	ALUSrcB=10	Si usa l'uscita dell'unità di estensione del segno come secondo ingresso della ALU.
	Extshft	ALUSrcB=11	Si usa l'uscita dell'unità di scalamento di due posizioni come secondo ingresso della ALU.
Controllo del registro	Lettura		Si leggono due registri usando i campi rs e rt di IR come numeri dei registri e si mette il dato nei registri A e B.
	Scrittura ALU	RegWrite, RegDst=1, MemtoReg=0	Si scrive un registro usando il campo rd di IR come numero del registro e il contenuto di ALUOut come dato.
	Scrittura MDR	RegWrite, RegDst=0, MemtoReg=1	Si scrive un registro usando il campo rt di IR come numero del registro e il contenuto di MDR come dato.
Memoria	Lettura PC	MemRead, lorD=0	Si legge la memoria usando PC come indirizzo; si scrive il risultato in IR (e in MDR).
	Lettura ALU	MemRead, lorD=1	Si legge la memoria usando ALUOut come indirizzo; si scrive il risultato in MDR.
	Scrittura ALU	MemWrite, lorD=1	Si scrive in memoria usando ALUOut come indirizzo, e il contenuto di B come dato.
Controllo della scrittura di PC	ALU	PCSource=00, PCWrite	Si scrive l'uscita della ALU in PC.
	ALUOut-cond	PCSource=01, PCWriteCond	Se è attiva l'uscita Zero della ALU si scrive in PC il contenuto del registro ALUOut.
	Indirizzo di salto	PCSource=10, PCWrite	Si scrive in PC l'indirizzo di salto contenuto nell'istruzione.
Sequenzializzazione	Seq	AddrCtl=11	Si sceglie l'istruzione successiva in sequenza.
	Preleva	AddCtl=00	Si va alla prima microistruzione che inizia una nuova istruzione.
	Smista 1	AddrCtl=01	Si esegue lo smistamento usando la ROM 1.
	Smista 2	AddrCtl=10	Si esegue lo smistamento usando la ROM 2.

Figura C.21 Ciascun campo del microcodice viene tradotto nell'insieme di segnali di controllo che devono venire forzati ad un certo valore. Questa tabella specifica un valore per ciascuno dei campi originariamente specificati nella figura 5.46. Questi 22 diversi valori dei vari campi specificano tutte le combinazioni necessarie delle 18 linee di controllo; le linee di controllo che non sono attivate e che corrispondono a delle azioni vengono messe a 0. Le linee di controllo dei multiplexer, se la loro uscita ha importanza, sono forzate a 0. Se una linea di controllo di un multiplexer non è esplicitamente specificata, allora la sua uscita è *don't care* e non è quindi usata.

Tabella di smistamento del microcodice 1		
Campo codice operativo	Nome del codice operativo	Valore
000000	Formato-R	Rformat1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

Tabella di smistamento del microcodice 2		
Campo codice operativo	Nome del codice operativo	Valore
100011	lw	LW2
101011	sw	SW2

Figura C.22 Il microcodice delle due ROM di smistamento nel formato simbolico e usando le etichette presenti nel microcodice.

struzione e non peggiora il CPI, anche se implica un costo hardware ulteriore legato al decodificatore da 3 a 8 necessario per generare le 8 linee di controllo quando queste sono richieste dall'unità di elaborazione. L'operazione può anche avere qualche impatto sul ciclo di clock, visto che il decodificatore si trova sul cammino dei segnali, tuttavia eliminare 5 bit dalla memoria di controllo è un vantaggio di solito superiore al costo del decodificatore, e l'impatto sul tempo di ciclo è normalmente piccolo o inesistente. Questa tecnica può ad esempio essere applicata ai bit 14-9 delle microistruzioni dell'elaboratore considerato, dal momento che solo 1 bit alla volta è attivo tra questi 7 all'interno della parola di controllo (si veda la figura C.19).

La tecnica esposta per la riduzione dell'ampiezza di un campo è chiamata *codifica* (encoding). Per risparmiare ancora altro spazio le linee di controllo possono essere codificate assieme se solo occasionalmente sono asserite all'interno della stessa microistruzione; in questo caso è necessario avere due microistruzioni anziché una quando entrambe le linee devono essere asserite. Se questo non avviene all'interno di procedure critiche, il fatto di avere delle microistruzioni più strette può giustificare la necessità di avere qualche parola in più nella memoria di controllo.

Le microistruzioni possono essere rese ancora più compatte se sono spezzate in diversi formati e si assegna loro un codice operativo o campo *formato* per distinguerle. Il campo formato fornisce il valore di default per tutte le linee di controllo che non sono specificate, così da non dover cambiare nient'altro all'interno dell'elaboratore, ed è analogo al codice operativo di un'istruzione all'interno di un set di istruzioni più potente. Si potrebbe ad esempio usare un formato diverso per le microistruzioni che eseguono accessi in memoria e per quelle che eseguono operazioni tra registri tramite la ALU, traendo vantaggio dal fatto che le linee di controllo per l'accesso alla memoria non sono necessarie nelle microistruzioni che controllano le operazioni della ALU.

La riduzione del costo dell'hardware attraverso l'uso dei campi formato comporta normalmente un costo aggiuntivo in termini di prestazioni, che si aggiunge al costo dei decodificatori. Un microprogramma che utilizza un solo formato per le microistruzioni può specificare una qualunque combinazione di operazioni per l'unità di elaborazione e può impiegare un numero inferiore di cicli di clock rispetto ad un microprogramma costituito da microistruzioni più corte che non possono eseguire una qualunque combinazione di operazioni all'interno di una singola microistruzione. Tuttavia se non si usano appieno le potenzialità delle parole di microcodice più ampie, allora buona parte della memoria di controllo è sprecata e l'elaboratore può essere reso più piccolo e più veloce restringendo le potenzialità delle singole microistruzioni.

L'approccio basato su un microcodice più stretto ma normalmente più lungo da eseguire è chiamato spesso *microcodice verticale*, mentre l'approccio basato su un microcodice più ampio ma anche più veloce è chiamato *microcodice orizzontale*. Va notato che i termini "microcodice verticale" e "microcodice orizzontale" non sono universalmente accettati: i progettisti dell'8086 consideravano la sua microistruzione di 21 bit più orizzontale di quanto non fossero quelle di altri calcolatori su chip singolo realizzati in quel periodo. I termini analoghi *massimamente codificato* e *minimamente codificato* sono probabilmente da preferirsi rispetto a verticale e orizzontale.

C.6 Note conclusive

Questa appendice è iniziata con l'analisi su come tradurre un diagramma a stati finiti in un'implementazione basata su una macchina a stati finiti; sono poi stati presentati i sequenzializzatori espliciti che utilizzano una diversa tecnica per realizzare la funzione di stato futuro. Anche se i grandi microprogrammi sono spesso finalizzati a implementazioni che usano questo approccio esplicito per il calcolo dello stato futuro, è anche possibile implementare un microprogramma con una macchina a stati finiti. Come si è visto sono possibili sia implementazioni delle funzioni logiche tramite ROM che tramite PLA; i vantaggi della forma esplicita dello stato futuro rispetto a quella codificata e quelli dell'implementazione tramite ROM o tramite PLA sono riassunti di seguito.

Quadro d'insieme

Indipendentemente dal fatto che la funzione di controllo sia rappresentata come un diagramma a stati finiti o come un microprogramma, la traduzione in una implementazione hardware è simile. Ciascuno stato o microistruzione attiva un insieme di uscite di controllo e specifica come scegliere lo stato futuro.

La funzione di stato futuro può essere implementata attraverso la sua codifica in una macchina a stati finiti o utilizzando un sequenzializzatore esplicito. Il sequenzializzatore esplicito è più efficiente se il numero di stati è grande e vi sono numerose sequenze composte da stati consecutivi, senza salti.

La logica di controllo può essere implementata tramite ROM o PLA (o tramite una combinazione delle due soluzioni). Le PLA sono più efficienti a meno che la funzione di controllo non sia molto densa; le ROM possono essere la soluzione più adeguata se la funzione di controllo è contenuta in una memoria a parte anziché sullo stesso chip dell'unità di elaborazione.

C.7 Parole chiave

I termini più importanti di questa appendice sono qui elencati e sono poi definiti nel Glossario.

assegnazione degli stati
assemblatore di microcodice
contatore di stato futuro
ingressi di stato
massimamente codificato
microcodice orizzontale
microcodice verticale
minimamente codificato
minimizzazione logica
uscite di stato futuro

C.8 Esercizi

C.1 [10] <§C.2> Anziché usare 4 bit di stato per implementare la macchina a stati finiti della figura C.6, si usino 9 bit di stato, ciascuno dei quali vale 1 solo se la macchina a stati finiti si trova in quel particolare stato. Ad esempio S1 vale 1 nello stato 1, S2 vale 1 nello stato 2, e così via. Si ridisegni la PLA della figura C.14.

C.2 [5] <§C.3> {Es. 5.6} Quanti termini prodotto ci vogliono in una PLA che implementa l'unità di elaborazione a ciclo singolo per `jal`, tenendo conto delle aggiunte descritte nell'Esercizio 5.6?

C.3 [5] <§C.3> {Es. 5.5} Quanti termini prodotto ci vogliono in una PLA che implementa l'unità di elaborazione a ciclo singolo e la funzione di controllo per `addiu`, tenendo conto delle aggiunte descritte nell'Esercizio 5.5?

C.4 [10] <§C.3> {Es. 5.27} Si determini il numero di termini prodotto in una PLA che implementa la macchina a stati finiti per l'istruzione `jal` costruita nell'Esercizio 5.27. Il modo più semplice per risolvere il problema consiste nel costruire le tabelle di verità corrispondenti a ciascuna nuova uscita e a ciascuna uscita modificata in seguito all'aggiunta.

C.5 [10] <§C.3> {Es. 5.15} Si determini il numero di termini prodotto in una PLA che implementa la macchina a stati finiti per l'istruzione `addi` costruita nell'Esercizio 5.15. Il modo più semplice per risolvere il problema consiste nel costruire le tabelle di verità per `addi`.

C.6 [20] <§C.4> {Es. 5.15} Si implementi la macchina a stati finiti dell'Esercizio 5.15 usando un contatore esplicito per la determinazione dello stato futuro. Si proceda inserendo nella figura C.19 i nuovi elementi relativi alle aggiunte. Si aggiungano inoltre gli elementi necessari nelle ROM di smistamento della figura C.22.

C.7 [15] <§§C.3-C.6> Si determini la dimensione delle PLA necessarie all'implementazione della macchina multiciclo del Capitolo 5 assumendo che la funzione di stato futuro sia implementata tramite un contatore. Si implementino le tabelle di smistamento della figura C.22 usando due PLA, ed il contenuto dell'unità di controllo principale della figura C.19 usando un'ulteriore PLA. In quale rapporto stanno la dimensione totale di questa soluzione rispetto a quella della soluzione basata su una singola PLA in cui si codifica lo stato futuro? Cosa accade se le PLA principali dei due approcci vengono divise in due PLA separate tramite la fattorizzazione dei segnali di stato futuro o di quelli di selezione degli indirizzi?

Glossario

abilitazione delle eccezioni Anche detta abilitazione delle interruzioni. Segnale o azione che controlla se un processo debba rispondere o meno alle eccezioni; necessaria a prevenire il verificarsi di un'eccezione durante gli intervalli precedenti il salvataggio dello stato da parte del processore; è una procedura indispensabile perché l'esecuzione possa essere ripresa.

abilitazione delle interruzioni *Vedere* abilitazione delle eccezioni.

accesso diretto alla memoria (DMA) Meccanismo con il quale il controllore di un dispositivo ha la capacità di trasferire dati da e verso la memoria senza coinvolgere il processore.

accesso in memoria non uniforme (NUMA) Tipo di sistema multiprocessore con un unico spazio di indirizzamento in cui alcuni accessi alla memoria sono più veloci di altri, a seconda del processore che fa richiesta e della parola di memoria desiderata.

accesso in memoria uniforme (UMA) Accesso alla memoria che richiede la stessa quantità di tempo qualunque sia il processore che fa la richiesta e qualunque sia la parola desiderata.

algoritmo di Booth Algoritmo basato sull'osservazione che a partire dalle operazioni di somma e sottrazione si possono calcolare i prodotti in più modi; considerando sequenze di bit è spesso possibile risparmiare operazioni aritmetiche.

aliasing Situazione in cui si accede allo stesso oggetto tramite due indirizzi diversi; può verificarsi nella memoria virtuale quando la stessa pagina fisica ha due indirizzi virtuali.

ALU *Vedere* unità aritmetico logica (ALU).

arbitraggio centralizzato parallelo Schema di arbitraggio del bus che utilizza più linee di richiesta con le quali i dispositivi possono richiedere l'uso del bus in modo indipendente, e che usa un arbitro centralizzato per scegliere quale dei dispositivi che ha richiesto il bus avrà l'accesso e per notificare a tale dispositivo che ora è il bus master.

arbitraggio del bus Processo in cui si decide quale bus master avrà la possibilità di usare il bus nel ciclo successivo.

arbitraggio distribuito tramite autoselezione Schema di arbitraggio del bus che fa sì che i dispositivi che richiedono l'accesso al bus abbiano la capacità di determinare quale dispositivo otterrà tale accesso: ciascun richiedente può sapere se sarà lui ad ottenere l'allocazione del bus.

arbitraggio distribuito tramite rilevamento delle collisioni Schema di arbitraggio del bus che consente a ciascun dispositivo di richiedere l'accesso al bus indipendentemente dagli altri e che usa uno schema per ripetere l'arbitraggio nel caso di più richieste simultanee.

arbitraggio in catena Schema di arbitraggio del bus in cui la linea di assegnazione del bus attraversa tutti i dispositivi, da quello con priorità più alta a quello con priorità più bassa (le priorità sono determinate dalla posizione sul bus), in modo che quando c'è una richiesta di bus il dispositivo a priorità maggiore è il primo a ricevere il segnale di assegnazione del bus.

architettura *Vedere* architettura del set di istruzioni.

architettura del set di istruzioni Anche detta architettura. Interfaccia astratta fra l'hardware ed il livello più basso del software di un calcolatore, che comprende tutte le informazioni necessarie per scrivere un programma in linguaggio macchina che sia eseguito in modo corretto, incluse le istruzioni, i registri, le dimensioni della memoria e così via.

area del die Dimensione del die.

arrotondamento Metodo per adattare i risultati intermedi delle operazioni in virgola mobile al formato della virgola mobile; l'obiettivo in generale è quello di trovare il numero più vicino fra tutti quelli rappresentabili.

assegnazione degli stati Ottimizzazione dell'unità di controllo volta ad assegnare un numero ai vari stati in modo che le equazioni logiche risultanti contengano ridondanze e possano quindi essere semplificate.

assemblatore Programma che traduce la versione simbolica di un'istruzione nella corrispondente versione binaria.

assemblatore di microcodice Programma che traduce i microprogrammi in microistruzioni implementabili in una ROM o in una PLA.

astrazione Modello in cui i dettagli di basso livello dei sistemi di elaborazione sono resi invisibili; è utile per facilitare il progetto di sistemi sofisticati.

backpatching Metodo per tradurre da linguaggio assembler a codice macchina, in cui l'assemblatore costruisce una rappresentazione binaria (a volte incompleta) di ciascuna istruzione con una prima passata sul programma, e torna indietro in un secondo tempo per completarla inserendo i riferimenti alle etichette precedentemente non definite.

benchmark SPEC *Vedere* benchmark System Performance Evaluation Cooperative (SPEC).

benchmark System Performance Evaluation Cooperative (SPEC) Insieme di benchmark standard interi e in virgola mobile, che sfruttano molto la CPU, basati su programmi reali.

bit *Vedere* cifra binaria.

bit di riferimento Anche detto bit di uso. Campo posto a 1 ogni volta che si accede ad una pagina, e che è usato per implementare uno schema LRU o un altro schema di sostituzione.

bit di uso *Vedere* bit di riferimento.

bit di validità Campo di una tabella delle gerarchie di memoria, che indica che il blocco associato contiene dati validi.

bit meno significativo Il bit più a destra in una parola MIPS.

bit più significativo Il bit più a sinistra in una parola MIPS.

blocco Minima unità di informazione che può essere presente o meno nella gerarchia a due livelli.

blocco base Sequenza di istruzioni che non contiene né salti condizionati (eccetto eventualmente alla fine) né etichette di destinazione di salti (eccetto eventualmente all'inizio).

branch delay slot Lo slot che si trova immediatamente dopo un'istruzione di salto condizionato ritardata, che nel caso dell'architettura MIPS è riempito con un'istruzione che non influenza il salto.

broadcast in scrittura Protocollo di snooping secondo cui il processore che effettua l'operazione di scrittura invia i dati sul bus, permettendo l'aggiornamento dei valori di tutte le copie.

buco *Vedere* stallo della pipeline.

buffer di previsione dei salti Anche detto tabella di previsione dei salti. Piccola memoria indicizzata dalla porzione meno significativa dell'indirizzo dell'istruzione di salto, che contiene uno o più bit che indicano se recentemente il salto è stato eseguito o meno.

buffer di riordino Registro che in un calcolatore con pipeline dinamica mantiene le istruzioni il cui risultato non è ancora stato scritto nei registri o nelle locazioni di memoria visibili al programmatore; i calcolatori con esecuzione fuori sequenza e consegna sequenziale prelevano un'istruzione dal buffer di riordino solo quando la sua esecuzione è terminata e tutte le istruzioni precedenti sono state completate.

buffer di scrittura Coda che memorizza i dati in attesa che siano scritti in memoria.

buffer replicato Anche detto registro replicato. Registro aggiuntivo interno al processore usato per mantenere i risultati in attesa che l'unità di consegna li memorizzi in uno dei registri veri e propri.

bus Insieme di linee di dato trattato dal punto di vista logico come un unico segnale; anche insieme di linee condivise con più sorgenti e usi diversi.

bus asincrono Bus che usa un protocollo di handshaking invece di un segnale di clock per coordinare gli accessi; è adatto ad un'ampia varietà di dispositivi con differenti velocità.

bus di backplane Bus progettato in modo da far coesistere su un unico bus processori, memoria e dispositivi di I/O.

bus master Unità collegata al bus che può richiederne l'accesso.

bus processore-memoria Bus che connette il processore alla memoria e che è di dimensioni ridotte, generalmente ad alta velocità ed adattato al sistema di memoria in modo da massimizzare l'ampiezza della banda memoria-processore.

bus sincrono Bus che comprende un segnale di clock fra le linee di controllo e prevede un protocollo per la comunicazione che fa riferimento al clock.

cache a corrispondenza diretta Struttura della cache in cui a ciascuna locazione di memoria corrisponde una sola locazione della cache.

cache a più livelli Gerarchia delle memorie costituita da più livelli di cache invece che dalla memoria principale e da una sola cache.

cache completamente associativa Struttura della cache in cui un blocco può essere posto in qualunque locazione della cache.

cache divisa Schema in cui un livello della gerarchia di memoria è composto da due cache indipendenti che operano in parallelo, una occupandosi delle istruzioni e l'altra gestendo i dati.

cache indirizzata fisicamente Cache indirizzata tramite un indirizzo fisico.

cache indirizzata virtualmente Cache alla quale si accede con un indirizzo virtuale invece che con un indirizzo fisico.

cache non bloccante Cache che consente al processore di fare riferimenti alla cache anche mentre la cache stessa sta gestendo una precedente situazione di miss.

cache set-associative Cache che ha un numero fisso di collocazioni possibili (almeno due) in cui può essere inserito ciascun blocco.

calcolatore a programma memorizzato Calcolatore le cui istruzioni sono rappresentate come numeri, facendo sì che la stessa memoria possa contenere sia i dati che le istruzioni, e consentendo quindi ai programmi di produrre programmi.

calcolatore elettronico general-purpose Calcolatore che non è stato costruito per una funzione specifica.

calcolatore load-store Anche detto calcolatore registro-registro. Architettura del set di istruzioni in cui tutte le operazioni sono fatte fra registri, ed in cui l'accesso ai dati in memoria può avvenire solo tramite le operazioni di load e di store.

calcolatore registro-registro *Vedere* calcolatore load-store.

cambio di contesto Cambiamento dello stato interno del processore che consente a processi diversi di usare il processore; comprende il salvataggio dello stato, necessario per ritornare al processo correntemente in esecuzione.

caricamento ritardato Struttura del software che richiede che le istruzioni di load siano seguite da un'istruzione indipendente dal dato caricato.

carico di lavoro Insieme di programmi in esecuzione su un calcolatore che può essere sia un gruppo di applicazioni vere e proprie eseguite dall'utente che un insieme di programmi che cercano di simularle. Un carico di lavoro tipico specifica sia i programmi che le relative frequenze di utilizzo.

chiamante Programma che invoca una procedura fornendo i parametri necessari.

chiamata di sistema Istruzione particolare che trasferisce il controllo dal modo utente ad una locazione dedicata all'interno dello spazio del codice supervisore, invocando il meccanismo delle eccezioni.

chiamato Procedura che esegue una sequenza di istruzioni basate su parametri forniti dal chiamante e che al termine restituisce il controllo al chiamante.

chip *Vedere* circuito integrato.

cicli di clock per istruzione (CPI) Numero medio dei cicli di clock necessari ad eseguire un'istruzione all'interno di un programma o di una porzione di programma.

ciclo *Vedere* ciclo di clock.

ciclo di clock Anche detto colpo di clock, periodo di clock, clock, ciclo, tick. Il tempo impiegato da un periodo del segnale di clock, solitamente relativo al clock del processore, che opera a frequenza costante. Il ciclo di clock è spesso usato per misurare la velocità con la quale l'hardware esegue le funzioni base.

cifra binaria Anche detta bit. Una delle due cifre in base due, 0 o 1, che rappresentano le componenti base dell'informazione.

circuito integrato Anche detto chip. Dispositivo costituito da un numero di transistor variabile tra le decine ed i milioni.

circuito integrato a larghissima scala di integrazione (VLSI) Dispositivo che contiene da decine di migliaia a milioni di transistor.

circuito VLSI *Vedere* circuito integrato a larghissima scala di integrazione (VLSI).

clock *Vedere* ciclo di clock.

cluster Insieme di calcolatori connessi in rete locale (LAN) che opera come un unico sistema multiprocessore di grandi dimensioni.

codice operativo Campo che determina l'operazione ed il formato di un'istruzione.

coerenza della memoria principale Coerenza dei valori dei dati in memoria nei sistemi multiprocessore connessi via rete.

coerenza delle cache Coerenza del valore dei dati all'interno delle cache appartenenti a processori diversi.

coerenza delle cache tramite snooping Metodo per mantenere la coerenza delle cache, in cui tutti i controllori delle cache osservano il bus per determinare se hanno o meno una copia del blocco richiesto.

colpo di clock *Vedere* ciclo di clock.

combinazione di istruzioni Misura della frequenza delle istruzioni all'interno di uno o più programmi.

compilatore Programma che traduce istruzioni scritte in linguaggio ad alto livello in codice assembler.

compilazione separata Suddivisione di un programma in più file, ciascuno dei quali può essere compilato separatamente senza bisogno di conoscere il contenuto degli altri file.

consegna fuori sequenza Situazione in cui il risultato di un'esecuzione tramite pipeline non ha il vincolo di essere scritto nello stato visibile al programmatore nello stesso ordine in cui è stato eseguito il prelevamento delle istruzioni.

consegna sequenziale Consegna in cui il risultato dell'esecuzione tramite pipeline viene scritto nello stato visibile all'utente seguendo lo stesso ordine in cui sono prelevate le istruzioni.

contatore di stato futuro Contatore che fornisce lo stato futuro di un circuito sequenziale.

convenzione di chiamata a procedura *Vedere* convenzione sull'uso dei registri.

convenzione sull'uso dei registri Anche detta convenzione di chiamata a procedura. Procedura software che governa l'utilizzo dei registri da parte delle procedure.

corrispondenza degli indirizzi *Vedere* traduzione degli indirizzi.

CPI *Vedere* cicli di clock per istruzione (CPI).

CPU *Vedere* unità centrale di elaborazione (CPU).

criticità dovuta ai salti condizionati Anche detta criticità sul controllo. Evento in cui un'istruzione non può essere eseguita nel ciclo di clock appropriato perché l'istruzione prelevata non è quella giusta; in altre parole il flusso degli indirizzi delle istruzioni non è quello previsto dalla pipeline.

criticità strutturale Evento che si verifica quando un'istruzione pianificata non può essere eseguita nell'opportuno ciclo di clock perché l'hardware non è in grado di supportare la combinazione di istruzioni messe in esecuzione nel ciclo di clock in questione.

criticità sui dati Anche detta criticità sui dati nella pipeline. Situazione in cui un'istruzione pianificata non può essere eseguita nel

ciclo di clock opportuno perché un dato necessario all'esecuzione dell'istruzione non è ancora disponibile.

criticità sui dati di tipo carica-e-usa Caso specifico di criticità sui dati in cui il dato caricato dall'istruzione load non è ancora disponibile nel momento in cui viene richiesto.

criticità sui dati nelle pipeline *Vedere* criticità sui dati.

criticità sul controllo *Vedere* criticità dovuta ai salti condizionati.

dati statici Porzione della memoria che contiene i dati la cui dimensione è nota al compilatore e la cui durata di vita è pari a quella dell'intera esecuzione del programma.

decodificatore Blocco logico con un ingresso di n bit e 2^n uscite, in cui soltanto una delle uscite è affermata per qualunque combinazione degli ingressi.

die Porzione rettangolare tagliata a partire da un wafer, chiamata informalmente chip.

difetto Imperfezione microscopica in un wafer o nei passi di creazione del pattern, che può provocare il malfunzionamento del die che contiene il difetto.

dipendenze sui dati Necessità di un dato specifico in un certo punto della pipeline.

directory Archivio che mantiene le informazioni sullo stato di ciascun blocco della memoria principale, compreso quali cache hanno copie del blocco, se è sporco, e così via.

direttiva assembler Operazione che dà indicazioni all'assemblatore su come tradurre un programma, ma che non produce istruzioni macchina; inizia sempre con un punto.

disco magnetico Memoria secondaria non volatile, costituita da piatti rotanti ricoperti da materiale magnetico in grado di registrare informazioni.

dispositivo di ingresso Meccanismo attraverso il quale il calcolatore riceve informazioni (ad esempio la tastiera o il mouse).

dispositivo di uscita Meccanismo che invia all'utente il risultato di un'elaborazione.

divisore Numero per il quale viene diviso il dividendo; quando viene moltiplicato per il quoziente e viene sommato il resto fornisce come risultato il dividendo.

DMA *Vedere* accesso diretto alla memoria (DMA).

doppia precisione Valore in virgola mobile rappresentato su due parole di 32 bit.

DRAM *Vedere* memoria dinamica ad accesso casuale (DRAM).

DSM *Vedere* memoria condivisa distribuita (DSM).

eccezione Anche detta interruzione. Evento non previsto che interrompe l'esecuzione di un programma; usata ad esempio per rilevare le condizioni di overflow.

eccezione imprecisa *Vedere* interruzione imprecisa.

eccezione precisa *Vedere* interruzione precisa.

elaborazione delle transazioni Tipo di applicazione in cui è richiesta la gestione di piccole e brevi operazioni (dette transazioni) che in generale necessitano sia di I/O che di calcoli. Le applicazioni di elaborazione delle transazioni in generale hanno requisiti sul tempo di risposta e misure di prestazioni basate sul throughput delle transazioni.

elemento dell'unità di elaborazione Unità funzionale usata per elaborare o memorizzare dati all'interno di un processore. Nell'architettura MIPS gli elementi dell'unità di elaborazione comprendono le memorie per le istruzioni e per i dati, il register file, l'unità aritmetico logica (ALU) e i sommatori.

elemento di stato Elemento di memoria.

errore di sincronizzazione Situazione in cui un flip-flop entra in uno stato metastabile, ed in cui alcuni blocchi logici ne interpretano l'uscita come 0 ed altri come 1.

esadecimale Numero in base 16.

esecuzione fuori sequenza Situazione che si verifica quando l'interruzione dell'esecuzione di un'istruzione in una pipeline non costringe l'istruzione seguente ad aspettare.

Esecuzione multipla di istruzioni Procedura in cui l'unità di prelevamento delle istruzioni può inviare più istruzioni allo stadio successivo della pipeline in un singolo ciclo di clock.

esecuzione sequenziale Esecuzione convenzionale tramite pipeline, in cui quando l'esecuzione di un'istruzione viene interrotta tutte le istruzioni successive devono aspettare.

esecuzione speculativa Tecnica di progettazione delle pipeline che unisce la schedulazione dinamica alla previsione dei salti.

espansione dei cicli Tecnica per ottenere migliori prestazioni dai cicli che accedono a vettori, in cui vengono fatte più copie del corpo del ciclo ed istruzioni appartenenti ad iterazioni diverse sono schedulate insieme.

esponente Nel sistema di rappresentazione dei numeri usato nell'aritmetica in virgola mobile è il valore inserito nel campo esponente.

estensione del segno Incremento delle dimensioni di un elemento di dato, ottenuto replicando il valore del bit di segno del dato originale in tutti i bit più significativi del dato (più ampio) da ottenere come risultato.

ethernet Rete di calcolatori la cui lunghezza è limitata a circa un chilometro. Originariamente poteva trasferire fino a 10 milioni di bit al secondo, mentre le nuove versioni possono arrivare fino a 100 milioni di bit al secondo o persino a 1.000 milioni di bit al secondo. Considera le connessioni come bus con più master ed utilizza la rilevazione delle collisioni e uno schema di ritrasmissione per gestire gli accessi simultanei.

etichetta esterna Anche detta etichetta globale. Etichetta che si riferisce ad un oggetto a cui è possibile fare riferimento da altri file oltre a quello in cui viene definita.

etichetta globale *Vedere* etichetta esterna.

etichetta locale Etichetta che si riferisce ad un oggetto che può essere usato solo all'interno del file in cui è definito.

fairness Proprietà di uno schema di allocazione, ad esempio di un protocollo per l'arbitraggio di un bus, che assicura che a nessun dispositivo, neanche a quello con priorità più bassa, viene negato l'accesso alla risorsa.

falsa condivisione Situazione di condivisione in cui due variabili condivise non correlate fra loro sono collocate nello stesso blocco della cache, e l'intero blocco è scambiato fra i processori anche se questi accedono a variabili diverse.

file eseguibile Programma in grado di essere eseguito, che ha il formato di un file oggetto ma che non contiene riferimenti non risolti, informazioni di rilocazione, tabella dei simboli o informazioni per il debugger.

firmware Microcodice implementato in una struttura di memoria, solitamente una ROM o una RAM.

flip-flop Elemento di memoria in cui l'uscita è uguale al valore dello stato memorizzato all'interno dell'elemento stesso, ed in cui lo stato interno cambia soltanto in corrispondenza di un fronte del segnale di clock.

flip-flop di tipo D Flip-flop con un ingresso di dato, che memorizza il valore del segnale di ingresso in una memoria interna quando si verifica un fronte del clock.

floppy disk Memoria secondaria portatile composta da un piatto rotante di mylar ricoperto da materiale magnetico in grado di registrare informazioni.

flusso di istruzioni multiplo, flusso di dati multiplo (MIMD) Classe di elaboratori della tassonomia di Flynn che si riferisce a calcolatori che usano più flussi di istruzioni e più flussi di dati.

flusso di istruzioni singolo, flusso di dati multiplo (SIMD) Classe di elaboratori della tassonomia di Flynn che si riferisce a calcolatori con un solo flusso di istruzioni ma più flussi di dati, ed in cui una sola istruzione opera su molti elementi di dato allo stesso tempo.

flusso di istruzioni singolo, flusso di dati singolo (SISD) Classe di elaboratori della tassonomia di Flynn che si riferisce a calcolatori con un solo flusso di istruzioni ed un solo flusso di dati; la classe SISD rappresenta i processori convenzionali di cui si è trattato nei primi otto capitoli.

formato delle istruzioni Formato di rappresentazione di un'istruzione composto da campi di numeri binari.

frame della procedura Anche detto record di attivazione. Il segmento dello stack che contiene i registri salvati da una procedura e le variabili locali.

frame di chiamata a procedura Blocco di memoria usato per mantenere i valori passati come argomenti ad una procedura, per conservare i registri che una procedura potrebbe modificare ma che il programma chiamante vuole mantenere inalterati, e per memorizzare le variabili locali di una procedura.

frequenza di clock La velocità del processore o del clock di sistema misurata come numero di cicli di clock al secondo, solitamente riportata in megahertz o milioni di cicli di clock al secondo. La frequenza di clock è l'inverso del periodo di clock. I progettisti fanno riferimento al tempo del ciclo di clock sia come durata di un periodo di clock, misurata come secondi per ciclo di clock (ad esempio 2 ns), sia come frequenza di clock, misurata in cicli di clock per secondo (ad esempio 500 MHz).

frequenza di hit Frazione degli accessi in memoria in corrispondenza dei quali il dato si trova nella cache.

frequenza di miss Frazione degli accessi in memoria in corrispondenza dei quali il dato non si trova in un determinato livello della gerarchia di memoria.

frequenza di miss globale Frazione degli accessi che danno luogo ad una situazione di miss in tutti i livelli di una cache multilivello.

frequenza di miss locale Frazione di riferimenti ad un livello della cache che danno luogo ad una situazione di miss; usata nelle gerarchie multi-livello.

funzione di stato futuro Funzione combinatoria che, dati gli ingressi e lo stato corrente, determina lo stato futuro di una macchina a stati finiti.

generazione di calcolatori Classificazione dei calcolatori generalmente basata sulla tecnologia di implementazione usata in ciascuna generazione, che in origine durava otto o dieci anni.

gerarchia delle memorie Struttura che utilizza più livelli di memoria; via via che cresce la distanza dalla CPU aumenta sia la dimensione della memoria che il suo tempo d'accesso.

gestore delle interruzioni Frammento di codice eseguito quando si verifica un'eccezione o un'interruzione.

gigabyte Tradizionalmente 1073741824 (2^{30}) byte, anche se alcuni sistemi di comunicazione o di memorizzazione secondaria lo hanno ridefinito come 1000000000 (10^9) byte.

GPR *Vedere* registro general-purpose.

guardia Il primo dei due bit aggiuntivi mantenuti a destra durante i calcoli intermedi che

riguardano i numeri in virgola mobile; usato per migliorare la precisione degli arrotondamenti.

hard disk Memoria secondaria composta da un piatto metallico rotante ricoperto da materiale magnetico in grado di registrare informazioni.

I/O basato su interruzioni Schema di I/O che utilizza le interruzioni per segnalare al processore che un dispositivo di I/O necessita di attenzione.

I/O mappato in memoria Schema di I/O in cui alcune parti dello spazio di indirizzamento sono assegnate a dispositivi di I/O, e le operazioni di lettura o scrittura a quegli indirizzi sono interpretati come comandi rivolti al corrispondente dispositivo di I/O.

implementazione Hardware che rispetta le specifiche imposte dall'astrazione dell'architettura.

implementazione a molti cicli di clock *Vedere* implementazione multi-ciclo.

implementazione a singolo ciclo Anche detta implementazione a singolo ciclo di clock. Implementazione in cui un'istruzione è eseguita in un solo ciclo di clock.

implementazione a singolo ciclo di clock *Vedere* implementazione a singolo ciclo.

implementazione multi-ciclo Anche detta implementazione a molti cicli di clock. Implementazione in cui un'istruzione è eseguita in più cicli di clock.

indirizzamento con spiazzamento *Vedere* indirizzamento di tipo base.

indirizzamento di tipo base Anche detto indirizzamento con spiazzamento. Modo di indirizzamento in cui l'operando si trova nella locazione di memoria il cui indirizzo è ottenuto dalla somma del contenuto di un registro e di un indirizzo specificato nell'istruzione.

indirizzamento di tipo registro Modo di indirizzamento in cui l'operando è un registro.

indirizzamento immediato Modo di indirizzamento in cui l'operando è una costante all'interno della stessa istruzione.

indirizzamento PC-relative Modo di indirizzamento in cui l'indirizzo si ottiene dalla somma del program counter (PC) e della costante contenuta nell'istruzione.

indirizzo Valore usato per indicare la locazione di un determinato elemento di dato all'interno di un vettore di memoria.

indirizzo assoluto Il reale indirizzo di memoria di una variabile o di una procedura.

indirizzo di destinazione del salto L'indirizzo specificato nell'istruzione di salto condizionato, che nel caso sia eseguito diventa il nuovo valore del program counter (PC). Nell'architettura MIPS l'indirizzo di destinazione del salto è dato dalla somma del campo offset dell'istruzione e dell'indirizzo dell'istruzione successiva al salto.

indirizzo di ritorno Collegamento che consente ad una procedura di restituire il controllo all'indirizzo opportuno del programma chiamante; nel MIPS è mantenuto nel registro `$ra`.

indirizzo fisico Indirizzo della memoria principale.

indirizzo virtuale Indirizzo che corrisponde ad una locazione dello spazio virtuale, e che quando avviene l'accesso in memoria è tradotto in un indirizzo fisico.

informazioni di rilocazione Segmento di un file oggetto Unix che identifica le istruzioni e le parole di dato che dipendono da indirizzi assoluti.

ingresso di stato Ingresso del circuito logico combinatorio che specifica lo stato corrente.

interruzione Eccezione che ha origine all'esterno del processore. Alcune architetture usano il termine *interruzione* per indicare tutti i tipi di eccezioni.

interruzione imprecisa Anche detta eccezione imprecisa. Interruzione o eccezione in un calcolatore dotato di pipeline non associata all'esatta istruzione che è stata la causa dell'interruzione o dell'eccezione.

interruzione precisa Anche detta eccezione precisa. Nei calcolatori con pipeline, interruzione o eccezione sempre associata all'istruzione corretta.

interruzione vettorizzata Interruzione per la quale l'indirizzo a cui è trasferito il controllo è determinato dalla causa dell'eccezione.

invalidazione in scrittura Protocollo di snooping secondo cui il processore che effettua l'operazione di scrittura invalida tutte le copie contenute nelle altre cache prima di modificare la propria copia locale; questo consente di aggiornare la copia locale finché un altro processore non ne fa richiesta.

istruzione di I/O Istruzione dedicata usata per inviare un comando ad un dispositivo di I/O, e che specifica sia il numero del dispositivo che la parola di comando (o la locazione della parola di memoria in cui si trova il comando).

istruzione di jump-and-link Istruzione che esegue un salto ad un dato indirizzo e che contemporaneamente salva l'indirizzo dell'istruzione successiva in un registro (`$ra` nel caso del MIPS).

istruzione di trasferimento dati Comando che sposta dei dati fra memoria e registri.

istruzione rieseguibile Istruzione che riprende l'esecuzione dopo che è stata risolta un'eccezione senza che quest'ultima abbia influenzato il risultato dell'istruzione.

kernel benchmark Frammento di codice di piccole dimensioni ma con tempo di esecuzione non trascurabile, estratto da un programma reale ed usato per valutare le prestazioni.

kilobyte 1024 (2^{10}) byte.

LAN *Vedere* rete locale (LAN).

larghezza di banda della rete Intuitivamente è la velocità di trasferimento di picco di una rete.

latch Elemento di memoria in cui l'uscita è uguale al valore dello stato memorizzato al suo interno, ed il cui stato cambia quando cambia il relativo ingresso ed il segnale di clock è affermato.

latenza della pipeline Numero di stadi di una pipeline o numero di stadi fra due istruzioni durante l'esecuzione.

latenza delle istruzioni Tempo di esecuzione intrinseco di un'istruzione.

latenza di rotazione Anche detta ritardo. Tempo richiesto perché il settore desiderato del disco si venga a trovare al di sotto della testina di lettura/scrittura; di solito si assume che sia la metà del tempo di rotazione.

legge di Amdahl Regola che dice che l'aumento massimo delle prestazioni in risposta ad un determinato miglioramento è determinato da quanto viene usata la funzionalità migliorata.

libreria di sottoprogrammi Collezione di programmi usati frequentemente.

lingotto di silicio cristallino Lingotto composto da cristalli di silicio di diametro variabile da 15 a 30 cm e di lunghezza compresa fra i 30 e i 60 cm.

linguaggio di programmazione ad alto livello Linguaggio portabile come C, Fortran o Java, composto da parole chiave inglesi e da notazioni algebriche, che può essere tradotto in linguaggio assembler da un compilatore.

linguaggio macchina Rappresentazione binaria usata per la comunicazione all'interno dei sistemi di elaborazione.

linguaggio sorgente Linguaggio ad alto livello in cui il programma è originariamente scritto.

link editor *Vedere* linker.

linker Anche detto link editor. Programma di sistema che combina programmi in linguaggio macchina assemblati indipendentemente e risolve tutte le etichette non definite, producendo un programma eseguibile.

loader Programma di sistema che copia un programma oggetto nella memoria principale in modo che sia pronto per essere eseguito.

località spaziale Principio di località il base al quale se si fa un riferimento ad una locazione di dato, le locazioni vicine hanno elevata probabilità di essere richieste in tempi brevi.

località temporale Principio di località in base al quale se si fa un riferimento ad una locazione di dato c'è un'elevata probabilità che la stessa locazione sia nuovamente richiesta in tempi brevi.

lock Dispositivo di sincronizzazione che consente l'accesso ai dati ad un solo processore alla volta.

logica combinatoria Circuito logico al cui interno non ci sono elementi di memoria, e le cui uscite quindi sono le stesse a parità di ingressi.

logica sequenziale Insieme di elementi logici che contengono della memoria, ed il cui valore quindi dipende non solo dagli ingressi ma anche dal contenuto della memoria.

LRU *Vedere* usato meno recentemente (LRU).

macchina a stati finiti Funzione logica sequenziale che consiste di un insieme di ingressi e di uscite, di una funzione di stato futuro che definisce un nuovo stato a partire dallo stato corrente e dagli ingressi, e di una funzione di uscita che determina un insieme di uscite affermate a partire dallo stato corrente ed eventualmente dagli ingressi.

macchina virtuale Calcolatore virtuale che sembra non avere salti condizionati e operazioni di caricamento ritardati, ed un set di istruzioni più ricco di quello dell'hardware reale.

macro Funzionalità di ricerca e sostituzione che fornisce un semplice meccanismo per assegnare un nome alle sequenze di istruzioni di uso frequente.

macroistruzione Istruzione implementata dall'architettura del set di istruzioni; termine usato per distinguere le istruzioni visibili al programmatore dalle microistruzioni di un'unità di controllo microprogrammata.

mantissa Nella rappresentazione dei numeri in virgola mobile è il valore posto nel campo mantissa.

massicciamente parallelo Calcolatore dotato di almeno 100 processori.

massimamente codificato Uso di forme di controllo codificate che richiedono più livelli di decodifica; il microcodice verticale è massimamente codificato.

matrice logica programmabile (PLA) Elemento logico strutturato costituito da un insieme di ingressi e dai corrispondenti ingressi negati e da due stadi di logica: il primo genera i termini prodotto a partire dagli ingressi e dagli ingressi negati, ed il secondo genera i termini somma a partire dai termini prodotto; le PLA infatti implementano le funzioni logiche come somme di prodotti.

media aritmetica Media dei tempi di esecuzione, che risulta direttamente proporzionale al tempo di esecuzione totale.

media aritmetica pesata Metodo di sintesi delle prestazioni che tiene conto dei tempi di esecuzione di un carico di lavoro pesato, con i pesi assegnati in modo da riflettere l'importanza di un programma in un carico di lavoro reale; calcolato come somma dei prodotti dei pesi per il tempo di esecuzione.

media armonica dei rapporti $HM = \frac{n}{\sum_{i=1}^{n} \frac{1}{\text{rapporto}_i}}$

Formula utile per sintetizzare i tempi di

esecuzione quando i dati sono forniti come rapporti invece che come tempi.

media geometrica $\sqrt[n]{\prod_{i=1}^{n} \text{rapporto dei tempi di esecuzione}_i}$ Formula utile per sintetizzare i tempi di esecuzione normalizzati.

megabyte Tradizionalmente 1.048.576 (2^{20}) byte, anche se alcuni sistemi di comunicazione o di memorizzazione secondaria lo hanno ridefinito come 1.000.000 (10^6) byte.

megaFLOPS *Vedere* milioni di operazioni in virgola mobile al secondo (MFLOPS).

memoria Area di memorizzazione in cui sono mantenuti i programmi in esecuzione e che contiene i dati necessari al programma in esecuzione.

memoria a sola lettura (ROM) Memoria il cui contenuto è definito al momento della creazione, e che in seguito può soltanto essere letta. Le ROM sono usate come logica strutturata per implementare insiemi di funzioni logiche, usando i termini delle funzioni logiche come indirizzi e i bit all'interno di ciascuna parola di memoria come uscite.

memoria ad accesso sequenziale Memoria il cui tempo di accesso differisce a seconda di dove si trova il dato da recuperare, dal momento che i dati sono memorizzati sequenzialmente e quindi per accedere all'ultimo bit è necessario scorrere tutti gli altri; opposta alla memoria ad accesso casuale, in cui il tempo di accesso a qualsiasi bit è lo stesso.

memoria cache Memoria piccola e veloce che funziona da buffer per memorie più grandi e più lente.

memoria condivisa Memoria di un processore parallelo con singolo spazio di indirizzamento, che implica l'uso di comunicazioni esplicite tramite operazioni di load e store.

memoria condivisa distribuita (DSM) Organizzazione della memoria che usa gli indirizzi per accedere ai dati remoti quando vengono richiesti, invece di trasferire i dati nel caso dovessero essere usati.

memoria dinamica ad accesso casuale (DRAM) Memoria che contiene i dati e le istruzioni di un programma in esecuzione, e che consente un accesso più veloce rispetto ad un disco magnetico.

memoria distribuita Memoria fisica suddivisa in moduli disposti vicino ai diversi processori di un sistema multiprocessore.

memoria non volatile Memoria che conserva i dati anche quando viene a mancare la sorgente di alimentazione e che viene usata per conservare i programmi fra un'esecuzione e l'altra. I dischi magnetici sono non volatili, mentre la DRAM è volatile.

memoria primaria Anche chiamata memoria principale. Memoria volatile usata per mantenere i programmi in esecuzione; nei calcolatori attuali è tipicamente costituita da DRAM.

memoria principale *Vedere* memoria primaria.

memoria secondaria Memoria non volatile usata per conservare i programmi ed i dati fra un'esecuzione e l'altra; attualmente è costituita principalmente dai dischi magnetici.

memoria statica ad accesso casuale (SRAM) Memoria in cui i dati sono conservati in modo statico (come nei flip-flop) invece che dinamico (come nelle DRAM). Le SRAM sono più veloci delle DRAM, ma meno dense e più costose a parità di bit.

memoria virtuale Tecnica che utilizza la memoria principale come cache della memoria secondaria.

memoria volatile Sistema di memorizzazione, come le DRAM, che mantiene i dati soltanto se alimentato.

memorizzazione dei programmi L'idea secondo la quale istruzioni e dati di tipo diverso possono essere mantenuti in memoria sotto forma di numeri, che ha condotto alla realizzazione dei calcolatori stored-program.

metastabilità Situazione che si verifica se un segnale viene campionato quando ancora non è stabile in base ai tempi di set-up e di hold richiesti, e che eventualmente può far sì che il valore campionato si trovi nella regione di indeterminatezza fra il valore basso ed il valore alto.

metodologia di temporizzazione Approccio usato per determinare quando un dato è valido e stabile in relazione al segnale di clock.

MFLOPS *Vedere* milioni di operazioni in virgola mobile al secondo (MFLOPS).

microcodice L'insieme delle microistruzioni che controllano un processore.

microcodice orizzontale Uso di microistruzioni contenenti molti campi che controllano in parallelo le varie parti dell'unità di elaborazione e che richiedono operazioni aggiuntive di decodifica minime. L'utilizzo di molti campi rende le microistruzioni più ampie o in altre parole più estese nel senso orizzontale.

microcodice verticale Uso di microistruzioni che contengono molti campi di piccole dimensioni e che richiedono dei blocchi di decodifica aggiuntiva per controllare le unità di elaborazione. L'uso di campi di dimensioni ridotte rende le microistruzioni strette, o in altre parole verticali.

microistruzione Rappresentazione dell'unità di controllo che utilizza istruzioni a basso livello, ciascuna delle quali afferma l'insieme di segnali di controllo attivi in un dato ciclo di clock e nello stesso specifica qual'è la successiva microistruzione da eseguire.

microprogramma Rappresentazione simbolica dell'unità di controllo sotto forma di istruzioni, chiamate microistruzioni, eseguite da un semplice microelaboratore.

milioni di istruzioni al secondo (MIPS) Misura della velocità di esecuzione di un programma basata sul numero di milioni di istruzioni che sono eseguite in un secondo. I MIPS sono calcolati come il numero di istruzioni eseguite dal programma diviso per 10^6 volte il tempo di esecuzione.

milioni di operazioni in virgola mobile al secondo (MFLOPS) Anche detti MFLOPS. Misura della velocità di esecuzione di un programma basata sul numero di milioni di operazioni in virgola mobile che sono eseguite in un secondo. I MFLOPS sono calcolati come il numero di operazioni in virgola mobile eseguite dal programma diviso per 10^6 volte il tempo di esecuzione.

MIMD *Vedere* flusso di istruzioni multiplo, flusso di dati multiplo (MIMD).

minimamente codificato Uso di forme di controllo non codificate che agiscono direttamente sull'unità di elaborazione; il microcodice orizzontale è minimamente codificato.

minimizzazione logica Tecnica per ridurre il numero di porte necessarie ad implementare un insieme di funzioni logiche.

mintermini Anche detti termini prodotto. Insieme di ingressi logici a cui è applicato un operatore di prodotto logico (AND); i termini prodotto costituiscono il primo stadio di una PLA.

MIPS *Vedere* milioni di istruzioni al secondo (MIPS).

miss certo Anche detto miss di partenza. Situazione di miss della cache causata dal primo accesso ad un blocco che non è mai stato nella cache.

miss della cache Richiesta di dati che la cache non può soddisfare perché il dato non è presente nella cache.

miss di partenza *Vedere* miss certo.

miss per capacità Situazione di miss della cache che avviene perché la cache, anche se completamente associativa, non è in grado di contenere tutti i blocchi necessari a soddisfare la richiesta.

miss per collisione *Vedere* miss per conflitto.

miss per conflitto Anche detto miss per collisione. Situazione di miss della cache che avviene nelle cache set-associative o a corrispondenza diretta quando più blocchi competono per lo stesso insieme di locazioni, e che viene eliminata nel caso di cache completamente associative delle stesse dimensioni.

modalità a pagine Meccanismo delle DRAM che consente di accedere a più bit della stessa riga cambiando solo l'indirizzo della colonna, e che quindi è più veloce di un normale ciclo di accesso, che modifica sia l'indirizzo della riga che quello della colonna.

modello delle tre C Modello di cache in cui le situazioni di miss sono classificate in tre categorie: miss certi, miss per capacità, miss per conflitti.

modo di indirizzamento Uno fra i diversi metodi di indirizzamento definiti dall'uso degli operandi e/o degli indirizzi.

modo kernel Anche detto modo supervisore. Modo che indica che il processo in esecuzione è un processo del sistema operativo.

modo supervisore *Vedere* modo kernel.

modulo di memoria single in-line (SIMM) Piccola scheda a circuito stampato che contiene da 4 a 24 circuiti integrati DRAM. Attualmente i calcolatori usano le SIMM per consentire agli utenti di estendere in un secondo tempo la capacità della memoria.

nop Istruzione che non provoca nessun cambiamento di stato.

normalizzato Numero codificato in virgola mobile senza zeri iniziali.

notazione polarizzata Notazione secondo la quale il minimo valore negativo è rappresentato da $00\ldots00_{due}$, ed il massimo valore positivo da $11\ldots11_{due}$, con lo zero che tipicamente ha il valore $10\ldots00_{due}$; il numero è polarizzato in modo tale che il numero più la polarizzazione abbia una rappresentazione non negativa.

notazione scientifica Notazione secondo la quale tutti i numeri hanno una sola cifra a sinistra del punto decimale.

NUMA *Vedere* accesso in memoria non uniforme (NUMA).

operazione AND Operazione che fornisce come risultato il valore 1 se e solo se entrambi gli operandi valgono 1.

operazione atomica Operazione con la quale il processore può sia leggere che scrivere una locazione di memoria durante lo stesso accesso al bus, e che fa sì che altri processori o dispositivi di I/O non possano leggere o scrivere la memoria fino a quando il processore non ha terminato.

overflow (virgola mobile) Situazione in cui un esponente positivo diventa troppo grande per essere contenuto nel campo riservato all'esponente.

page fault Evento che si verifica quando la pagina a cui si desidera accedere non si trova nella memoria principale.

parallelismo dei dati Tipo di parallelismo ottenibile quando si dispone di grandi quantità di dati.

parametro formale Variabile argomento di una procedura o di una macro; sostituita dall'argomento quando la macro viene espansa.

parola Unità naturale di accesso ai dati e alle istruzioni all'interno del calcolatore, di solito costituita da un gruppo di 32 bit; nell'architettura MIPS corrisponde alla dimensione dei registri.

PC *Vedere* program counter (PC).

penalità di miss Tempo richiesto per prelevare un blocco da un livello inferiore della gerarchia delle memorie allo scopo di memorizzarlo nel livello superiore, comprendente il tempo necessario per accedere al blocco, trasmetterlo da un livello all'altro ed inserirlo nel livello che ha segnalato la situazione di miss.

periodo di clock *Vedere* ciclo di clock.

personal computer Calcolatore general-purpose progettato per essere prodotto su larga scala e ad un prezzo sufficientemente conveniente per l'uso casalingo.

pipeline Tecnica di implementazione in cui viene sovrapposta l'esecuzione di più istruzioni, in analogia a quanto avviene in una catena di montaggio.

pipeline superscalare Tecnica che replica le componenti interne del calcolatore per lanciare ed eseguire più istruzioni in ogni stadio della pipeline.

pixel Il più piccolo elemento di un'immagine. Gli schermi contengono un numero di pixel tra le decine di migliaia e i milioni, organizzati in una matrice.

PLA Vedere matrice logica programmabile (PLA).

polling Processo che consiste nel controllare periodicamente lo stato di un dispositivo di I/O per determinare la necessità o meno di fornire un servizio a tale dispositivo.

porta AND Circuito che implementa l'operazione di AND fra i segnali di ingresso e che fornisce in uscita un solo segnale risultato.

porta logica Dispositivo che implementa le funzioni logiche fondamentali, come l'AND o l'OR.

porta NAND Porta AND invertente.

porta NOR Porta OR invertente.

porta OR esclusivo Circuito che svolge l'operazione di OR esclusivo sui segnali di ingresso, e che fornisce un solo segnale di uscita che è risultato dell'operazione di OR esclusivo; è anche l'operazione che produce un risultato uguale a 1 solo se i due bit degli operandi sono diversi.

pre-caricamento Tecnica in cui i blocchi di dati necessari in un secondo tempo sono trasferiti in anticipo nella cache usando istruzioni particolari che specificano l'indirizzo del blocco.

previsione dei salti Metodo per risolvere le criticità dovute ai salti condizionati, che assume che un salto abbia un determinato esito e procede con quella assunzione piuttosto che aspettare fino al momento in cui si conosce l'esito effettivo.

procedura Sottoprogramma memorizzato che svolge un compito specifico basandosi sui parametri che gli vengono forniti.

procedura di invio dei messaggi In sistemi di elaborazione con memorie private, procedura usata da un processore per inviare messaggi ad un altro processore.

procedura di ricezione dei messaggi In sistemi di elaborazione con memorie private, procedura usata da un processore per accettare messaggi da un altro processore.

procedure ricorsive Procedure che chiamano se stesse direttamente o indirettamente attraverso una serie di chiamate.

processore vettoriale Modello di architettura e di compilatore, diffuso fra i supercomputer, in cui esistono operazioni ad alto livello adatte a lavorare su vettori lineari di numeri.

program counter (PC) Registro contenente l'indirizzo dell'istruzione del programma correntemente in esecuzione.

programma oggetto Combinazione di istruzioni in linguaggio macchina, dati ed informazioni necessarie ad allocare opportunamente il programma in memoria.

programma per l'elaborazione parallela Singolo programma in esecuzione contemporaneamente su molti processori.

PROM *Vedere* ROM programmabile (PROM).

propagazione Anche detta scavalcamento. Metodo per risolvere una criticità sui dati leggendo gli elementi di dato mancanti da buffer interni invece di aspettare che si trovino nei registri o nelle locazioni di memoria visibili al programmatore.

protezione Insieme dei meccanismi che assicurano che più processi che condividono il processore, la memoria o i dispositivi di I/O non interferiscano, intenzionalmente o meno, con altri processi leggendone o scrivendone i dati. Questi meccanismi inoltre isolano il sistema operativo dai processi utente.

protocollo a transazione divisa Protocollo che prevede che durante una transazione il bus sia rilasciato mentre il richiedente aspetta la trasmissione di un dato, consentendo al bus di rispondere alla richiesta successiva.

protocollo di handshaking Serie di passi usati per coordinare i trasferimenti asincroni sul bus in cui sia chi invia che chi riceve procede al passo successivo solo se entrambe le parti convengono che il passo corrente è stato completato.

protocollo MESI di coerenza delle cache Protocollo di invalidazione in scrittura il cui nome è l'acronimo dei quattro stati del protocollo: Modificato, Esclusivo, Condiviso, Invalido (in inglese Modified, Exclusive, Shared, Invalid).

pseudoistruzione Variante delle istruzioni in linguaggio assembler frequentemente utilizzata, spesso trattata come una vera e propria istruzione.

puntatore al frame Valore che individua la posizione dei registri salvati e delle variabili locali di una data procedura.

puntatore allo stack Valore che denota l'indirizzo dell'elemento dello stack allocato più di recente, che mostra dove i registri possono essere salvati o dove si trovano i vecchi valori dei registri.

puntatore globale Registro riservato ai dati statici.

quoziente Risultato principale di una divisione; numero che moltiplicato per il divisore e sommato al resto produce il dividendo.

RAID *Vedere* schiere ridondanti di dischi economici (RAID).

record di attivazione *Vedere* frame della procedura.

register file Elemento di stato che consiste di un insieme di registri che possono essere letti o scritti tramite il numero del registro.

registro general-purpose (GPR) Registro che in linea di principio può essere usato per gli indirizzi o per i dati di qualsiasi istruzione.

registro replicato *Vedere* buffer replicato.

registro salvato dal chiamante Registro salvato dal programma che esegue una chiamata a procedura.

registro salvato dal chiamato Registro salvato dal programma chiamato.

resa Percentuale di die funzionanti rispetto al numero totale.

resto Risultato secondario di una divisione; numero che sommato al prodotto del quoziente per il divisore produce il dividendo.

rete completamente connessa Rete che connette nodi processore-memoria mediante un collegamento dedicato fra ciascuna coppia di nodi.

rete crossbar Rete che consente a ciascun nodo di comunicare con qualunque altro in un solo passaggio attraverso la rete.

rete geografica (WAN) Rete che si estende per centinaia di chilometri, arrivando a coprire un continente.

rete locale (LAN) Rete progettata per trasportare dati in un'area geograficamente limitata, tipicamente un singolo edificio.

rete multistadio Rete che ha un piccolo commutatore in corrispondenza di ogni nodo.

ricerca Processo che consiste nel posizionare la testina di lettura/scrittura sull'opportuna traccia del disco.

richiesta di bus Segnale sul bus che ne richiede l'accesso.

riferimento in avanti Etichetta usata prima di essere stata definita.

riferimento non risolto Riferimento che per essere completato richiede ulteriori informazioni contenute in un modulo esterno.

ritardo *Vedere* latenza di rotazione.

ROM *Vedere* memoria a sola lettura (ROM).

ROM programmabile (PROM) Forma di memoria a sola lettura che può essere programmata nel momento in cui il progettista ne conosce il contenuto.

salto condizionato Istruzione che richiede il confronto fra due valori e che consente il trasferimento del controllo ad un nuovo indirizzo all'interno del programma a seconda del risultato del confronto.

salto condizionato eseguito Salto condizionato in cui la condizione è verificata; il program counter (PC) assume il valore della destinazione del salto. I salti incondizionati sono tutti salti eseguiti.

salto condizionato non eseguito Salto condizionato in cui la condizione è falsa; il program counter (PC) assume il valore dell'indirizzo dell'istruzione immediatamente successiva a quella di salto.

salto condizionato ritardato Tipo di salto condizionato in cui l'istruzione immediatamente successiva al salto viene sempre eseguita, indipendentemente dal fatto che la condizione per il salto sia vera o falsa.

scambio di messaggi Comunicazione fra più processori ottenuta per mezzo dell'invio e della ricezione esplicita di informazioni.

scartare (istruzioni) Eliminare istruzioni da una pipeline, solitamente in conseguenza di un evento non previsto.

scavalcamento *Vedere* propagazione.

scheda madre Scheda che contiene i vari circuiti integrati o chip, compreso il processore, la cache, la memoria ed i connettori per i dispositivi di I/O come la rete e i dischi.

schedulazione dinamica della pipeline Modalità di schedulazione che va oltre le situazioni di stallo per trovare istruzioni successive che possano essere eseguite nell'attesa che lo stallo sia risolto.

schermo a raggi catodici (CRT) Schermo analogo a quello televisivo, che scandisce un'immagine riga per riga, da 30 a 75 volte al secondo.

schiere ridondanti di dischi economici (RAID) Organizzazione di dischi che utilizza una schiera di dischi piccoli e poco costosi in modo da incrementare sia le prestazioni che l'affidabilità.

SCSI *Vedere* small computer systems interface (SCSI).

segmentazione Schema di traduzione degli indirizzi a dimensione variabile, in cui un indirizzo è composto da due parti: un numero di segmento, che corrisponde ad un indirizzo fisico, ed un offset all'interno del segmento.

segmento di dato Il segmento di un file oggetto o eseguibile Unix che contiene la rappresentazione binaria dei dati inizializzati usati dal programma.

segmento di stack Porzione di memoria usata per mantenere i frame di chiamata a procedura.

segmento di testo Segmento di un file oggetto Unix che contiene il codice in linguaggio macchina delle procedure del file sorgente.

segnale affermato Segnale (logicamente) vero, o pari a 1.

segnale di controllo Segnale usato per la selezione all'interno di un multiplexer o per

dirigere le operazioni di un'unità funzionale; contrapposto al segnale di dato, che contiene le informazioni su cui operano le unità funzionali.

segnale non affermato o negato Segnale (logicamente) falso, o pari a 0.

semiconduttore Sostanza che non conduce bene l'elettricità.

set di istruzioni Il vocabolario dei comandi comprensibili ad una data architettura.

settore Uno dei segmenti che costituiscono una traccia dei dischi magnetici; il settore è la più piccola quantità di informazione che può essere letta o scritta su disco.

sfasamento del clock Differenza misurata in tempo assoluto fra gli istanti in cui due elementi di stato vedono il fronte del clock.

silicio Sostanza ricavata dalla sabbia, che non conduce bene l'elettricità.

SIMD *Vedere* flusso di istruzioni singolo, flusso di dati multiplo (SIMD).

SIMM *Vedere* modulo di memoria single in-line (SIMM).

sincronizzazione Attività che consiste nel coordinare il comportamento di due o più processi, in esecuzione anche su processori diversi.

sincronizzazione a barriera Schema di sincronizzazione in cui i processori si fermano ad una barriera e non procedono nell'esecuzione finché tutti i processori non l'hanno raggiunta.

singola precisione Valore in virgola mobile rappresentato su una parola di 32 bit.

SISD *Vedere* flusso di istruzioni singolo, flusso di dati singolo (SISD).

sistema multicomputer Processori paralleli con più spazi privati di indirizzamento.

sistema multiprocessore Processore parallelo con un solo spazio di indirizzamento condiviso.

sistema multiprocessore simmetrico (SMP) Anche detto sistema multiprocessore UMA. Sistema multiprocessore in cui gli accessi alla memoria principale richiedono la stessa quantità di tempo indipendentemente dal processore che ne fa richiesta e dalla parola di memoria desiderata.

sistema multiprocessore UMA *Vedere* sistema multiprocessore simmetrico (SMP).

sistema operativo Programma di supervisione che gestisce le risorse di un calcolatore a vantaggio dei programmi eseguiti su tale macchina.

sistema sincrono Sistema di memoria che utilizza dei segnali di clock, ed in cui i segnali di dato sono letti soltanto quando il clock indica che i valori corrispondenti sono stabili.

slave Dispositivo che risponde a richieste di lettura o di scrittura ma che non è in grado di generarle, e che quindi non può essere un bus master.

small computer systems interface (SCSI) Bus usato come standard per i dispositivi di I/O.

smistamento Operazione di un'unità di controllo microprogrammata in cui la microistruzione successiva è scelta in base ad uno o più campi di una macroistruzione; solitamente viene implementata creando una tabella che contiene gli indirizzi delle microistruzioni ed indirizzando tale tabella attraverso un campo all'interno della macroistruzione. Le tabelle di smistamento normalmente sono implementate su ROM o PLA. Il termine *smistamento* è anche usato nei processori con schedulazione dinamica per fare riferimento al processo di invio di un'istruzione ad una coda.

SMP *Vedere* sistema multiprocessore simmetrico (SMP).

software di sistema Software che fornisce servizi di comune utilità, e comprende i sistemi operativi, i compilatori e gli assemblatori.

somma di prodotti Formato di rappresentazione delle funzioni logiche che utilizza la somma logica (OR) di prodotti (termini combinati attraverso l'operatore AND).

speedup Misura delle prestazioni di un elaboratore facente riferimento alle sue prestazioni precedenti ad un miglioramento implementativo. Lo speedup è pari al rapporto fra il tempo di esecuzione prima del miglioramento e quello successivo al miglioramento.

SRAM *Vedere* memoria statica ad accesso casuale (SRAM).

stack Struttura di tipo last-in first-out che permette di salvare i registri.

stack frame *Vedere* frame di chiamata a procedura.

stadio della pipeline Passo di esecuzione di un'istruzione che avviene simultaneamente ad altri passi di istruzioni diverse, e che generalmente impiega un ciclo di clock.

stallo della pipeline Anche detto buco. Stallo introdotto per risolvere una criticità.

stazione di prenotazione Buffer all'interno di un'unità funzionale che mantiene operandi ed operazione.

sticky bit Bit usato per l'arrotondamento oltre al bit di guardia, posto a 1 ogni volta che ci sono dei bit diversi da zero a destra del bit di arrotondamento.

supercomputer Il tipo di calcolatore più veloce e più costoso, usato tipicamente per i calcoli scientifici. I supercomputer generalmente costano dai 2 ai 50 miliardi di lire.

supercomputer vettoriale Supercomputer le cui istruzioni lavorano su vettori di numeri, tipicamente su 64 numeri in virgola mobile contemporaneamente.

superpipeline Tecnica che incrementa la velocità del processore allungando le pipeline.

superscalare Tecnica avanzata di progetto delle pipeline che consente al processore di

eseguire più di un'istruzione per ciclo di clock.

tabella degli indirizzi di salto Anche detta tabella dei salti. Tabella che contiene gli indirizzi di sequenze di istruzioni da eseguire in alternativa.

tabella dei salti *Vedere* tabella degli indirizzi di salto.

tabella dei simboli Tabella che fa corrispondere i nomi delle etichette agli indirizzi delle parole di memoria occupate dalle istruzioni.

tabella delle pagine Tabella contenente le traduzioni da indirizzo virtuale ad indirizzo fisico in un sistema di memoria virtuale. La tabella, conservata in memoria, di solito è indicizzata tramite il numero della pagina virtuale; ciascun elemento della tabella contiene il numero della pagina fisica corrispondente a quello della pagina virtuale se la pagina al momento è in memoria.

tabella di storia dei salti *Vedere* buffer di previsione dei salti.

tag Campo di una tabella delle gerarchie di memoria che contiene le informazioni sull'indirizzo necessarie per identificare se il blocco associato corrisponde alla parola richiesta.

tempo di CPU *Vedere* tempo di esecuzione della CPU.

tempo di CPU relativo al sistema Tempo di CPU speso dal sistema operativo per svolgere un compito su richiesta del programma.

tempo di CPU relativo all'utente Il tempo di CPU speso all'interno del programma vero e proprio.

tempo di esecuzione *Vedere* tempo di risposta.

tempo di esecuzione della CPU Anche detto tempo di CPU. Il tempo effettivo che la CPU impiega a compiere un determinato compito.

tempo di hit Tempo richiesto per accedere ad un livello della gerarchia di memoria, incluso il tempo necessario a determinare se l'accesso conduce ad una situazione di hit o di miss.

tempo di hold Tempo minimo dopo il fronte del segnale di clock in cui l'ingresso deve essere valido.

tempo di propagazione Tempo richiesto perché l'ingresso di un flip-flop si propaghi fino alla sua uscita.

tempo di risposta Anche detto tempo di esecuzione. Tempo totale richiesto ad un calcolatore per completare un compito, compresi gli accessi a disco, gli accessi in memoria, le attività di I/O, l'overhead del sistema operativo, il tempo di esecuzione della CPU e così via.

tempo di set-up Tempo minimo prima del fronte del clock durante il quale il valore di un ingresso deve essere valido.

tempo di trasferimento Tempo richiesto per trasferire un blocco di bit, tipicamente un settore, durante gli accessi a disco.

temporizzazione sensibile ai fronti Metodologia di temporizzazione in cui tutti i cambiamenti di stato avvengono in corrispondenza di un fronte del segnale di clock.

temporizzazione sensibile ai livelli Metodologia di temporizzazione in cui i cambiamenti di stato avvengono in corrispondenza del livello alto o del livello basso del clock, ma non sono istantanei come nel caso della temporizzazione sensibile ai fronti.

terabyte Tradizionalmente 1099511627776 (2^{40}) byte, anche se alcuni sistemi di comunicazione o di memorizzazione secondaria lo hanno ridefinito come 1000000000000 (10^{12}) byte.

termine indifferente (don't-care) Elemento di una funzione logica in cui l'uscita non dipende da tutti i valori degli ingressi. I termini indifferenti possono essere specificati in vari modi.

termini prodotto *Vedere* mintermini.

tick *Vedere* ciclo di clock.

TLB *Vedere* translation-lookaside buffer (TLB).

traccia Uno dei 1000-5000 cerchi concentrici che costituiscono la superficie di un disco magnetico.

traduzione degli indirizzi Anche detta corrispondenza degli indirizzi. Processo con il quale un indirizzo virtuale è convertito in un indirizzo con il quale si può accedere direttamente alla memoria.

transazione sul bus Sequenza di operazioni sul bus che comprendono una richiesta e possono comprendere una risposta, una delle quali può trasportare dati. Una transazione inizia con una singola richiesta e può richiedere molte operazioni sul bus.

transistor Interruttore acceso/spento controllato elettricamente.

translation-lookaside buffer (TLB) Cache che mantiene traccia delle traduzioni degli indirizzi usate più di recente, per evitare gli accessi alle tabelle delle pagine.

ulp *Vedere* unità nell'ultima posizione.

UMA *Vedere* accesso in memoria uniforme (UMA).

underflow (virgola mobile) Situazione in cui un esponente negativo diventa troppo grande per essere contenuto nel campo riservato all'esponente.

unità aritmetico logica (ALU) Circuito che esegue le operazioni aritmetiche e logiche.

unità centrale di elaborazione (CPU) Anche detta processore. La parte attiva del calcolatore, che contiene l'unità di elaborazione e l'unità di controllo e che è in grado di sommare numeri, confrontarli, inviare i segnali di attivazione ai dispositivi di I/O e così via.

unità di consegna *Unità all'interno di una pipeline dinamica o con esecuzione fuori*

sequenza che decide quando è sicuro scrivere il risultato di un'operazione in un registro o in una locazione di memoria visibile al programmatore.

unità di controllo Componente del processore che invia i comandi all'unità di elaborazione, alla memoria ed ai dispositivi di I/O allo scopo di eseguire le istruzioni del programma.

unità di controllo cablata Implementazione di un'unità di controllo mediante una macchina a stati finiti che generalmente utilizza una PLA o un insieme di PLA e logica sparsa.

unità di controllo microprogrammata Metodo di specifica di un'unità di controllo mediante microcodice anziché macchine a stati finiti.

unità di elaborazione Componente del processore che esegue le operazioni aritmetiche.

unità nell'ultima posizione (ulp) Numero di bit errati nella parte meno significativa della mantissa che corrispondono all'errore fra il numero vero e proprio ed il più vicino numero rappresentabile.

usato meno recentemente (LRU) Schema di sostituzione in cui il blocco rimpiazzato è quello inutilizzato da più lungo tempo.

uscita di stato futuro Uscita del circuito combinatorio che specifica il numero dello stato futuro.

uso dei registri *Vedere* convenzione sull'uso dei registri.

valore di controllo *Vedere* valore di selezione.

valore di selezione Anche detto valore di controllo. Segnale di controllo usato per selezionare come uscita di un multiplexer uno dei suoi valori di ingresso.

valvola Componente elettronico, predecessore del transistor, che consiste di un tubo di vetro cavo lungo da 5 a 10 cm dal quale viene rimossa più aria possibile.

virgola mobile Aritmetica del calcolatore che rappresenta numeri in cui la posizione della virgola non è fissa.

wafer Fetta di lingotto di silicio, di non più di 2 mm di spessore, usata per costruire i circuiti integrati.

write-back Schema che gestisce delle operazioni di scrittura aggiornando solo il valore all'interno del blocco della cache, e copiando il blocco modificato nel livello inferiore della gerarchia di memoria soltanto quando questo viene sostituito.

write-through Schema secondo cui le operazioni di scrittura aggiornano sempre sia la cache che la memoria, assicurando la coerenza fra le due.

Indice analitico

Simboli

(commenti), 719
" (doppio apice), 719
_ (sottolineatura), 719

A

accesso alle colonne, 770
accesso alle righe, 770
accumulatore, 162-163
  architetture, 162-163
  esteso, 163
  istruzioni ad, compilazione delle istruzioni di assegnazione in, 162
  memoria, 162
  modo di indirizzamento degli operandi basato su variabili e, 162
  set di istruzioni, 162, 172
algebra booleana, 746
  leggi, 746
  operatori, 746
algoritmi di divisione
  prima versione, 227-228
  seconda versione, 229-230
  terza versione, 230-231
algoritmi di moltiplicazione
  prima versione, 214-216
  seconda versione, 216-217
  terza versione, 217-220
algoritmo di Booth, 220-224
  classificazione dei gruppi di bit, 221
  esempio con moltiplicatore negativo, 221, 222
  forma tabulare, 224
  moltiplicazione in complemento a due, 224
  per numeri positivi, 221
  velocità e, 225
algoritmo di Tomasulo, 460
aliasing, 525
Alto, 617, 618
ALU, 196-212, 293
  a 1 bit, 196-199
  a 4 bit con riporto veloce, 210
  a 32 bit, 199-212
  bit più significativo, 202
  blocchi fondamentali hardware, 197
  CMOS, 213
  codici di funzione, 791
  come elemento dell'unità di elaborazione, 299
  condivisa, 327
  ingressi, 334, 357, 417
  ingresso immediato con segno, 424
  microistruzioni, 351
  multiplexer della, 212
  operazioni in parallelo, 446-448
  per istruzioni di tipo-R, 303
  per istruzioni lw/sw, 303
  realizzazione, 196-213
  riporto veloce e, 204-212
  simbolo, 206
  uscite, 334, 357
  uso nelle istruzioni di riferimento alla memoria, 294
ALU a 1 bit, 197-199
  31 copie di, 204
  con multiplexer esteso, 201, 203
  per AND/OR, 197
  spiegazione, 199, 201
  vedi anche ALU
ALU a 32 bit, 199-212
  a partire da 31 copie di ALU di 1 bit, 204
  adattata al MIPS, 201-204
  finale, 205
  spiegazione, 200, 203
  vedi anche ALU
American Standard Code for Information Interchange
  vedi ASCII
ampiezza di banda
  bisezione, 654
  bus, 577, 584-588
  cache, 486
  effettiva, 492
  I/O, 564, 611
  memoria, 492
  rete, 575
  scrittura, 493
  totale della rete, 654
  vincoli, 605, 606
anelli, 552
Apple
  Apple IIC, 34
  Macintosh II, 276
  Macintosh serie 7200, 607
applicazioni, 6
  come benchmark, 55
  misure di prestazioni e, 44
arbitraggio del bus, 591-592, 639
  centralizzato parallelo, 591
  distribuito mediante auto-selezione, 591
  distribuito mediante rilevamento delle collisioni, 592
  equità (fairness), 590
  in daisy chain, 591, 592
  multiprocessore a singolo bus, 640
  priorità e, 590
  richieste sul bus e, 589
  schemi, 591-592
arbitraggio in daisy chain, 591
  equità (fairness), 590
  spiegazione, 591
  vedi anche arbitraggio del bus
architettura Alpha, 368, 455
  Alpha 21264, 451, 458
architettura 80x86, 150-158
  8086, 151, 166, 275
  8087, 151, 259, 275
  80286, 152
  80386, 152
  80486, 152
  codifica delle istruzioni, 156-157
  dimensione normale dei dati, 154
  double word, 154
  formato delle istruzioni, 157
  istruzioni aritmetiche, 154
  istruzioni intere, 154-156
  istruzioni logiche, 154
  modi di indirizzamento, 152-154
  modi di indirizzamento a 32 bit, 154
  operazioni, 156
  pietre miliari, 151
  potenza, 158
  registri, 152-154
  tipi di istruzioni, 154, 166
  set di istruzioni, 167, 360, 361
  sottoinsieme a 32 bit, 153
  storia, 166-167
  virgola mobile, 259-261
    classi di operazioni, 259
    istruzioni, 260
    operandi, 259
    prestazioni, 261
    tipi degli operandi nella somma, 261
  vedi anche processore Pentium; processore Pentium Pro
architettura del set di istruzioni, 15, 26
  implementazione e, 292
  tenere invariata la, 26
architettura Harvard, 30
architettura RISC, 166, 368, 551, 663
  vedi anche Estensione Web I
architettura VAX, 367
  set di istruzioni, 367
  VAX-11/780, 56, 72
  vedi anche Estensione Web III
architetture, 15
  80x86, 151-158
  Alpha, 455, 545
  a registri general-purpose, 163-164

complesse, sfide nelle, 361
evoluzione rispetto a rivoluzione nelle, 662-664
Harvard, 30
implementazione di, 15
linguaggio di alto livello, 165-166
memoria-memoria, 164
per Power-1, 461
registro-memoria, 163, 164
registro-registro, 163
RISC, 166, 368
senza pipeline, 360
set di istruzioni, 15, 26, 292
System/360, 31
VAX, 164
vedi anche Estensioni Web I e III; MIPS (architettura)
aritmetica
dei calcolatori rispetto a quella del mondo reale, 256
errori e trabocchetti, 261-265
in virgola mobile, 235-261, 265
istruzioni, 161, 721-726
precisa, 255-256
vedi anche somma; divisione; moltiplicazione; sottrazione
aritmetica in virgola mobile in singola precisione, 237
esponente massimo, 242
polarizzazione dell'esponente, 238
rappresentazione binaria, 239
rappresentazione generale, 239
registri, 244
ARPANET, 575
arrotondamento
bit aggiuntivi, 256
caso peggiore, 256
con cifre di guardia, 255-256
modi, 255
precisione, 242
prodotto, 246
sempre verso il basso, 257
sempre verso l'alto, 257
troncamento, 257
verso il numero pari più vicino, 257
ASCI (Accelerated Strategic Computing Initiative), 659-660
ASCII
numeri binari rispetto a, 179
rappresentazione dei caratteri, 120
assegnazione degli stati, 776, 808
assemblatori, 4, 682-688
backpatching, 684
direttive, 678
file oggetto, 134, 136, 677, 682-683
funzione degli, 677-682
informazioni di rilocazione, 685
microassemblatore, 349
microcodice, 809-813
primo passo, 683
sintassi, 719-720
sintesi dei modi di indirizzamento, 718
tabella dei simboli, 685
velocità degli, 684
astrazione, 15
livelli di, 206
principi di, 15
ATM (Asynchronous Transfer Mode), 575

B

backpatching, 684
benchmark, 55
applicazioni come, 55
Dhrystone, 67
I/O, 563-567
I/O del file system, 567
I/O di supercomputer, 565
I/O per elaborazione di transazioni, 566
kernel, 73
piccoli, 56, 57
scelta, 59
SDM (Systems Development Multitasking), 75
SFS (System-level File Server), 75
sintesi dei dati, 57-59
sintetici, 67
SPEC, 56, 57, 74-75
SPEC89, 57, 74
SPEC92, 74, 533, 538
SPEC95, 59-62
descrizione, 60
elenco, 60
SPECfp95, 59, 60, 61, 62
SPECint95, 59, 60, 61
tipi di programmi per, 55
Whetstone, 67
vedi anche prestazioni
benchmark Dhrystone, 67, 82
creazione, 67
distorsioni dovute all'ottimizzazione, 67
vedi anche benchmark
benchmark per l'I/O, 565-567
elaborazione di transazioni, 566
TPC-C, 566
TPC-D, 566
file system, 567
supercomputer, 565-566
throughput e, 565
vedi anche benchmark
benchmark sintetici, 67
Dhrystone, 67
svantaggi, 67
Whetstone, 67
vedi anche benchmark
benchmark SPEC, 56
descrizione, 59
esecuzione, 60
insieme SPEChpc96, 75
per il processore, 56
rapporti, 59
SDM, 75
SFS, 75
SPECbase, 74
SPECfp, 74
SPECint, 74
vedi anche benchmark
benchmark Whetstone, 67
prestazioni, 73
vedi anche benchmark
BINAC, 30
bit, 4 97
combinazione di, 256, 265
di accesso in scrittura, 523
di arrotondamento, 255, 289
di completamento (done), 596
di errore, 596
di guardia, 255, 289
di riferimento, 517
di segno, 181
di uso, 517
di validità, 479
impaccare e disimpaccare i, 192
meno significativo, 179
modificato (dirty), 519
più significativo, 179
quoziente, 230
scalamento dei, 192
Somma, 199
sporco (dirty), 519
sticky, 256, 289
bit map, 9
bit meno significativo, 202
del moltiplicatore, 215
della mantissa, 256
bit più significativo, 202
blocchi a corrispondenza diretta, 499
ritrovamento, 500
sostituzione, 506
vedi anche blocco
blocchi base, 106
blocchi combinatori, 744
blocchi completamente associativi, 499, 509
numero di, 503
ritrovamento, 500
sostituzione, 504
vedi anche blocco
blocchi logici, 747-748
blocchi set-associative
ritrovamento, 499-501
sostituzione, 506
blocco, 474
campo indice, 487
collocazione a corrispondenza diretta, 499, 500
collocazione completamente associativa, 499, 500
collocazione flessibile dei, 499-504

collocazione set-associative, 499
di cache, 482, 487-491
di dimensione fissa, 513
di dimensioni variabili, 513
di quattro parole, 407
di una parola, 488
dimensione del, 489-490, 494-495
indirizzi di, 488
logico, 747, 748
memoria virtuale, 510
numero nella cache, 542
posizione del, 500
ritrovamento, 504, 532-534
scelta per la sostituzione, 504
sostituzione, 534
strategia di collocazione, 501
blocco della cache, 486-491
aumentare la dimensione, 489-490
di più parole, traduzione degli indirizzi per, 488
dimensione, 488-490, 493-494
frequenza di miss e, 489-490
indice, 479
indirizzi, 488
località spaziale e, 487
ricerca, 504
scelta per la sostituzione, 506
vedi anche blocco
bonding, 20
breakpoint, 712
buco, 382
buffer
a tre stati, 766, 767
delle scritture, 451, 485, 496, 542
delle store, 451, 537
di predizione dei salti, 434, 437
di quadro, 11
di rinomina, 453
di riordino, 453
di storia dei salti, 434
di supporto alle traduzioni (TLB), 519-524
di write-back, 542
per registri rinominati, 453
buffer delle scritture, 451, 485, 541
nella DECStation 3100, 485
profondità, 496
stalli, 496
Burks, Arthur, 28, 102, 269
Burroughs
B5000, 166
B5500, 64
burst, 767-769
bus, 296, 577-594
accesso, 589
ampiezza, 593
ampiezza di banda, 578, 584-588
asincroni, 581-586
caratteristiche, 592
condiviso, 330
dati, ampiezza, 586
di backplane, 579-581
di I/O, 579-580
difficoltà nel progetto, 577
linee di controllo, 578, 579
linee di dato, 578, 579
lunghezza, 592
master del, 589-590
multivendor, 618
ottimizzazione, 592
prestazioni, 584-588
processore-memoria, 579-581
caratteristiche, 579
proprietario, 618
protocolli, 592
richieste, 589
sincrono, 581-586
standard, 593-594
IDE, 607
ISA, 607
PCI, 593, 594, 607
SCSI, 593, 594, 607
svantaggi, 578
tipi, 579-581
trasferimento di blocchi, 586
vantaggi, 578
vedi anche arbitraggio del bus
bus asincroni, 581-586
analisi di prestazioni, 584
bus sincroni rispetto a, 584
overhead, 584
posizione dei dispositivi, 581
protocollo di sincronizzazione, 581, 583
scalabilità, 586
segnali di controllo, 583
vedi anche bus
bus condivisi, 330
svantaggi, 330
usi, 330
vedi anche bus
bus di backplane, 579-581
interfaccia, 579-580
spiegazione, 582
standard, 580-581
vantaggi relativi al costo, 580
vedi anche bus
bus di I/O, 579-581
interfaccia, 579
progetto, 579
standard, 580
vedi anche bus
bus PCI
ampiezza di banda di picco, 609
standard, 593, 594, 607
velocità di trasferimento realistiche, 609
vedi anche bus
bus SCSI
controllori, 618
invenzione, 618
standard, 593, 594, 607
vedi anche bus
bus sincroni, 581-586
ampiezza di banda, 588
analisi delle prestazioni, 584
bus asincroni rispetto a, 584
clock, 581
vedi anche bus
bypassing
vedi propagazione
byte
aggiuntivo di codice operativo, 156-157
caricamento di, 120
codice operativo, 156
copia di, 120
di 8 bit, 94
indice di, 253
indirizzi di, 94, 126
meno significativo, 94
scrittura di, 120
vettori di, 121

C

cache, 476-495
a corrispondenza diretta, 477, 479-481, 500, 501, 504
a due livelli, 506
a più livelli, 506-508
accesso alle, 479-482
ampiezza di banda, aumento della, 486
associatività nelle, 502
bit nella, 482
combinate, 486
combinazione di eventi nelle, 524
completamente associative, 499, 501, 509, 532-534
con tag fisici, 525
con tag virtuali, 525
DECStation 3100, esempio della, 483-486
disco magnetico, 573
divise, 486
esempio, prima/dopo i riferimenti, 477
ideali, 497
indicizzata fisicamente, 525
indicizzata virtualmente, 525
indirizzata virtualmente, 525
instradamento dell'I/O attraverso le, 605
miglioramenti evolutivi, 663
non bloccanti, 451, 514
numero di blocchi nelle, 542
principale, 507, 541
ricerca, 478, 479
secondaria, 506-507, 539-540
set-associative, 499, 500, 504, 509
sfide progettuali, 541
simulatore SPIM e, 715
simulazione di, 542
snooping, 648
storia, 551
supporto del sottosistema di memoria, 491-494

svuotare, 604
tag, 478, 487
write-back, 486, 494, 535, 537, 542
write-through, 485, 493, 494, 496, 535, 541
vedi anche gerarchia di memorie
cache a corrispondenza diretta, 477, 499, 500
bit, 482
bit dei tag, 504
contenuto, 479-480
illustrazione delle configurazioni, 501
indirizzi, 481, 504
miss, 502, 504
prestazioni, 503
scelta, 532
spiegazione, 478
vedi anche cache
cache a più livelli, 495, 506-508
cache principale, 507
cache secondaria, 507
interazione tra le cache, 507
prestazioni, 506-507
vedi anche cache
cache completamente associative, 499, 509
bit dei tag, 504
illustrazione delle configurazioni, 501
prestazioni, 503,
scelta, 532-534
sostituzione dei blocchi, 534
vedi anche cache
cache set-associative, 499, 509
a quattro vie, implementazione, 505
contenuto, 503
dimensione dei tag e, 504
illustrazione delle configurazioni, 501
indirizzi, 504
prestazioni, 503
scelta, 532
sostituzione dei blocchi, 534
sostituzione LRU, 506
vedi anche cache
cache write-back, 486, 494
implementazione, 542
in sistemi multiprocessore basati su cache, 638
nella memoria virtuale, 519
scrittura, 537
stalli, 496
vantaggi, 535
vedi anche cache
cache write-through, 485, 493, 494
buffer delle scritture, 542
miss, 536
organizzazione, 496
scrittura, 537
vantaggi, 535
vedi anche cache
calcolatore Atlas, 551
calcolatore con set di istruzioni ridotto
vedi architettura RISC
calcolatore di von Neumann, 28
calcolatore memoria-memoria, 164
istruzione di somma, 173
set di istruzioni, 164, 172
calcolatori
applicazioni, 2
aspetti economici, 2
calcolatrici e, 103
caratteristiche commerciali dei, 35
componenti, 13, 26, 43
comunicare con, 17
con una sola istruzione, 176
connessi in rete, 17
crescita tecnologica, 2
da tavolo, 8, 657
dispositivi di ingresso, 8
dispositivi di uscita, 8
embedded, 680
generazioni di, 35
minicomputer, 32
nelle automobili, 2
personal computer, 32-34
portatili, 2
prestazioni tecnologiche dei, 18
riduzione dei costi, 2
rivoluzione dei, 2-3
supercomputer, 33
sviluppo commerciale dei, 30-34
vettoriali, 667-668
calcolatori Cray, 272
calcolatore vettoriale T90, 277, 451
Cray-1, 33
moltiplicatore, 273
overflow, 273
calcolatori da tavolo (desktop), 8
costo, 657
fotografia, 8
calcolatori vettoriali, 667-668
caratteristiche, 668
supercomputer, 668
vantaggi, 667-668
cambio di contesto, 527
campi dell'istruzione, 98, 131
formato-I, 99, 131
formato-J, 131
formato-R, 131
nomi, 99
offset, 728
vedi anche campi di bit
campi di bit, 195-196
campo formato, 812
capability, 552
caricamento, 690
caricamento ritardato, 461, 708
carico di lavoro, 55, 59
misurazione, 69
tempi di esecuzione relativi, 59
CarryIn, 198, 201, 206
propagazione, 207
specifica, 198
vedi anche sommatore di 1 bit
carry lookahead, 204-212
analogia idraulica, 208, 209
cammino veloce, 212
importanza del, 212
livelli di astrazione, 206
primo livello di astrazione, 206-207
ripple carry rispetto a, 212
secondo livello di astrazione, 207-210
CarryOut, 198, 199, 206
generazione, 207
specifica, 198
vedi anche sommatore di 1 bit
CAS (Column Address Strobe), 770
CDC 6600, 64, 164, 617
fotografia, 460
pipeline, 460
programmi Algol 60, 64, 65
sommatore, 273
unicità, 460
chiamante, 693, 694, 695
chiamata a procedura, 692-701
allocazione dello stack, 117
chiamante, 693-696
chiamato, 693-696
convenzioni, 692-696
cosa viene preservato dopo una, 116
esempi, 696-701
frame, 693
ordine last-in first-out (LIFO), 693
ricorsiva, 696-699
stack e, 114
chiamata di sistema, 716-717
chiamato, 693, 694, 695
chip, 19
configurazione delle SRAM, 765
DRAM, 14, 19
IEEE 754, 275-277
maggior capacità dei, 493
memoria, 493
Pentium, produzione, 21-23
processo di produzione, 20
scheda madre, 11
cicli, 106-108
compilazione con indice di vettore variabile, 106
for, 107, 141, 142-143, 144, 252
nel linguaggio C, 107
predizione dei salti e, 437
temporizzazione, 448
while, 107
cicli di clock, 49
CPU, 54
di stallo in lettura, 495
di stallo in scrittura, 496

di stallo per la memoria, 495, 496
diagrammi delle pipeline a singolo ciclo, 401-403
durata media, 324
equazione, 53
esempio, 50, 51
fissi, 323
in progetti a più cicli, 326
in progetti a singolo ciclo, 321
in sistemi a clock variabile, 324
numero di, 49, 50
per le classi di istruzioni, 337-342
riduzione, 50
stadio di pipeline, 378
tempo di CPU, 495
variabili, 323-324
cicli di clock di stallo per la memoria, 495, 496
dovuti a letture/scritture, 495, 496
numero, 508
cicli di clock per istruzione (CPI), 51, 79, 292
a singolo ciclo, 321
calcolo, 54, 292
complessivi, 53
con un livello di cache, 507
confronto, 54
esempio, 51
implementazione e, 292
nelle CPU a più cicli, 342
numero medio di, 53
ottenere, 53
per le classi di istruzioni MIPS, 283
ciclo di attesa, 641
ciclo for, 107, 141, 142-143, 144
PowerPC, 151
variabili, 252
vedi anche cicli
cifre di guardia, 255-289
arrotondamento con, 255-256
doppia precisione del S/360 e, 271
scalamento, 256
cilindri, 570
circuito integrato, 19-20, 550
costo, 40
transistor, 19
VLSI, 19
classi di istruzioni
cicli di clock per le, 342
condivisione di elementi dell'unità di elaborazione tra le, 303
frequenza delle istruzioni, 342
load e store, 308
MIPS, 161
passi di esecuzione, 330-336
salti condizionati, 308
salti incondizionati, 342
spiegazione, 308
tipo-R, 308
unità di controllo della ALU e, 306
vedi anche istruzioni
clock, 757-760
cluster, 650-653
Berkeley, 650
calcolatori indipendenti, 652
caratteristiche dei, 652
IBM SP2, 650
"ibridi", 653
software, 652
svantaggi, 651
Cobol, 6
Cocke, John, 461
codice di funzione, 99
codice di Hamming, 773
codice Gray, 784
codice macchina, 98, 131
codice operativo (opcode), 98
80x86, 156
mappa per il MIPS, 722
codici di condizione, 155
codici di correzione dell'errore (ECC), 771-773
a distanza 3, 771
a distanza 4, 773
di 1 bit, 771
di Hamming, 773
codifica, 812
codifica delle istruzioni, 721
80x86, 156-157
MIPS, 100, 130
virgola mobile MIPS, 251
codifica minimale, 813
codificatore, 749
coerenza
delle cache, 637-640
mantenere la, 637
sincronizzazione con, 637-644
coerenza delle cache
basata su directory, 669
passi, 644
problema della, 644
scalabilità, 670
coerenza sequenziale, 640
collisioni, 627
collo di bottiglia, 459
Colossus, 29
COMA (Cache Only Memory Architecture), 670
combinazione di istruzioni, 71, 73
commenti, 719
nel linguaggio assembler MIPS, 89
simbolo # e, 719
compilatori, 4, 6, 26, 678, 681
C, 159
codifica in linguaggio assembler rispetto a, 159
creazione di salti ed etichette, 104
gcc, 699
istruzioni generate, 51
linguaggio macchina senza assemblatori e, 682
MIPS, 96, 699
ottimizzazioni, 55
ottimizzazioni specifiche, 56
programmatori rispetto a, 681
risoluzione dei conflitti dovuti a dipendenze, 427
sostituzione dei cicli, 681
trasformazione di programmi C, 133
compilazione
con istruzioni load e store, 95
con un indice di matrice variabile, 95-96
controllo del minore, 108
dell'istruzione switch usando una tabella di indirizzi di salto, 109-110
della procedura di copia di stringhe, 121-122
di assegnazioni complesse in C nel MIPS, 90
di assegnazioni con operandi in memoria, 93
di assegnazioni in C usando registri, 92
di cicli con indice di vettore variabile, 106
di cicli while, 107
di if-then-else in salti condizionati, 105
di istruzioni di assegnazione in istruzioni accumulatore, 162-163
di istruzioni di assegnazione in istruzioni di stack, 165
di istruzioni di assegnazione in istruzioni memoria-memoria, 164
di istruzioni if in salti condizionati, 104
di procedure C in virgola mobile con matrici bidimensionali nel MIPS, 252-254
di procedure che non chiamano altre procedure, 113-114
di programmi C in virgola mobile in codice assembler MIPS, 249-252
complemento, 744
completamento fuori ordine, 451
completamento in ordine, 451
concetto di programma memorizzato, 101-103
invenzione, 101
principi, 160
spiegazione, 101
condizioni di indifferenza (don't care), 755, 756
esempio, 755-756
ingressi, 755

tipi di, 755
uscite, 755
confronti
in virgola mobile, 248
istruzioni, 727-728
operandi immediati e, 123
registri, 335
tra schemi di controllo, 438
contatore di programma (PC), 112
controllo della scrittura, 340-342
destinazione del salto, 352
incremento, 313, 328, 333, 357
leggere in, 333
scrivere da, 333
segnale di scrittura, 330
sequenziale, 349
unità di controllo a singolo ciclo e, 328
contatore di programma per le eccezioni (EPC), 190, 441
copia, 190, 441
registro di salto e, 192
salvare il PC nel, 356
salvare l'indirizzo nel, 356
scrittura nel, 357
contatore di stato futuro, 806
contatore esplicito, 804, 807
controllore di disco, 571
Coonen, Jerome T., 275
coprocessore 0, 702, 717
coprocessore 1, 717
corrispondenza a due livelli, 518
Cosmic Cube, 669
costanti
di 32 bit, caricamento, 124
nel linguaggio assembler MIPS, 113
nelle istruzioni, 122
operandi, 122-124
troppo grandi, 160
costruzione di chip Pentium, 20-23
die, 22, 24
pin, 24
wafer, 21, 22
CPU (unità centrale di elaborazione), 11, 717
a più cicli, CPI nelle, 342-344
a singolo ciclo, 325
cicli di clock, 54
cicli di esecuzione, 495
prestazioni, 48
tempo relativo al sistema, 48
tempo relativo all'utente, 48
Cray, Seymour, 33, 272, 460
critical word first, 491
criticità nella pipeline, 380-388
strutturali, 381, 396
sui dati, 385-388
sul controllo, 381-384, 432-445
criticità sui dati, 385-388, 413-432
condizioni, 416
controllo di MEM, 419
definizione, 420
di tipo carica-e-usa, 386
difficoltà, 420
EX, 415
identificata nello stadio ID, 425
MEM, 417
modifica dell'unità di elaborazione mediante propagazione, 421
propagazione e, 385-424
rilevamento, 415, 416
spiegazione, 416
stadio WB e, 420
stalli e, 425-432
stalli nella pipeline e, 385
vedi anche criticità nella pipeline; pipeline
criticità sul controllo, 381-384, 432-445
decisioni ritardate e, 384
eccezioni, 440-445
frequenza, 433
predizione di non esecuzione dei salti, 383
schemi risolutivi, 433
soluzioni, 383
stalli sui salti condizionati e, 382
vedi anche criticità nelle pipeline; pipeline

D

dati dinamici, 691
dati statici, 691
DEC
8700, 455
Alpha 21264, 451, 458
architettura Alpha, 368, 454, 545
architettura VAX, 368
DECStation 3100, 483-486, 716
dischi, 616
NVAX, 455
PDP-8, 32
VAX-11/780, 56
decisioni ritardate, 384
decodifica, 338
decodificatore, 748
a due livelli, 770
accesso alle colonne, 770
accesso alle righe, 770
da 3 ad 8, 749
DECStation 3100, 483-486, 716
buffer delle scritture, 485
elaborazione delle scritture, 485
frequenze di miss della cache, 486
MIPS R2000, 521
schema write-back, 486
schema write-through, 485
spiegazione della cache, 484
diagrammi delle pipeline, 386, 399-404
diagrammi delle pipeline a più cicli di clock, 399
di due istruzioni, 399
esecuzione delle istruzioni, 414
spiegazione, 399
tradizionali, 399
usi, 399
vedi anche diagrammi delle pipeline
diagrammi delle pipeline a singolo ciclo di clock, 399, 401-404
usi, 399
vedi anche diagrammi delle pipeline
die, 20
buoni, 20
costo, 40
difetti, 20
Pentium Pro, 22, 24
Pentium, 24
resa, 20, 40
spiegazione, 22
vedi anche wafer
dimensione del codice, 64
DIMM (dual inline memory module), 550
dipendenze nelle pipeline, 414
classificazione, 416
latenza e, 456
rilevamento, 416
risoluzione da parte del compilatore, 427
spiegazione, 414
tra registri della pipeline, 417
direttive sulla disposizione dei dati, 687
dischi magnetici, 569-574
cache, 573
capacità, 612
caratteristiche, 573
cilindri, 570
controllori di disco, 571
costo per megabyte, 17, 573, 612
densità, 572
errori e trabocchetti, 609
floppy, 16, 569
hard, 16, 569
latenza di rotazione, 571
memoria principale rispetto a, 17
nella gerarchia delle memorie, 473
offline, 613
online, 613
piatti, 569, 570
RAID, 612
settori, 569, 570
tempi di accesso, 17, 571
tempo del controllore, 571
tempo di lettura, 571-572
tempo di seek, 569, 570, 572
tempo di trasferimento, 571, 612
tracce, 569, 570
variazioni nel diametro, 572-573
vedi anche dispositivi di I/O
dischi ottici compatti (CD), 16
disco RAMAC, 615, 616

disco Winchester, 617
dispositivi di I/O, 7-10, 567-574
bus e, 577-594
comportamento, 567
controllo di basso livello, 595
disco magnetico, 16-17, 569-574
diversità, 567
fornire comandi ai, 596
frequenza dei dati, 567
interfacciare, 594-605
mouse, 8, 568
numero massimo di, 610
registri, 596
rete, 17, 574-577
tastiera, 567
tipi di, 567
tipi di comunicazione, 597
trasferimento dei dati, 593, 600-603
video grafico, 9-11
vista d'insieme, 562
vedi anche I/O; sistemi di I/O
dispositivi rimovibili, 617
dividendo, 226, 232
divisione, 226-234
con numeri decimali, 226
con segno, 232
dividendo, 226, 232
divisore, 226
nel MIPS, 233
non restoring, 289
operandi, 226
prima versione dell'algoritmo, 227-229
quoziente, 226, 229
resto, 226, 232
restoring, 233
seconda versione dell'algoritmo, 229-230
SRT, 254
termini, 226
terza versione dell'algoritmo, 230-231
vedi anche aritmetica
divisore, 226
DMA (accesso diretto alla memoria), 601, 603-605
controllori, 601, 602, 617
dispositivi multipli, 602
in sistemi con memoria virtuale, 603-604
interfacce, 623
interfaccia dell'hard disk, 602
overhead sull'I/O usando, 602
passi del trasferimento, 601
processori, 602
scritture, 603
sottosistema di memoria e, 603-605
trasferimenti concatenati, 604
don't care
vedi condizioni di indifferenza
doppia precisione, 83
aritmetica in virgola mobile, 237, 255
esempio di rappresentazione, 239
intervallo di denormalizzazione, 258
MIPS, 237
polarizzazione dell'esponente, 238
registri, 248
S/360, 272
double word, 154
DRAM (dynamic random access memory), 13-15, 491, 545, 769-771
andamento storico della capacità, 41
aumento di capacità, 26
aumento di dimensione, 494
aumento di velocità, 545
capacità dei chip, 19
chip, 14
costo per megabyte, 17, 473
divario di prestazioni, riduzione del, 547
legge di crescita, 19
miglioramenti tecnologici, 545
opzioni, 493-494
organizzazione a matrice quadrata, 493
package, 549
pass transistor, 769
SDRAM, 493
sincrone (SDRAM), 767, 771
SRAM rispetto a, 473
storia, 549
struttura di decodifica a due livelli, 770
tempo di accesso, 17, 473, 506, 545, 546, 770
transistor per bit, 769
vedi anche memoria

E

eccezione, 190, 355-360, 702-705
cause, 441, 444
chiamata di sistema, 526
controllo, 357-360
criticità sul controllo, 440-445
disabilitazione, 529
elaborazione, 356
gestione, 356-357
gestione nel sistema operativo, 356
implementazione, 359
imprecisa, 444, 461
in calcolatori con pipeline, 441
indirizzi, 356
interruzione di I/O rispetto a, 599
istruzione indefinita, 357
memorizzata nel registro Causa, 444
metodi di comunicazione della causa, 356
multipla, 444
overflow aritmetico, 355, 356, 357
page fault, 528, 529, 703
per chiamata di sistema, 526
precisa, 444
procedure, 192
progetto dell'unità di controllo e, 356
requisiti per il supporto, 356
riabilitazione, 529
rilevazione, 357-359
gestione con macchine a stati finiti, 360
tipi, 357
tipi, 703
TLB, 530
unità di elaborazione con controlli per la gestione, 442
uso dei termini, 355
uso nell'architettura PowerPC, 355
vedi anche interruzione
Eckert, J. Presper, 27, 29, 30, 365, 548
EDSAC, 28-29, 365, 472
EDSAC 2, 366
linee di ritardo in mercurio, 548, 549
spiegazione, 28
EDVAC, 28-29
articolo, 28
completamento, 29
Eispack, 271
elaboratori
vedi calcolatori
elementi combinatori, 293
elementi dell'unità di elaborazione
ALU, 299
condivisione, 303
contatore di programma (PC), 297
operazioni in serie, 333
register file, 298, 299
unità di memoria, 297
elementi di stato, 294
in corrispondenza dei fronti del clock, 758
ingressi, 294
logica combinatoria e, 296
metodologia sensibile ai fronti, 295
uscite, 294, 759
validità, 759
ELXSI 6400, 276
emulazione, 367
ENIAC, 27, 548
brevetti, 29
dimensioni, 27
prestazioni, 70
spiegazione, 28
errori, 23
aumentare la profondità della

pipeline aumenta sempre le prestazioni, 455
i benchmark sintetici predicono le prestazioni, 67
i calcolatori usano un metodo di calcolo antiquato e diventeranno presto obsoleti, 23
istruzioni più potenti implicano migliori prestazioni, 158
la legge di Amdahl non si applica ai calcolatori paralleli, 661-662
la media geometrica dei rapporti dei tempi di esecuzione è proporzionale al tempo totale di esecuzione, 69
la somma in virgola mobile è associativa, 261-262
le memorie magnetiche saranno presto obsolete, 609
le metriche indipendenti dall'hardware predicono le prestazioni, 63
le pipeline possono essere implementate in modo indipendente dalla tecnologia, 454
le pipeline sono facili da progettare e implementare, 454
se c'è spazio nella memoria di controllo, nuove istruzioni sono gratuite, 363-364
solo i matematici teorici si preoccupano della precisione della virgola mobile, 262-265
un bus di 100 MB/s può trasferire 100 MB di dati in 1 secondo, 608
usare il linguaggio assembler per ottenere prestazioni migliori, 158
vedi anche trabocchetti
esadecimale, 134
diffusione, 185
tabella di conversione binaria, 185
esecuzione delle istruzioni, 332-336
fase di accesso alla memoria o completamento dell'istruzione di tipo-R, 335
fase di completamento della lettura dalla memoria, 336
fase di decodifica dell'istruzione e caricamento dei registri, 334
fase di esecuzione, calcolo dell'indirizzo di memoria o completamento dei salti condizionati, 334-335
fase di prelievo dell'istruzione, 333
riepilogo delle fasi, 337
esecuzione in pipeline
con etichette, 407-413
flusso dei dati, 389
spiegazione, 389
esecuzione speculativa, 451
esempio con clock variabile, 323-324
espansione dei cicli, 448
codice, 449
per pipeline superscalari, 449
esponente
davanti alla mantissa, 238
massimo, 241
negativo, 238
nel prodotto, 244
polarizzazione, 238, 242
estensione del segno, 183
estensioni dell'indirizzo a 64 bit, 129
etichette, 676, 679
creazione da parte del compilatore, 104
dichiarazione, 719
esterne, 682
locali, 682
nomi, 684
salti, 106

F

falsa condivisione, 639
fattore di espansione, 681
fattore di peso, 59
fetch, 292
fetch-on-miss, 536
fetch-on-write, 536
file eseguibili, 138, 677, 690
formato, 135
processo di creazione, 677
vedi anche linker
file oggetto, 134, 136, 677, 682
componenti, 134, 682
esecuzione, 682
formato, 685
sezioni, 685
spiegazione, 685
vedi anche assemblatori
file sorgente, 677
file system, benchmark dell'I/O, 567
finestra di attivazione, 117
firmware, 367
flip-flop, 761-763
di tipo D, 761-763
fronte di discesa, 763
latch rispetto a, 761
schiere, 763
sensibili ai fronti, 777
stato metastabile, 780
tempo di hold, 762
tempo di propagazione, 777
tempo di set-up, 762
vedi anche latch
floppy disk, 16
svantaggi, 569
vedi anche dischi magnetici
formati delle istruzioni, 98, 380
80x86, 156
campi, 131
determinazione, 100
tipo-I, 99, 100
tipo-J, 125, 126
tipo-R, 99, 100, 130, 131
formato delle istruzioni di tipo-I, 99, 100
campi, 99, 131
distinzioni, 100
vedi anche formati delle istruzioni
formato delle istruzioni di tipo-J, 125, 126
campi, 131
per le istruzioni j/jal, 125
vedi anche formati delle istruzioni
formato delle istruzioni di tipo-R, 99, 100, 130, 131
ALU e, 306
campi, 131
campo shamt, 193
distinzioni, 100
implementazione delle operazioni della ALU, 299
vedi anche formati delle istruzioni
frame pointer, 117
frequenza dei dati, 566, 567
ampiezza di banda del bus e, 578
tastiera, 608
frequenza delle interruzioni, 600
frequenza di clock, 49, 66, 758
aumentare la, 49, 61
implementazione e, 292
miss nella cache e, 498
Pentium Pro, 362
Pentium, 61
frequenza di hit, 475
frequenza di miss, 475
cache, 486
cache divisa, 480
cause dei miss e, 537-538
combinata, 486
come unica metrica, 543-544
DECStation 3100, 485-486
dimensione dei blocchi e, 488-491
dimensione della cache e, 533
effettiva, 486
maggiore, svantaggi, 486
miglioramento, 533
programma gcc, 489, 503
programma spice, 489, 503
riduzione, 505, 530
tempo di esecuzione e, 486
vedi anche miss
frequenza di rinfresco, 9
funzione di controllo, 345
memorizzazione in una ROM, 353
microprogramma, 354

funzione di sequenzializzazione, 353-354
funzione di stato futuro, 337, 773, 814

G

generazione nel riporto veloce, 207
livelli, 211-212
"super", 211
gerarchia di trasformazione, 131
gerarchie di memorie, 3, 473
blocchi, 474
cache, 476-495
dati, 474
diagramma della struttura, 475
dipendenza dalle, 474
dischi magnetici, 474
elementi, 474
frequenza di hit, 475
frequenza di miss, 475
funzionamento complessivo, 524
gestione, 529
hit, 474
importanza, 548
livelli, 474
livello inferiore, 474
livello superiore, 474
località spaziale e, 472, 475
località temporale e, 472
meccanismi di protezione, 552
memoria virtuale, 509-531
miss, 474
modello delle tre C, 537-539
opzioni per la scrittura, 535
parametri progettuali quantitativi, 531
penalità di miss, 475
Pentium Pro, 539-542
quadro di riferimento comune, 531-539
sfide progettuali, 539, 546
spiegazione della struttura, 474
sviluppo, 551-552
tecnologie, 473
tempo di hit, 475
gestore delle interruzioni, 703
esempio, 704
ritorno dal, 705
salvataggio dei registri, 704
spostamento dei registri, 704
Goldstine, Herman, 28, 102, 269, 271
Goodyear MPP, 666, 667

H

halfword, 122
handshake, 581
hard disk, 16
illustrazione, 17
rimovibili, 615
testina di lettura/scrittura, 16
vantaggi, 569
vedi anche dischi magnetici
hardware
componenti, 12
costo, riduzione del, 812
dispositivi di I/O, 8-11
livelli gerarchici, 7
scheda madre, 11
traduzione del microprogramma in, 353, 809-813
hardware per la moltiplicazione
prima versione, 214
seconda versione, 216
terza versione, 217
hit, 474
HP PA-RISC, 545

I

IBM
305 RAMAC, 615, 616
360/65, 551
360/85, 552
360/91, 460-461
704, 664
7090, 273, 366, 664
7094, 271-272, 273
architettura Power-1, 539
dischi Winchester, 617
Personal Computer, 32, 163
PowerPC, 149-151
Powerstation 550, 57
progetto ACS, 461
RS/6000, 459, 461, 669, 670
SHARE, 271, 272
SP2, 650
Stretch (7030), 70, 459
System/360, 31, 167, 271, 366, 551
System 370, 167, 551
uso del microcodice, 366
identificatore di task, 527
Illiac IV, 666
implementazione
a singolo ciclo, 321-326
architettura del set di istruzioni e, 292
controllore con microcodice, 354
eccezioni, 355
elementare, 303-326
funzione di stato futuro, 802-809
istruzione beq, 300
macchina a stati finiti, 337, 344, 775, 776
microprogramma, 353-354
MIPS, spiegazione, 284
multi-ciclo, 326-345
operazioni ALU di formato-R, 299
passi di un'istruzione, 292
Pentium Pro, 359-362
principi di progetto e, 292
quadro d'insieme, 292-293
salti, 320-321
set di istruzioni 80x86, 360
sottoinsieme MIPS, 296
unità di controllo, 307-321, 336-345
unità di controllo cablata, 367
unità di controllo combinatoria, 788-792
unità di controllo come macchina a stati finiti, 792-802
implementazione multi-ciclo, 326-345
a singolo ciclo rispetto a, 326, 438-440
ciclo di clock, 327
completa dei segnali di controllo, 329
conseguenze sulle prestazioni, 458
definizione dell'unità di controllo, 336-345
esecuzione delle istruzioni, 330-336
registri temporanei, 327-328
supporto ai salti condizionati, 330
supporto ai salti incondizionati, 330
unità di controllo, 330, 331
unità di controllo con pipeline rispetto a, 439
unità di elaborazione, 330, 331, 361, 364
unità di elaborazione del MIPS, 328
unità funzionali, 326
vantaggi, 326
visione ad alto livello, 327
indici
blocco della cache, 481
matrice, 95-96, 106
tabella delle pagine, 513, 514, 515
indici dei vettori, 144
compilazione con, 95-96
compilazione dei cicli, 106
indirizzamento
base, 93, 128, 129, 154
con spiazzamento, 128, 129
di tipo update, 149-150
spiegazione, 150
codice MIPS, 150
immediato, 128,129
modi dell'80x86, 152-154
modi di, 128, 129, 717-719
nei processori paralleli, 647-649
nei salti condizionati, 125-128
nei salti incondizionati, 125-128
pseudo diretto, 128, 129
registro, 128, 129
relativo a PC, 126-127, 128, 129
scaled index, 154
tramite indice, 149
codice MIPS, 149
spiegazione, 149
indirizzi, 128
base, 93

blocco di cache, 488
byte, 94, 125
cache a corrispondenza diretta, 481, 502-503
destinazione del salto, 300, 335, 352
di parole consecutive, 139, 159
eccezione, 356
estensione a 64 bit, 129
fisici, 510
memoria, 93, 94, 334
microistruzione, 347
salto incondizionato, 334
set-associative, 505
troppo grandi, 160
virtuali, 510-512, 514, 530
indirizzi di memoria, 93, 94, 334
calcolo, 337, 339
MIPS, 94
spiegazione, 93, 94
indirizzi fisici, 510, 511
indirizzi virtuali, 514
causa di page fault, 527
di 32 bit, 517
spazio, 514, 526
traduzione, 510, 512
traduzione veloce, 519-521
indirizzo di ritorno, 110-112
ingressi (input), 26
ALU, 334, 357, 417
asincroni, 779
elementi di stato, 294
indifferenti (don't care), 755
logica combinatoria, 759
nell'I/O, 594-605
periferici, 8, 567
unità di controllo, 316, 320
unità di controllo della ALU, 306
Intel
famiglia di microprocessori 80x86, 151-158
iPSC 860, 669
microprocessore 4004, 32, 33, 166
microprocessore 8008, 166
microprocessore 8080, 166
microprocessore 8088, 166
Paragon, 669
sviluppo del set di istruzioni con HP, 551
interleaving, 493, 495
interruzione, 355, 702-705
generazione, 355
I/O, 595, 599
imprecisa, 444, 461
precisa, 444
uso del termine, 355
uso nell'architettura Intel 80x86, 355
uso nell'architettura PowerPC, 355
vettorizzata, 356
vedi anche eccezione
interruzione di I/O
eccezione rispetto a, 599
priorità, 599
inverso, 744
inverter, 197
I/O
ampiezza di banda, 562, 611
controllori, 603, 617
errori e trabocchetti, 608-610
frequenza, 600
inizio sotto il controllo del, 599
instradamento attraverso la cache, 604
istruzioni, 596
lettura, 604
macchina a stati finiti per il controllo del, 584, 585
mappato in memoria, 596
overhead del DMA, 602
pilotato da interruzioni, 599, 600-601
scrittura, 604
tempo, 563
throughput, 578
vedi anche dispositivo di I/O; sistema di I/O
I/O pilotato da interruzioni, 599
overhead, 600-601
processore, 601
vedi anche I/O
IPC (istruzioni per ciclo di clock), 445
istruzione
`abs.d`, 736
`abs.s`, 736
`add`, 100, 101, 106, 107, 116, 129, 189, 292, 378, 721, 723
attraverso la pipeline, 407-413
eccezioni di overflow nella, 442, 443
nello stadio EX, 442, 443
registro destinazione, 440
stalli e, 426
`add.d`, 736
`add.s`, 248, 736
`addi`, 122, 124, 129, 189, 723
`addiu`, 189, 196, 723
`addu`, 189, 723
`and`, 195, 201, 292, 303, 378, 434, 723
attraverso la pipeline, 407-413
stalli e, 426
`andi`, 193, 196, 723
`bcctr`, 363
`bclf`, 248
`bclt`, 248
`bczf`, 729
`bczt`, 729
`beq`, 104, 108, 292, 299, 378, 729
impatto della pipeline su, 433
implementazione, 299
operandi, 299
passi di esecuzione, 316
unità di elaborazione mentre esegue, 319
`bgez`, 729
`bgezal`, 729
`bgtz`, 729
`blez`, 729
`bltz`, 730
`bltzal`, 730
`bne`, 104, 107, 108, 730
campo di indirizzo, 127
vedi anche salti condizionati
`break`, 740
`c.eq.d`, 736
`c.eq.s`, 736
`c.le.d`, 736
`c.le.s`, 737
`c.lt.d`, 737
`c.lt.s`, 737
`cvt.d.s`, 737
`cvt.d.w`, 737
`cvt.s.d`, 737
`cvt.s.w`, 737
`cvt.w.d`, 737
`cvt.w.s`, 738
`c.x.d`, 248
`c.x.s`, 248
`div`, 233, 723
`div.d`, 248, 738
`div.s`, 248, 738
`divu`, 233, 723
`j`, 105, 106, 127, 292, 731
formato, 321
istruzione di salto condizionato rispetto a, 321
unità di controllo ed unità di elaborazione, 322
`jal`, 110, 112, 126, 138, 731
`jalr`, 731
`jr`, 109, 110, 112, 705, 731
`lb`, 120, 122, 181, 732
`lbu`, 181, 732
`lh`, 732
`lhu`, 732
`lui`, 123, 124, 727
effetto, 123
funzione, 123
`lw`, 92, 100, 115, 292, 378, 732
ALU, 305
attraverso la pipeline, 407-413
esempio di compilazione, 94-95
identificazione, 101
`lwcl`, 248, 264
`lwcz`, 732
`lwl`, 732
`lwr`, 732
`mfc`, 190
`mfc0`, 190
`mfcz`, 735
`mfhi`, 225, 734
`mflo`, 225, 735
`mov.d`, 738
`mov.s`, 738
`mtcz`, 735
`mthi`, 735
`mtlo`, 735
`mul.d`, 248, 738
`mul.s`, 248, 738
`mult`, 225, 724
`multu`, 225, 724
`neg.d`, 739
`neg.s`, 739
`nop`, 415, 420, 426, 740
esempio di utilizzo, 415
nello stadio EX, 426
`nor`, 284, 724
`or`, 292, 303, 378, 725
attraverso la pipeline, 407-413
stalli e, 426
`ori`, 193, 196, 725
`rfe`, 739
`sb`, 120, 122, 733
`sh`, 733
`sll`, 192, 725
`sllv`, 725
`slt`, 108, 201, 292, 348, 727
`slti`, 182, 727
`sltiu`, 182, 196, 727
`sltu`, 182, 727
`sra`, 725
`srav`, 725
`srl`, 192, 726
`srlv`, 726
`sub`, 189, 282, 726
`sub.d`, 248, 739
`sub.s`, 248, 739
`subu`, 189, 726
`sw`, 94, 100, 113, 292, 338, 378, 733
ALU per, 305
esempio di compilazione con, 94-95
identificazione, 101
`swcl`, 248
`swcz`, 734
`swl`, 734
`swr`, 734
`syscall`, 526, 740
`xor`, 726
`xori`, 726
istruzione and con operando immediato
vedi istruzione `andi`
istruzione branch condition to count register
vedi istruzione `bcctr`
istruzione branch if equal
vedi istruzione `beq`
istruzione branch if not equal
vedi istruzione `bne`
istruzione case
vedi istruzione switch
istruzione di break, 740
istruzione for, 141
vedi anche ciclo for
istruzione go to, 104, 110
istruzione if, 104
compilazione in salti condizionati, 104
spiegazione delle varianti, 105
istruzione if-then-else, 104-105, 108
istruzione jump-and-link
vedi istruzione `jal`
istruzione jump register
vedi istruzione `jr`
istruzione load byte unsigned
vedi istruzione `lbu`
istruzione load upper intermediate
vedi istruzione `lui`
istruzione load word
vedi istruzione `lw`
istruzione LOOP, 363
istruzione or immediate
vedi istruzione `ori`
istruzione return from exception (RFE), 526, 705, 739
istruzione set on less than
vedi istruzione `slt`
istruzione store word
vedi istruzione `sw`
istruzione switch, 108-110
compilazione mediante tabella di indirizzi di salto, 109-110
salto all'uscita, 110
istruzioni, 4, 87-176
80x86, 155, 164
aritmetiche, 161, 721-726
base, 266, 268
blocchi base, 106
cicli di miss, 497
confronto, 727-728
costanti, 727
costanti nelle, 122
decodifica, 334
di scelta, 102-110
diffusione, 160
eccezioni/interruzioni, 739-740
errori e trabocchetti, 158-160
frequenza, 268
frequenza relativa, 71
I/O, 596
latenza, 456
linguaggio assembler MIPS, 89, 98, 130
load, 92, 299, 732-733
logiche, 284, 721-726
macchina, 98
miss, 482
negli elaboratori, 97
non definite, 356, 357
rilevamento, 357
vedi anche eccezioni
`nop`, 415, 424, 426, 740
numeri, 4
or, 303, 378, 407-413, 725
PowerPC, 150-151
prelievo, 298, 333
pseudo istruzioni, *vedi* pseudo istruzioni
rappresentazione, 88, 97-102
riduzione, 149
rieseguibili, 529
riferimento alla memoria, 292
salto condizionato, 728-731
salto incondizionato, 731
scalamento, 193
sequenza in pipeline, 426
somma, 100, 102, 105, 106, 107, 116, 126, 187, 303, 378, 407-413, 723
sottrazione, 189, 303, 378, 407-413, 726
store, 94, 299, 733-734
stringa, 155
successive, 326
svuotamento, 433, 440
syscall, 526, 740
tipo-R, 298, 312
trasferimento dati, 92, 161, 734-735
vettoriali, 667
virgola mobile, 248-254, 735-739
vedi anche le istruzioni specifiche; esecuzione delle istruzioni; set di istruzioni; microistruzioni
istruzioni che manipolano costanti, 727
istruzioni del nucleo principale, 266, 267, 268
istruzioni di caricamento (load), 92, 732-733
attraversamento delle pipeline, 391-395
bug, 396
con segno, 182
elementi dell'implementazione, 300
elenco, 732-733
funzionamento, 318
indirizzi dei dati, 104
microistruzioni, 350
passi, 316
ritardate, 461, 708
unità di elaborazione, 301
istruzioni di memorizzazione (store), 94, 733-734
attraversamento delle pipeline, 394-397
elementi dell'implementazione, 300
elenco, 733-734
indirizzi dei dati, 104
microistruzioni, 351
unità di elaborazione, 300
istruzioni di accesso alla memoria, 292

macchina a stati finiti per il controllo delle, 338, 340
microprogramma per le, 350
istruzioni di tipo-R
implementate mediante macchina a stati finiti di due stati, 341
passi di esecuzione, 312
passo di completamento, 335
prima fase, 313
quarta fase, 317
seconda fase, 314
sequenza di microprogramma, 351
terza fase, 315
unità di elaborazione, 299
istruzioni di trasferimento dati, 92, 161, 734-735
offset, 93, 96
registro base, 93
istruzioni logiche, 284, 721-726
istruzioni MIPS, 89, 92, 126-127
lunghezza, 380
traduzione in istruzioni macchina, 98
iterazione di Newton, 254

J

Jobs, Steve, 32

K

Kahan, William, 273-275
kernel, 73

L

larghezza di banda
vedi ampiezza di banda
latch, 761-763
aperto, 761
chiuso, 761
D, 761
flip-flop rispetto a, 761
S-R, 760, 761
trasparente, 762
vedi anche flip-flop
latenza, 456, 491
di risposta, 611
di rotazione, 571
vincoli, 605
LCD (display a cristalli liquidi), 10
legge dei rendimenti decrescenti, 63
legge di Amdahl, 62, 63, 216, 456, 497, 609
processori paralleli e, 661
speedup, 85, 634
tempo di esecuzione dopo un miglioramento, 634
tempo di esecuzione non influenzato, 634
tempo di miglioramento, 63
librerie
di programmi, 676
di subroutine, 6
linea di bit, 766
linguaggi di programmazione ad alto livello, 4, 678
architettura degli elaboratori, 165
benefici, 6
C, 88, 90-93, 118
Cobol, 6
Fortran, 6
gerarchia delle traduzioni, 133
Lisp, 6
vantaggio di sinteticità, 6
linguaggio ADA, 189
linguaggio assembler, 4, 132-134, 676
compilatori rispetto a, 159
esempio di procedura, 678
etichette e, 677-679
file oggetto, 134, 677
file sorgente, 677
locazioni di memoria, 677
prestazioni e, 158-159
programmatori, 680
programmi, 133, 682, 684
programmi C compilati in, 5
quando usare, 680-681
ruolo, 677-678
simboli, 676
svantaggi, 6
vantaggi, 677, 681
vedi anche assemblatori; linguaggio assembler MIPS
linguaggio assembler MIPS, 89, 97, 102, 111, 119, 132, 186, 191, 194, 234
codice per un ciclo in C, 159
commenti, 89
compilazione di assegnazioni complesse in C, 90-91
costanti, 113
rappresentazione simbolica, 92
struttura, 89, 97
traduzione del C in, 90
traduzione in linguaggio macchina, 100
versione della procedura di ordinamento, 145
virgola mobile, 250
linguaggio C, 88
cicli, 107
classi di memorizzazione, 118
compilatori, 159
compilazione delle istruzioni di assegnazione in, 90
compilazione delle istruzioni di assegnazione usando registri, 92
compilazione di assegnazioni complesse nel MIPS, 90-91
variabili, 118
vedi anche linguaggi di programmazione ad alto livello
linguaggio destinazione, 680
linguaggio Fortran, 6, 189
linguaggio macchina, 98
decodifica, 130-131
MIPS, 101, 111, 677
offset dei salti, 125-126
rappresentazione simbolica, 125
traduzione in, 100-101
virgola mobile MIPS, 250
linguaggio sorgente, 680
linker, 135-138, 676, 688-690
aggiornamento dei campi di indirizzo, 137
file eseguibile, 135, 137, 690
funzione, 135, 689
passi, 135
riferimenti assoluti e, 135
spiegazione, 689
Linpack, 73, 271
Lisp, 6
località
principio, 472
spaziale, 472-473, 487-491
temporale, 472, 473, 475
tipi, 472
località spaziale, 472-473, 475, 487-491
blocchi della cache e, 487
gerarchia di memorie e, 473, 475
trarre vantaggio dalla, 487-491
vedi anche località
logica, 293-296
combinatoria, 295, 296, 344, 744
componenti sequenziali, 295
elementi di stato, 295
progetto, 295
logica combinatoria, 295, 296, 344, 744, 748-757, 759
condizioni di indifferenza (don't care), 755-757
descrizione delle funzioni, 745
ingressi, 759
long instruction word (LIW), 461
Loop Livermore, 73

M

macchina a stati finiti, 337, 773-776
archi, 338
con il rilevamento delle eccezioni, 360
controllo dell'I/O, 584
esecuzione dell'istruzione di salto condizionato, 341
esecuzione dell'istruzione di salto incondizionato, 342
funzione di stato futuro, 773, 775
funzione di uscita, 773, 775
implementazione, 337, 776
implementazione con registro di stato, 776
implementazione delle istruzioni di tipo-R, 341
ingressi, 776
macchina di Mealy, 345
macchina di Moore, 345, 774
memorizzazione interna, 773, 774
passi di esecuzione e, 337
per controllare le istruzioni di accesso alla memoria, 340
per lo schema di predizione a 2 bit, 438
protocollo di handshaking, 585
segnali di uscita, 774
sincrona, 773
stati, 338, 339
unità di controllo, 792-802
illustrazione, 344, 793
implementazione con PLA, 802
implementazione con ROM, 796-801
visione ad alto livello, 338
vedi anche unità di controllo
macchina di Mealy, 345, 774
macchina di Moore, 345, 774
macchine load-store, 163
esempi, 164
set di istruzioni, 172-173
macchine registro-memoria, 163, 164
macchine registro-registro, 163
macro, 687-688
chiamate, 687
esempio, 687
parametri formali, 687
uso, 688
mantissa, 237
bit meno significativi, 255
esponente davanti alla, 238
moltiplicazione, 245, 246
somma, 241
mappe di Karnaugh, 756-757
Mark-I, 28, 29
Mark-II, 29
Mark-III, 29
Mark-IV, 29
maschera, 193
master del bus, 589-590
multipli, 589-590
segnali di rilascio del bus, 591
singolo, 589
matrici
vedi schiere
massimamente codificato, 813
Mauchly, John, 27, 29, 30, 365
media aritmetica, 59
dei rapporti, 68
dei tempi di esecuzione, 67-68
dei tempi di esecuzione normalizzati, 68
pesata, 59, 68
standard, 59
media armonica, 85
media geometrica
formula, 68
indipendenza dalla serie di dati, 68
svantaggi, 69
tempo totale di esecuzione e, 68
vantaggi, 68
memoria, 11, 26
a nuclei magnetici, 548-549
accesso, 335, 451, 475
ampia una parola, 492
ampiezza di banda, 492
aumento dell'ampiezza, 492
cache, 13, 14, 22
vedi anche cache
centralizzata, 645
chip, 493
completamento della lettura, 336
condivisa, 644-645
contenuto, 93
disco magnetico, 473
distribuita, 645
DRAM, 14-15, 473, 491, 493-494, 545
elementi, 760
elementi non temporizzati, 760
elementi temporizzati, 760
errori e trabocchetti, 542-545
fisica, 510, 516
non volatile, 16, 569
operandi, 93, 381
organizzazione, 691
più veloce, 474
principale, 16, 473, 510
privata, 631
privata multipla, 645
ROM, 344, 347, 366, 753, 754
scheda madre, 11
secondaria, 16
segmento dei dati, 691
segmento di stack, 691
segmento di testo, 691
solo cache, 649
SRAM, 473
tecnologia, 3
tempo medio di accesso (AMAT), 555
trasferimento di dati dai dispositivi di I/O e, 600-603
unità, 323, 326
uscite, 337
utilizzo, 690-692
virtuale, 509-531
memoria centralizzata in sistemi multiprocessore, 645
memoria condivisa, 644
distribuita, 669
processori, 631
virtuale, 648
vedi anche memoria
memoria principale, 16, 510
dischi magnetici rispetto a, 17
penalità di miss, 506
tabella delle pagine, 517
velocità, 62
vedi anche memoria
memoria virtuale, 509-531
blocchi, 510
collocamento completamente associativo, 534
combinazioni di eventi, 524
condivisa, 648
DMA e, 603-604
funzione di protezione, 525-527
illustrazione, 511
implementazione della protezione con, 525-527
indirizzo, 514
motivazione, 509-510
page fault, 510
pagine, 510
processo di traduzione, 511, 530
rilocazione, 511
sistemi, 511, 534
write-back, 519
vedi anche indirizzi virtuali
memorie private, 631
memory data register (MDR), 335
metastabilità, 780, 781
metodologia di temporizzazione, 295-297, 776-781
a due fasi, 778-779
sensibile ai fronti, 776-778
sensibile ai livelli, 778-779
metodologia di temporizzazione sensibile ai fronti, 758, 776-778
durata del clock e, 778
fronte di discesa, 758
fronte di salita, 758
spiegazione, 759
svantaggi, 778
vantaggi, 759
metriche, 49-55
formula, 49
indipendenti dall'hardware, 63-65
MFLOPS come, 83-85
MIPS come, 65-66
per diverse applicazioni, 44
vedi anche prestazioni
MFLOPS, 71, 72
affidabilità, 84
come metrica per le prestazioni, 83-85
formula, 83
normalizzati, 84
valutazione, 83-84
microarchitetture, 367
microassembler, 346
microcodice, 808
assemblatore, 808, 810
controllore, 354
memorizzazione, 354
orizzontale, 813
ROM di smistamento, 812
tempo di fetch, 362
traduzione dei campi, 811
unità di controllo come, 809
uso da parte di IBM, 366
vantaggi/svantaggi, 363
verticale, 813
microistruzioni, 345-346, 808
accesso alla memoria, 351

campi, 346, 347
campo ALU control, 347
campo label, 348
campo Memory, 347, 348
campo PCWrite control, 347, 348
campo Register control, 347, 348
campo Sequencing, 347, 348
campo SRC1, 347, 348
campo SRC2, 347, 348
caricamento, 350
definizione del formato, 346-347
destinazione, 347
esecuzione, 346
etichette simboliche, 348
indirizzi, 347
operazione della ALU, 351
per il controllo dell'unità di elaborazione, 354
sequenze, 350
valori dei campi, 348
vedi anche microcodice; microprogramma
Microprocessor without Interlocked Pipelined Stages (MIPS di Stanford), 461
microprocessore, 33, 630
diffusione, 32
economicità, 630
gara di velocità, 634-635
Intel 4004, 32, 33
successo, 35
VLSI, 35
vedi anche processori
microprogramma
come rappresentazione testuale di una macchina a stati finiti, 353
contatore, 354, 809
creazione, 349-353
firmware, 367
formato, 346
funzione di controllo principale, 354
implementazione, 353-355
per istruzioni di accesso alla memoria, 351
per istruzioni di tipo-R, 351
per l'unità di controllo, 352
rappresentazione simbolica, 346
sequenza per i salti condizionati, 352
sequenza per i salti incondizionati, 352
sequenzializzatore, 366
traduzione in hardware, 353, 809-813
vedi anche unità di controllo; microistruzioni
microprogrammazione, 336, 345-355
benefici, 363
diffusione, 367-368
MIMD, 665, 668-670
minicomputer, 32
minimizzazione logica, 808
minterm, 751, 802
MIPS (architettura), 88
allocazione della memoria, 136
classi di istruzioni, 161
codifica delle istruzioni, 100, 130
compilatori, 92, 699
convenzione dei registri, 118
frequenza delle istruzioni, 268
implementazione, 286
indirizzamento, 122-132
indirizzi di memoria, 94
linguaggio assembler, 90, 97, 102, 111, 119, 132, 136, 191, 194, 234, 250
linguaggio macchina, 100, 111, 119, 126-131, 250, 677
mappa dei codici operativi, 722
moltiplicazione, 224
operandi, 97, 102, 111, 119, 132, 186, 191, 194, 234, 250
parole, 179
processore, 88
registri, 97
dimensione, 91
numero di, 91
tempo di accesso, 96
riassunto dei modi di indirizzamento, 128, 129
rilevamento dell'overflow, 189
set di istruzioni, 89, 161, 266
virgola mobile
codifica delle istruzioni, 251
linguaggio macchina, 250
numeri, 235
operandi, 250
MIPS (milioni di istruzioni per secondo)
calcolatore da 1 MIPS, 72
calcolo, 66
come misura di confronto tra calcolatori, 65
come misura di prestazioni, 65-66
di picco, 71
equazione, 65
nativi, 65
relativi, 72
MIPS I, 266, 368
MIPS M/2000, 455
MIPS R2000, 717-740
CPU, 718
FPU, 718
page fault/miss nel TLB e, 527
TLB, 521
MIPS R4000, 545
MISD, 665
miss, 474
cache, 482-483, 489-490, 498
certi, 537, 538
cicli di, 498
di partenza a freddo, 537
in lettura, 486
in scrittura, 488
per capacità, 537, 538
per collisioni, 537
per conflitti, 537
per istruzione, 543
suddivisione, 538
sui dati, 483
sulle istruzioni, 482
TLB, 521, 524, 527-530
vedi anche page fault
miss certi, 537, 538
miss dei dati, 483
miss di partenza a freddo, 537
miss nella cache, 482-483
a corrispondenza diretta, 501
cicli di stallo del processore e, 483
frequenza di, 486, 489, 533
frequenza di clock e, 498
gestione dei, 482
in lettura, 486
in scrittura, 488
passi intrapresi nei, 483
penalità di, 490-491, 495-496
riduzione della penalità, 483
riduzione mediante collocazione flessibile dei blocchi, 499-504
sostituzione del blocco per un, 534
unità di controllo e, 482
vedi anche cache; miss
modalità a pagine, 493
modello dello stack, 165, 173
modello di coerenza della memoria, 638
modulo, 677, 688
cambiamenti, 689
riferimenti, 678
vedi anche programma
moltiplicando, 213
algoritmo di Booth e, 220
scalamento a sinistra, 215, 224
moltiplicatore, 213
algoritmo di Booth e, 220
bit meno significativo del, 215
rappresentazione in complemento a due, 224
valori dei bit del, 224
moltiplicazione, 213-225
con segno, 220, 224
delle mantisse, 245, 246
di numeri binari, 214
di numeri decimali, 213
in virgola mobile, 243-248
moltiplicando, 213, 215, 217, 221
moltiplicatore, 213, 215, 217, 221
nel MIPS, 224
termini, 213
vedi anche aritmetica
Motorola
68000, 166
68881, 276

PowerPC, 149-151
mouse, 8, 568
contatori X/Y, 569
interpretazione della posizione, 568
tasti, 568
versione meccanica, 9
vedi anche dispositivi di I/O
MSI (media scala di integrazione), 367
multicalcolatori, 669
multiplexer, 303, 749, 757, 758
2 in 1, 424
a due ingressi, 749
combinare le unità di elaborazione mediante, 304
componenti, 749
controllo, 312, 337
espanso, 203
implementazione, 749
propagazione, 419, 428
segnali di controllo, 310, 811
spiegazione, 197, 749
multiprocessore
vedi sistemi multiprocessore

N

nastro a scansione elicoidale, 613
Network Computer (NC), 165
no-allocate-on-write, 536
no-fetch-on-write, 536
notazione polarizzata, 187
notazione scientifica, 235
numeri binari, 235
numeri decimali, 235
numeri reali, 236
numeri
binari, 1, 4, 97, 160, 179, 262
con segno, 178-187
decimali, 97, 181
denormalizzati, 258, 289
esadecimali, 134, 185
in complemento a due, 101, 181
in virgola mobile, 235-258
normalizzati, 235
polarizzati, 244
rappresentazione, 180
senza segno, 178-187
numeri binari, 1, 4, 97, 160, 262
ASCII rispetto a, 179
conversione con estensione del segno, 184-185
conversione in decimale, 181
moltiplicazione di, 213
notazione scientifica per, 235
tabella di conversione esadecimale, 185
vedi anche numeri
numeri con segno, 178-187
`addi/slti` e, 195
divisione, 232
moltiplicazione, 220
numeri senza segno rispetto a, 181, 182
vedi anche numeri
numeri decimali, 97
conversione dal binario, 181
divisione, 226
moltiplicazione, 213
notazione scientifica, 235
rappresentazione, 101, 178
vedi anche numeri
numeri senza segno, 178-187
`andi/ori` e, 195
numeri con segno rispetto a, 181, 182
vedi anche numeri
numero di istruzioni, 292
confronto, 54
determinazione, 292
misura, 53
numero di pagina fisica, 511
numero di pagina virtuale, 511, 514
numero di registro, 298, 334
specifica, 298
stadio WB e, 397

O

offset, 95, 96
salto, 126
segmento, 514
offset nella pagina, 511
olografia, 614
operandi, 91-97
costanti, 122-125
divisione, 226
immediati, 122-125
intermedi, spostamento, 216
memoria, 94
MIPS, 97, 102, 111, 132, 186, 191, 194, 234
primo operando sorgente, 99
registro destinazione, 99
secondo operando sorgente, 99
sottrazione e, 188
virgola mobile MIPS, 250
operandi immediati, 122-125
operatore NOT, 746
operazione AND, 193, 201, 746
ALU a 1 bit per, 197
maschera, 193
operazione atomica di scambio, 641
operazione OR, 193, 201, 746
ALU a 1 bit per, 197
esclusivo, 434
operazioni
AND, 193, 746
logiche, 192-196
moltiplicazione, 213-225
nor, 284
NOT, 746
OR, 193, 746
smistamento, 347, 350
somma, 187-192
sottrazione, 187-192
virgola mobile, 83, 84, 251
xor, 284, 726
operazioni di smistamento, 347, 350, 805
operazioni logiche, 192-196
ordinamento per colonne, 254
ordinamento per righe, 254
ordine dei byte, 716
big-endian, 716
little-endian, 716
organizzazione a sala da ballo, 646
overflow, 180, 265
azione predefinita in risposta a, 356
calcolatori Cray, 272
eccezioni, 355, 356, 357, 440-445
esponente, 236
gestione, 189
rilevamento, 189, 190, 212, 356
rilevamento nello stadio EX, 440
soglie, 272
somma, 188, 189
sottrazione, 188, 189
test e, 190
trap in corrispondenza di, 190
verifica, 241, 242
vedi anche eccezione
overlay, 510

P

page fault, 510, 515-518
eccezioni, 528, 529
frequenza, 512
gestione, 527-530
indirizzo virtuale causa del, 528
istruzione, 528
occorrenza dei, 515, 516
passi del sistema operativo, 528-529
penalità, 513
segnalazione, 529
vedi anche miss; memoria virtuale
pagina, 509
dimensione, 512
fisica, 518
indirizzabile, 511
LRU, 517
numero, 511
tener traccia della, 519
vedi anche memoria virtuale
PAL, 753
parallelismo nei dati, 665
parallelismo massiccio, 659-660
ASCI, 659
SIMD, 664
parola
definizione dei campi interni, 193
doppia, 154
indirizzi consecutivi, 139, 159
MIPS, 179
rappresentazione, 178
parola richiesta per prima, 491

PDP-11, 618, 669
penalità di miss, 475
aumento della, 490
cache, 490
calcoli, 492
componente dovuta alla latenza, 490
dimensione dei blocchi e, 490
in lettura, 495
in scrittura, 496
riduzione, 490, 540
riduzione con cache a più livelli, 506-508
verso la memoria principale, 506
vedi anche miss
periodo di clock, 49, 292, 757
aumentare il, 50
determinazione, 293
vincoli, 776
personal computer, 32-34
Alto, 617-618
Apple, 34
IBM, 34
vedi anche elaboratori
piano AND, 752, 753
piano OR, 752, 753
piatti, 569
spiegazione, 570
vedi anche dischi magnetici
più flussi di istruzioni, più flussi di dati (MIMD), 665, 668-670
più flussi di istruzioni, un flusso di dati (MISD), 665, 668
pipeline, 359-360, 376-469
a cinque stadi, 389
analogia del bucato, 376, 377
architetture prive di, 360
CDC 6600, 460
complessità del set di istruzioni e, 455
dinamiche, 388, 445-446, 450-451
eccezioni e, 441
impatto sull'istruzione di salto condizionato, 433
inserimento di buchi, 426
inserimento di stalli, 427
istruzione load attraverso la, 393-395
istruzione store attraverso la, 396-397
paradosso, 376
Pentium Pro, 452-454
Pentium/Pentium Pro, 361
più lunga, 456
PowerPC, 452-454
primo processore con, 361
profondità, 456
progressi, 459-461
rappresentazione grafica, 386, 399-400
sequenza di istruzioni, 426
set di istruzioni, 380
set di istruzioni MIPS, 380
sfruttamento del parallelismo, 388
speedup, 379
statiche, 451
superpipeline, 445
superscalari, 361, 388, 445, 446-449
tempo di esecuzione delle istruzioni e, 388
trabocchetti, 454
virgola mobile, 455
vista d'insieme, 376-389
pipeline dinamiche, 388, 445-446, 450-451
completamento fuori ordine, 451
completamento ordinato, 450
costo, 446
motivazioni, 450
predizione dei salti combinata a, 451
risorse hardware, 447
stazioni di prenotazione, 450
unità principali delle, 450
vedi anche pipeline
pixel, 9
PLA (programmable logic array), 344, 347, 751, 753, 791, 792
caratteristiche, 751
contenuto, 752
dimensione totale, 752, 802
esempio, 752
generazione delle microistruzioni, 362
implementazione dell'unità di controllo a stati finiti, 802
ingressi, 751-752
memorizzazione del microcodice, 354
mintermini, 802
ROM rispetto a, 755
struttura, 752
suddivisione, 802
termini prodotto, 802
polling, 568, 597-599
frequenza, 600
modalità d'uso, 597
overhead, 597-598
svantaggi, 597
porte logiche, 744, 747-748
AND, 197, 199, 747, 791, 792
matrice di, 752
spiegazione, 197
uscita, 330,
logica sparsa, 756
NAND, 748
NOR, 748
accoppiate, 761
OR, 197, 747, 791, 792
due livelli di, 212
matrice di, 752
spiegazione, 197
uscita, 330
OR esclusivo, 213
traduzione dell'unità di controllo della ALU in, 789-792
traduzione dell'unità di controllo principale in, 793
PowerPC, 149-151
cache principale, 541
cicli for, 151
eccezioni/interruzioni e, 355
gerarchia delle memorie, 540-542
indirizzamento di tipo update, 149-150
indirizzamento tramite indice, 149
istruzione moltiplica-e-somma, 259
istruzioni uniche, 150-151
organizzazione delle pipeline, 452-454
penalità di miss, riduzione, 540
salti condizionati, 155
SRAM, 539
supporto alla cache secondaria, 539-540
TLB, 540
traduzione degli indirizzi, 540
virgola mobile, 258
precaricamento, 547
predizione dei salti, 382-384
assumendo che il salto non venga eseguito, 434
buffer, 434, 437
cicli e, 437
comportamento a regime, 437
dinamica, 383, 434, 438
precisione, 437
schedulazione dinamica combinata alla, 451
schema ad 1 bit, 437
schemi a 2 bit, 438
vedi anche salti
predizione dinamica dei salti, 434-438
prestazioni, 43-86
a singolo ciclo, 323-326
aeroplani, 45
aspetti complessi, 45
bus, 584-588
cache, 495-499
confronto, 57-59
CPU, 48
definizione, 45-47
determinazione, 44
di picco, 662
equazione, 51-52
errori e trabocchetti, 62-69
esempio di sequenza di codice, 54-55
fattori chiave che influenzano le, 52
gerarchie di memorie e, 474
kernel e, 73
linguaggio assembler e, 159-160

massimizzare, 46
metodi di determinazione, 45-48
metriche, 44, 48-55
miglioramento, 25, 50-51
miglioramento delle workstation, 25
osservate, 662
pipeline, 378
precisione delle predizioni, 63
prima misura delle, 71
processori paralleli, misurazione, 660
programmi di valutazione, 55-57
rapporti, 83
relative, 47, 58
reti, 577
richiesta di I/O, 609
riproducibilità delle misure, 57
sintesi, 57-59, 69, 74
sistema, 49
sistemi operativi e, 48
sottosistema di I/O, 563, 611
tasso di crescita, 25
tasso di miglioramento, 25
tempo di esecuzione e, 48
vedi anche benchmark
prestazioni a singolo ciclo, 323-326
con istruzioni in virgola mobile, 325-326
conseguenze, 458
prestazioni con pipeline rispetto a, 378-380
prestazioni delle cache, 495-599
a corrispondenza diretta, 501
a più livelli, 506-507
al crescere della frequenza di clock, 498
completamente associative, 499
esempio di calcolo, 497-498
ideali, 497
misurazione, 495-496
set-associative, 501
prestazioni delle pipeline, 449
aumento del throughput e, 380, 388, 449, 456
miglioramento, 380
prestazioni a singolo ciclo rispetto a, 378-380
vedi anche prestazioni
prestazioni dell'I/O, 563
aspetti, 564
misura, 565-567
predizione, 607
prestazioni di picco, 662
irraggiungibili, 662
percentuale delle, 662
prestazioni osservate rispetto a, 662
vedi anche prestazioni
prestazioni di sistema, 49
problema della coerenza, 603
cache, 637
evitare il, 605
procedura azzera
confronto tra le versioni, 148
versione con matrici, 146-147
versione con puntatori, 147-148
procedure, 110-119
allocazione dei registri, 111
annidate, 114-116
che non chiamano altre procedure, compilazione, 113-114
foglie, 114
frame, 117
inline, 144
parametri aggiuntivi e, 118
passi di esecuzione del programma, 110
registri richiesti, 111-114
ricorsive, 115, 174, 696, 698-699
saltare a, 112
stampa, 624, 625
vedi anche chiamata a procedura
procedure ricorsive, 115, 174, 696
esempio, 698-701
iterazione e, 174
stack, 698-699
valori, 174
vedi anche chiamata a procedura; procedure
processo, 515
attivo, 515
cambio di contesto, 527
identificatore, 527
inattivo, 515
spazio di indirizzamento, 515
processore di I/O, 603
periferico, 617
prestazioni e, 609
programmabile, 617
trasferimento delle funzioni dalla CPU al, 609
processore Pentium, 152
algoritmo di divisione in virgola mobile, 262-263
bachi, 263-265
implementazione, 359-362
istruzioni eseguite per ciclo di clock, 361
pipeline, 361
unità di controllo cablata, 367
vedi anche architettura 80x86
processore Pentium Pro, 152
aumento di prestazioni, 61-62
cache principale, 541
confronto, 61-62
frequenze di clock, 362
gerarchia delle memorie, 539-542
hardware del TLB, 540
implementazione, 361-362
istruzioni eseguite per ciclo di clock, 362
organizzazione delle pipeline, 452-454
penalità di miss, riduzione, 540
pipeline, 361, 452-454
segnali di controllo, 362
supporto alla cache secondaria, 539-540
traduzione degli indirizzi, 540
unità di controllo cablata, 367
unità di elaborazione, 362
valore SPECfp95, 62
valore SPECint95, 61
vedi anche architettura 80x86
processori, 11
con spazio di indirizzamento singolo, 630
DMA, 601
I/O, 602, 609
Intel 80x86, 151-158, 166-167
interno, 13
Java, 368
memoria condivisa, 631
MIPS, 91
nell'organizzazione degli elaboratori, 13, 43
parallelismo, 26
Pentium, 152, 361-362
Pentium Pro, 152, 361-362
periferici, 617
tecnologie chiave, 23
vedi anche unità di controllo; unità di elaborazione; microprocessore; multiprocessore
processori paralleli su larga scala, 647-650
cache, 647
memoria virtuale condivisa, 648
opzioni per uno spazio di indirizzamento singolo, 649
protocolli basati su directory, 648
realtà, 657-659
vedi anche sistemi multiprocessore
processori periferici, 617
prodotto, 213
arrotondamento, 246
denormalizzato, 245
normalizzato, 246
parziale, 287
per numeri con segno, 220
scalamento a destra, 217
segno del, 245
termini, 802
vedi anche moltiplicazione
progetto ACS, 461
progetto a singolo ciclo, 321-326
ciclo di clock, 324
confronto con l'unità di controllo a più cicli o con pipeline, 438
CPI, 321
esecuzione delle istruzioni, 391
inefficienza, 326
penalità, 325

periodo del clock, 326
progetto a più cicli rispetto a, 326
unità di elaborazione, 331, 358, 390
progetto di calcolatori
a basso costo, 70
ad alte prestazioni, 70
costo/prestazioni, 70
principi, 160
principio di rendere veloce l'evento più frequente, 124, 160, 292
principio per cui minori sono le dimensioni maggiore è la velocità, 91, 160
principio per cui semplicità e regolarità sono strettamente correlate, 89, 160, 292
principio per cui un buon progetto richiede buoni compromessi, 99, 160
progressi, 3
progetto Whirlwind, 30, 549
programma
avvio, 131-138
condivisione della memoria, 510
fattore di espansione, 681
gerarchia di trasformazione, 133
linguaggio assembler, 132, 681
misure di profiling, 681
moduli, 676, 688
programma Algol 60, 64, 65, 73
programma di I/O, 603
concatenamento, 617
memorizzazione, 603
programma gcc, frequenze di miss, 489, 503
programmatori, 681-682
compilatori rispetto a, 681
incremento di produttività, 6
programmi paralleli, 630, 633
esempio di multiprocessore a bus singolo, 636
esempio di scambio di messaggi, 646
frequenza di sincronizzazione, 659
incremento di velocità, sfida, 634, 635
scrivere, 633
PROM (ROM programmabili), 754
propagazione, 385
con due istruzioni, 385
criticità sui dati e, 413-425
esempio, 420-425
istruzione sub e, 420, 422
istruzioni di store dipendenti e, 424
livelli, 211-212
multiplexer, 428
origine del nome, 389
rappresentazione grafica, 386
registri della pipeline prima della, 418
risoluzione delle criticità mediante, 421
stalli della pipeline e, 387
"super", 211
unità di controllo, 418
validità dei cammini, 386
valori di controllo dei multiplexer, 419
protocolli
bus, 593
coerenza delle cache, 638-640
handshaking, 581, 583, 584
snooping, 638-640
write-invalidate, 639
write-update, 639
TCP/IP, 575, 576
transazioni divise, 588
write-invalidate, 639-641
write-update, 639-643
protocolli per la coerenza delle cache, 638-640
aggiornamento in scrittura (write-update), 639, 643
esempio, 640-641
invalidazione in scrittura (write-invalidate), 639-641
MESI, 641
snooping, 638-640
stato dei blocchi della cache, 640
varianti, 640
protocollo di handshaking, 581
funzionamento, 583
macchina a stati finiti che implementa il controllo del, 584
spiegazione, 583
transazione divisa, 588
vedi anche bus asincroni
protocollo MESI, 641
protocollo write-invalidate, 639-641
MESI, 641
spiegazione, 641
vedi anche protocolli per la coerenza delle cache
protocollo write-update, 639, 643
pseudo istruzione
`abs`, 721
`b`, 728
`beqz`, 730
`bge`, 730
`bgeu`, 730
`bgt`, 730
`bgtu`, 730
`ble`, 730
`bleu`, 731
`blt`, 688, 731
`bltu`, 731
`bnez`, 731
`div`, 724
`divu`, 724
`la`, 732
`ld`, 733
`l.d`, 738
`li`, 727
`l.s`, 738
`mfc1.d`, 735
`move`, 734
`mul`, 721, 724
`mulo`, 724
`mulou`, 724
`neg`, 724
`negu`, 724
`not`, 725
`rem`, 725
`remu`, 725
`rol`, 726
`ror`, 726
`sd`, 734
`s.d`, 739
`seq`, 727
`sge`, 728
`sgeu`, 728
`sgt`, 728
`sgtu`, 728
`sle`, 728
`sleu`, 728
`sne`, 728
`s.s`, 739
`ulh`, 733
`ulhu`, 733
`ulw`, 733
`ush`, 734
`usw`, 734
pseudo istruzioni, 133, 168, 688, 708
identificazione, 721
traduzione SPIM delle, 715
vedi anche le pseudo istruzioni specifiche; istruzioni
pseudo MIPS, 266, 267
puntatore dello stack, 112
modificare, 116
sommare, 112
valori, 114
puntatori
a variabili locali, 160
al frame, 117
allo stack, 112, 114, 116
globali, 118
per azzerare vettori, 147-148
vettori rispetto a, 144-148

Q

quoziente, 226
bit, 229
vedi anche divisione

R

RAID (redundant arrays of inexpensive disks), 612
RAM, 367
EDO, 493, 771
modalità di accesso a colonne, 771
modalità di accesso a nibble, 771
modalità di accesso a pagine, 771
vedi anche DRAM; SRAM
rappresentazione a due livelli, 749
formati, 750
relazione con le tabelle di verità, 751
rappresentazione come prodotto di somme, 750
rappresentazione come somma di prodotti, 750
equazione, 750
esempio, 750
rappresentazione in complemento a due, 101, 181
algoritmo di Booth e, 224
aritmetica intera binaria, 266-267
complemento, 187
numero negativo, 181
rilevamento dell'overflow, 189
scorciatoia binario-esadecimale, 185
scorciatoia per cambiare il segno, 183
scorciatoia per l'estensione del segno, 184
vantaggi, 181
rappresentazione in complemento a uno, 187, 200
complemento a due rispetto a, 187
negativo, 187
rappresentazione in modulo e segno, 180, 236
RAS (Row Access Strobe), 770
reali (numeri), 235
esempi, 235
notazione scientifica, 236
register file, 323, 398, 763-765
accessi, 333, 335
come elemento dell'unità di elaborazione, 299
contenuto, 763
copia, 451
implementazione della porta di lettura, 764
implementazione della porta di scrittura, 764, 765
lettura, 312
scrittura, 336
spiegazione, 763
uscite, 299
registri di pipeline, 392
dipendenze, 416
prima della propagazione, 418
registri temporanei, 92, 96, 114
con controllo della scrittura, 333
implementazione multi-ciclo, 326-328
implementazione con macchina a stati finiti, 344
registro, 91
80x86, 152-154
allocazione nelle chiamate a procedura, 111
ALUOut, 351
associazione delle variabili ai, 92
base, 95, 96
Causa, 356, 357, 359, 444, 599, 703-704
calcolo del, 359
campi implementati da SPIM, 703
memorizzazione delle eccezioni, 444
raccolta delle eccezioni, 444
compilazione di assegnazioni in C usando, 92
contatore, 151
Controllo del Ricevitore, 706
Controllo del Trasmettitore, 707
convenzione MIPS, 118
convenzioni d'uso, 694
coprocessore 0, 702
corrispondenza dei nomi, 97
Dati del Ricevitore, 706
dedicato, 163
destinazione, 99
di codice dell'eccezione, 703
dimensione, 91
dispositivo di I/O, 596
di Stato, 596, 703
Divisore, 227, 229
doppia precisione, 248
EPC, 190, 356
EX/MEM, 392, 394, 396, 417, 420, 424
general-purpose, 163-164
Hi, 225
ID/EX, 392, 396
IF/ID, 392, 395, 396, 433, 434
indice, 95
intero, 158
IR (Instruction register), 333
limite, 517-518
Lo, 225
MEM/WB, 394, 397, 420, 424
Moltiplicatore, 215, 217
nelle istruzioni di trasferimento dati, 96
numero di, 91
per le procedure, 112-114
pipeline, 391-394, 417
prelievo, 333
preservare durante la procedura di ordinamento, 143-144
Prodotto, 216, 217, 218, 221, 222
program counter, 112
Quoziente, 227
regole di utilizzo, 692-701
Resto, 227, 229, 231
rinominare, 452
riservato, 192
riversato, 96, 112
salto, 109, 110
salvato dal chiamante, 695
salvato dal chiamato, 695, 700
singola precisione, 248
special-purpose, 163
tabella delle pagine, 514
tempo di accesso, 96
temporaneo, 92, 96, 113, 327
virgola mobile, 248, 254
resto, 227
calcolo, 232
scalamento a sinistra, 229
segno del, 232
vedi anche divisione
reti, 18, 574-577
a commutazione di pacchetto, 575
a lunga distanza, 575
ampiezza di banda, 575
ARPANET, 575
caratteristiche, 574
crossbar, 655, 656
Ethernet, 18, 574
geografiche (WAN), 18
interconnessione, 653-656
locali (LAN), 18, 574
multi-stadio, 656
Omega, 655, 656
pile di protocolli, 575
prestazioni, 577, 653
switch, 654-655
totalmente connesse, 653, 654, 655
reti Ethernet, 18, 574, 627
algoritmo di back-off esponenziale, 628
cavo coassiale, 627
collisioni, 627
commutate, 574-575, 612
latenza, 577
pacchetti, 574
vedi anche reti
reti geografiche (WAN), 18
reti locali (LAN), 18, 574
scambio di messaggi, 631
vedi anche reti
ricevitore, 705
richiesta di I/O, 562, 594
grande, 564
prestazioni, 610
riduzione della forza, 225
riferimenti, 676
esterni, 689
in avanti, 683
non risolti, 676
rilocazione, 511, 685
riporto veloce, 212
primo livello di astrazione, 206-207
secondo livello di astrazione, 208-210
vedi anche carry lookahead
ripresa immediata (early restart), 491

ritardi delle porte logiche, 212
riversamento dei registri, 96, 112
riduzione, 114
struttura dati, 112
ROM (read-only memory), 344, 347, 367, 753
altezza, 754
ampiezza, 754
forma, 754
implementazione dell'unità di controllo a stati finiti, 796-801
memorizzazione del microcodice, 354
memorizzazione della funzione di controllo nella, 353
PLA rispetto a, 755
programmabile (PROM), 754
smistamento, 805, 806, 807
unità di controllo, 800
velocità, 365
vedi anche memoria
ROM di smistamento, 805, 806, 807
microcodice, 812
riempire il contenuto, 810

S

salti
buffer di previsione, 434
classificazione, 105
condizionati, 103-105, 122, 125-126
condizione nei, 301
creazione da parte dei compilatori, 104
criticità
vedi criticità nel controllo
delay slot, 439
destinazione dei, 432
esecuzione con macchina a stati finiti, 341, 342
incondizionati, 105
indirizzamento nei, 125-127
indirizzi di destinazione, 300, 335
istruzioni, 728-731
lontani, 128
nelle pipeline, 434
nello stadio MEM, 432
non eseguiti, 301, 384, 432
progetto a più cicli e, 330
riduzione del ritardo, 433-434
ritardati, 110, 302, 384, 433, 437-438, 461, 708
stalli, 432
unità di elaborazione, 301, 302, 435
salti condizionati, 103, 161
compilazione dell'istruzione if mediante, 104
compilazione dell'istruzione if-then-else mediante, 105
distinzioni, 105
indirizzamento, 125-126
nel PowerPC, 155
traduzione in linguaggio macchina, 122
vedi anche salti
salti incondizionati, 105
esecuzione con macchina a stati finiti, 342
implementazione, 321
indirizzamento, 125-127
indirizzi, 335
istruzioni, 731
non condizionati (jump), 109, 127, 161
progetto multi-ciclo e, 330
salti ritardati, 110, 302, 384, 461, 708
diffusione e, 438
esecuzione sequenziale delle istruzioni, 384
motivazione, 302
pipeline, 381
slot, 384
vedi anche salti
scalamento, 192
a destra, 217, 262
a sinistra, 215, 223, 262
aritmetico a destra, 222
cifra di guardia, 255
dimensione dello, 193
istruzioni, 194
scambio di messaggi
calcolatori con memorie private e, 631
esempio di programma parallelo, 646-647
su reti LAN, 631
scheda madre, 11
chip, 11
componenti, 11
dissipatore, 14
illustrazione, 12
memoria, 11
processore, 13
visione ravvicinata, 14
schema LRU (usato meno di recente), 506, 516
approssimato, 534
implementazione, 517
schiere (o matrici), 93, 94, 144
azzeramento tramite indici, 146
con indice variabile, 95-96, 106
di byte, 122
di dischi, 628
di flip-flop, 763
di porte AND, 752
di porte OR, 752
di puntatori, 144-148
inizializzazione a zero, 144
schiere di dischi, 628
scorciatoia binario-esadecimale, 185
scorciatoia per il cambio di segno, 183
SDRAM (synchronous DRAM), 493-494
segmentazione, 513
segmenti, 518
aggiungere, 544
di dato, 134, 691
di stack, 691
di testo, 134, 691
numero, 513
offset, 513
tabella delle pagine, 517-518
utilizzo, 513
segnale di controllo NegaB, 202
segnali di controllo, 310, 365
ALU, 205, 789
calcolare i valori dei, 310, 312
campi delle microistruzioni, 347
di 1 bit, 310, 332
di 2 bit, 332
etichette, 405
funzione dei, 310
illustrazione, 309, 311
multiplexer, 811
Pentium Pro, 362
per gli ultimi tre stadi, 407
porre a 0, 432
stadio EX, 400
stadio ID, 400
stadio IF, 400
stadio MEM, 400-407
stadio WB, 407
unità di elaborazione multi-ciclo, 329
unità di elaborazione con pipeline, 404
uscite, 796
valori per la ALU, 332
vedi anche unità di controllo
selettore dati, 303
semafori, 641, 643
semiconduttori, 19
sequenzializzatori
implementazione della funzione di stato futuro mediante, 802-809
microprogramma, 366
set di istruzioni, 88, 365
80x86, 166, 361
accumulatore, 162, 172
complessità dovute alle pipeline, 454
confronti tra elaboratori e, 65
confronto, 172-173
load-store, 172
memoria-memoria, 163, 172
MIPS, 88, 160, 266
MIPS I, 266
modello a stack, 172
pipeline, 380
principi di progetto, 160-161
pseudo MIPS, 266
rappresentazione simbolica, 97
registro-memoria, 163
RISC, 551
VAX, 367
settori, 569

illustrazione, 570
vedi anche dischi magnetici
sfasamento del clock, 777
riduzione, 778
spiegazione, 777
silicio, 19, 26
SIMM (single inline memory module), 14, 550
dimensioni, 550
fotografia, 550
vedi anche memoria
simulatore SPIM, 676, 707-717
abbreviazioni dei comandi, 714
cache e, 715
campi implementati del registro Causa, 703-704
comandi, 705, 714
componenti, 707
console, 715
debugger, 707
esecuzione, 713
istruzione `syscall`, 716-717
latenza della memoria e, 715
opzioni sulla linea di comando, 712-713
ordine dei byte, 716
primi passi, 709-712
registri del coprocessore 0, 702
servizi di sistema, 716-717
simulazione del coprocessore, 717
spim, 709, 713-714
supporto di un sottoinsieme delle direttive assembler, 719-720
terminale, 705
traduzione delle pseudo istruzioni, 715
utilità, 707
velocità, 707
versioni, 709
xspim, 709-712, 715
sincronizzatori, 584, 779-781
con flip-flop D, 781
fallimento, 780
metastabilità e, 780, 781
sincronizzazione, 631
a barriera, 637
con coerenza, 641-644
frequenza di, 659
mancanza di, 584
sindrome di von Neumann, 663
singolo programma con dati multipli (SPMD), 668
sintesi delle prestazioni, 57-59, 69, 74
sistema basato su R3000, 71
sistema di I/O, 562-628
affermazione, 564
Apple Macintosh serie 7200, 607
capacità, 564
caratteristiche, 595
condivisione da parte di più programmi, 595
confronto, 564
esempio di progetto, 606-607
overhead dovuto al polling, 597-599
prestazioni, 563, 611
processo di progettazione, 605-607
progetto, 562, 589, 611
storia, 614-618
sviluppi futuri, 612-614
tipico, 607
uso delle interruzioni, 595
valutazione delle caratteristiche, 611
vincoli sulla larghezza di banda, 605
vincoli sulla latenza, 605
vedi anche I/O; dispositivo di I/O
sistema sincrono, 759
sistemi basati su calcolatore
decomposizione, 7
organizzati su livelli gerarchici, 26
valutazione delle prestazioni, 55
sistemi multiprocessore, 630-674
a bus unico, 635-644
basati su cache, 638
basati su directory, 648
capacità di memoria, 656
con spazio di indirizzamento singolo, 630
connessi ad ipercubo, 654
connessi mediante bus, 655
dimensioni, 657
errori e trabocchetti, 660-662
NUMA, 631, 650
numero di processori, 632
programmazione, 632-635
rete, 644-651
simmetrici (SMP), 631
sincronizzazione, 631
SMP, 631
spinte progettuali, 630
Stanford DASH, 669
su larga scala, 633
sviluppi futuri, 656-660
UMA, 631, 650
vedi anche processori
sistemi multiprocessore a bus unico, 635-644
arbitraggio, 640
caratteristiche, 636
con coerenza delle cache di tipo snooping, 638
illustrazione, 636
limitazioni, 644
mezzo, 644
programma parallelo, 636
protocolli di coerenza delle cache, 637-641
traffico per processore, 635
vedi anche sistemi multiprocessore
sistemi multiprocessore con accesso alla memoria non uniforme (NUMA), 631, 650
sistemi multiprocessore con accesso alla memoria uniforme (UMA), 631, 650
sistemi multiprocessore connessi in rete, 644-650
caratteristiche, 645
costo/prestazioni, 650, 651
organizzazione, 645
su media scala, 644
vedi anche sistemi multiprocessore
sistemi operativi, 6
cambio di contesto, 527
comunicazione con i dispositivi di I/O, 526
implementazione delle protezioni nella memoria virtuale, 525
passi di un page fault, 528
prestazioni e, 48
small computer systems interface
vedi bus SCSI
software
applicativo, 6
di sistema, 6
livelli gerarchici, 7
numerico portabile, 271
prestazioni, miglioramento, 44
somma, 187-192
binaria, 187, 188
carry save, 287
in virgola mobile, 240-243
overflow, 188, 189
vedi anche aritmetica
sommatore a 1 bit, 204
connessione, 199
illustrazione, 198
specifiche di input/output, 198
vedi anche sommatori
sommatore a 4 bit, 206, 211
sommatore a propagazione di riporto, 200
riporto veloce rispetto a, 212
spiegazione, 286
vedi anche sommatori
sommatore carry save, 225, 286, 287
sommatori
a 1 bit, 197, 198, 206
a 4 bit, 206, 211
a 16 bit, 206, 210
CarryIn, 198, 200, 206
CarryOut, 198, 206
carry save, 225, 287
CDC 6600, 273
generazione, 206
logica per il segnale di riporto in uscita, 199
più ampi, 285
propagazione, 206
propagazione del riporto, 200

riporto veloce, 206-212
sottrazione, 200
sottolineatura (_), 719
sottosistema di memoria
opzioni progettuali, 492
supporto delle cache, 491-494
sottrazione, 189, 303, 378, 407-413, 726
binaria, 187
numero negativo da un numero positivo, 189
operandi di ugual segno, 188
overflow, 188, 189
sommatore, 200
vedi anche aritmetica
spazio di indirizzamento, 510
collocazione delle tabelle delle pagine nello, 526
di un processo, 515
piatto, 545
unico, 630, 649
speedup, 85
reale, 661
relativo, 661
spettro evoluzione-rivoluzione, 662-664
spiazzamento, 154
SRAM (static random access memory), 473, 765-769
altezza, 765
ampiezza, 765
configurazione dei chip, 765
costo per megabyte, 473
DRAM rispetto a, 473
organizzazione, 768
PowerPC, 540
sincrone (SSRAM), 767
struttura, 768
tempo di accesso, 473
vedi anche memoria
stack, 112
"crescita", 112
allocazione, 117
chiamate a procedura e, 113
frame di procedura, 117
pop, 112, 114, 116
push, 112
registri preservati, 116
valori, 114
vedi anche procedura
stack di protocolli, 575
stack frame, 693, 697
contenuto, 693, 697
disposizione, 695
metodi di costruzione, 693
nelle procedure ricorsive, 698-699
stadi della pipeline, 376
ciclo di clock, 376
elenco, 389
EX, 392, 394, 396
ID, 392, 393
IF, 392, 393
istruzione load, 391-394
istruzione store, 394-396
MEM, 395, 397
segnali di controllo e, 400-406
WB, 392, 395, 397
stadio EX, 411, 412
campi di controllo, cambiamento dei, 426
criticità, 416
etichetta delle istruzioni, 422
istruzione add nello, 443
istruzione load nello, 394
istruzione nop nello, 426
istruzione store, 396
nomi degli operandi, 422
registri della pipeline durante lo, 417
rilevamento dell'overflow, 441
segnali di controllo, 400
spiegazione, 394, 396
svuotare le istruzioni nello, 440
vedi anche stadi della pipeline
stadio ID, 409, 410
determinazione dei salti eseguiti, 436
esecuzione dei salti nello, 433
identificazione delle criticità nello, 425
istruzione load, 393
istruzione store, 393
segnali di controllo, 400
sommatore per i salti, spostamento nello, 433
spiegazione, 393
svuotamento delle istruzioni nello, 440
vedi anche stadi della pipeline
stadio IF, 410
istruzione load, 393
istruzione store, 393
segnali di controllo, 400
spiegazione, 393
svuotamento delle istruzioni nello, 434, 440
vedi anche stadi della pipeline
stadio MEM, 412
campi di controllo, cambiamento dei, 426
controllo delle criticità, 419
criticità, 416
istruzione load, 395
istruzione store, 397
nomi degli operandi, 422
registri della pipeline durante lo, 417
salti nello, 432
segnali di controllo, 400
spiegazione, 395, 397
vedi anche stadi della pipeline
stadio WB, 392, 415
campi di controllo, cambiamento dei, 426
criticità sui dati e, 419
istruzione load, 395
istruzione store, 397
nomi degli operandi, 422
numero del registro, 395
segnali di controllo, 400
spiegazione, 395, 397
vedi anche stadi della pipeline
stalli della pipeline, 381
criticità sui dati e, 385, 425-432
esempi, 426
evitarli riordinando il codice, 386-387
inattesi, 427
inserimento, 427
necessità, 387
prestazioni dei salti, 382
propagazione e, 385
salti, 432
spiegazione, 382
vedi anche criticità nella pipeline
stallo all'utilizzo, 483
standard per la virgola mobile IEEE 754, 237, 238, 275
chip, 275-276
implementazione, 276
oggi, 277
parti che hanno avuto successo, 277
simboli, 257
vedi anche virgola mobile
Stanford DASH, sistema multiprocessore, 670
stazioni di prenotazione, 450
Stretch (IBM 7030), 70
stringhe
C, come usarle, 121-122
doppi apici (") e, 719
istruzioni, 155-156
procedura di copia, 121-122
scelte per la rappresentazione, 120
Sun SPARC, 545, 716
supercomputer, 32
ASCI, 659
benchmark per l'I/O, 566
costo, 657
vettoriali, 667
vedi anche calcolatori
superpipeline, 445
superscalare, 361, 388, 445, 446-449
analogia del bucato, 445
aumento del throughput, 449
DEC Alpha 21264, 451
espansione dei cicli, 449
funzionamento delle pipeline, 446
limite all'efficacia, 446, 448
macchina, 445
maggior ampiezza, 449
MIPS, 446-449
progetto LIW rispetto a, 461
schedulazione del codice, 448
sviluppo, 461
unità di elaborazione, 447
vedi anche pipeline

svuotamento delle istruzioni, 434, 440
nello stadio EX, 440
nello stadio ID, 440
nello stadio IF, 440
Synapse N+1, 669
System/360, 31-32, 366, 551
cifre di guardia e, 272
doppia precisione, 272
modelli, 31
parole in singola precisione, 271
prezzo/prestazioni, 31
System Performance Evaluation Cooperative
vedi benchmark SPEC

T

tabella dei simboli, 134, 683
tabella delle pagine, 513, 515, 533
calcolo delle dimensioni, 517
contenuto, 515
elementi, 513-514
indicizzare, 513, 514, 515
invertita, 518
memoria principale, 518
memorizzazione, 519
nell'organizzazione della memoria virtuale, 516
nello spazio di indirizzamento del sistema operativo, 526
registro, 514
riduzione delle dimensioni, 517-518
riferimenti, 520
segmento, 518
TLB e, 519-521
vedi anche memoria virtuale
tabelle dei segmenti, 518
tabelle di salto, 109
compilare istruzioni switch mediante, 109-110
tabelle di smistamento, 347, 352, 808
tabelle di verità, 320, 745
esempio, 745
funzione di controllo, 791
funzione di controllo dell'implementazione a singolo ciclo, 320
per i bit di controllo della ALU, 306, 307, 789
per i bit di stato futuro, 798
per le uscite di controllo dell'unità di elaborazione, 799
relazione con le rappresentazioni a due livelli, 751
tag, 479, 487
cache a corrispondenza diretta e, 504
cache completamente associative e, 504
controllo, 504
dimensione rispetto a grado di associatività, 504
memorizzazione, riduzione, 494
TLB, 522
tamburi magnetici, 614, 615
TCP/IP (Transmission Control Protocol/Internet Protocol), 575, 576
tecnologia a bobine, 614
tempo del controllore, 571
tempo di CPU relativo all'utente, 48
tempo di CPU relativo al sistema, 48
tempo di esecuzione, 46, 47
della CPU, 48, 49
di programmi reali come metrica, 70
dopo il miglioramento, 62
frequenza di miss e, 486
media geometrica dei, 69
medio, 59
medio per istruzione, 71
misurazione, 48, 59
normalizzata, media aritmetica, 67-68
numero di istruzioni e, 51
pipeline e, 388
prestazioni e, 48-49
rapporti, 68
relativo, 59
totale, 59, 68
validità del, 69
tempo di esecuzione della CPU, 48, 49
cicli di clock, 495
confronto, 323
misurazione, 52
tempo di hit, 474
tempo di risposta, 46, 48
esempio, 46
minimizzazione, 46
vedi anche prestazioni; throughput
tempo di seek, 570
media misurata, 572
medio, 570
vedi anche dischi magnetici
tempo di trasferimento, 571
tempo medio di accesso alla memoria (AMAT), 555
tempo medio di esecuzione per istruzione, 71
tempo trascorso, 48
temporizzazione a due fasi, 778
progetto, 778-779
spiegazione, 779
temporizzazione sensibile ai livelli, 778
teoremi di DeMorgan, 746, 750, 782
test
disuguaglianza, 107
minore di, 108
uguaglianza, 107
testina di lettura/scrittura, 16
Thinking Machines CM-5, 669
throughput, 46
aumento grazie alle pipeline, 380, 388, 449, 456
aumento superscalare del, 449
benchmark per l'I/O e, 566
esempio, 46
I/O, 578
vedi anche tempo di risposta
TLB (buffer di supporto alla traduzione), 519
aggiornamento, 524
associatività, 520
bit di accesso in scrittura, 524
combinazioni di eventi, 524
come cache, 530
contenuto, 521
DECStation 3100, 522, 523
eccezioni, 529
elementi, 521, 527
hit, 521
integrazione, 524-525
MIPS R2000, 521
miss, 520, 521, 524
elaborazione, 528
gestione, 527-530
indicazione, 527
occorrenza, 527
segnalazione, 529
Pentium Pro, 540
PowerPC, 540
tabelle delle pagine, 519-521
tag, 522
valori, 521
topologie di rete, 653-656
ad anello, 654
crossbar, 655, 656
implementazione, 655-656
importanza, 658
lunghezza dei collegamenti, 655
multistadio, 655, 656
n-cubo booleano, 655
Omega, 655, 656
trabocchetti, 23
attendersi che il miglioramento delle prestazioni di un elemento aumenti in proporzione le prestazioni complessive, 62-63
credere che l'istruzione MIPS addiu estenda il segno al suo campo immediato di 16 bit, 262
dimenticare l'indirizzamento dei byte o la dimensione dei blocchi della cache nella simulazione delle cache, 542
estendere lo spazio di indirizzamento aggiungendo i segmenti ad uno spazio non segmentato, 544-545
ignorare il comportamento del sottosistema di memoria nello scrivere i programmi, 544

ignorare il progresso dell'hardware nel pianificare un nuovo calcolatore, 23
implementare istruzioni complesse in microcodice può non essere più veloce che usare istruzioni più semplici, 362-363
le prestazioni di picco predicono le prestazioni osservate, 661
misurare le prestazioni dei processori paralleli con lo speedup lineare in funzione del tempo di esecuzione, 660
non considerare che il progetto del set di istruzioni può avere un impatto negativo sulle prestazioni della pipeline, 455
sbagliare gli indirizzi di parole sequenziali in sistemi in cui gli indirizzi dei byte non differiscono di uno, 159
spostare delle funzioni dalla CPU al processore di I/O sperando di migliorare le prestazioni, 609-610
usare i MIPS come metrica per le prestazioni, 63-65
usare la frequenza di miss come sola metrica nella valutazione delle gerarchie di memoria, 543
usare la media aritmetica dei tempi di esecuzione normalizzati per predire le prestazioni, 67
usare la velocità di trasferimento di picco per prevedere o confrontare le prestazioni, 609
usare puntatori a variabili automatiche all'esterno della procedura che le definisce, 160
vedi anche errori
tracce, 569
bit, 573
esterne, 573, 574
settori, 569, 570
spiegazione, 570
vedi anche dischi magnetici
traduzione degli indirizzi, 511
Transaction Processing Council (TPC), 566
transazioni sul bus, 578
operazioni di input, 579
operazioni di output, 579
transistor, 19
aumento del numero, 19
pass, 769
trasmettitore, 705

U

ulp (unità nell'ultima posizione), 256
un flusso di istruzioni, più flussi di dati (SIMD), 664
caratteristiche, 666
compromessi, 665
istruzioni, 665
motivazione, 665
parallelismo massiccio, 665
virtù, 665
un flusso di istruzioni, un flusso di dati (SISD), 664
underflow, 237, 265
controllo, 241
graduale, 258
soglie, 273
vedi anche overflow
uni isolati, 225
Unicode, 122
unità aritmetico logica
vedi ALU; unità di controllo della ALU
unità centrale di elaborazione
vedi CPU
unità di controllo, 12, 26, 304, 307-321, 345
a più cicli, 337
cablata, 367
combinatoria, 788-792
come microcodice, 809
compromessi tra gli approcci, 365
con contatore esplicito, 804
con pipeline, 400-413
con unità di elaborazione multi-ciclo, 331, 332
confronto tra gli schemi, 438
controllo delle eccezioni, 357-359
criticità di tipo MEM, 420
della ALU, 304-307, 788-792
equazioni logiche, 795
implementazione, 307-321, 336-345, 364, 365
ingressi, 316
logica di selezione degli indirizzi, 806
metodi di specifica, 366
microprogramma per, 353
microprogrammata, 366, 808
MIPS, 794
miss nella cache e, 482
nell'organizzazione dei calcolatori, 13, 43
ottimizzazione, 808-809
parti della, 788
piccola, 304
progetto, 345-346, 364-365, 367
propagazione, 417
rappresentazioni, 364, 365
riduzione dei circuiti, 810-813
sfide, 355, 364
specifica mediante rappresentazione grafica, 345
unità di rilevamento delle criticità, 425
uscite, 316
vedi anche macchine a stati finiti; microprogramma
unità di controllo a più cicli, metodo delle macchine a stati finiti, 337
unità di controllo cablata, 367
implementazioni, 367
Pentium/Pentium Pro, 368
vedi anche unità di controllo
unità di controllo combinatoria, 788-792
implementazione, 788-792
vedi anche unità di controllo
unità di controllo con pipeline, 400-413
obiettivi, 438
prestazioni, 438
specifica, 400
unità di controllo a singolo ciclo o a più cicli rispetto a, 438
vista d'insieme, 428
unità di controllo della ALU, 304-307
assegnazioni dei segnali, 334
bit della, 306, 307, 789
blocco, 309, 791
classe delle istruzioni e, 306
di 3 bit, 306
ingressi, 306
linee, 205, 790
logica, 791
realizzazione con porte logiche, 788-792
segnali, 339, 791
tabella di verità, 306, 307
unità, 338
uscite, 305
vedi anche ALU; unità di controllo
unità di elaborazione, 12, 26
a singolo ciclo, 330, 361, 364, 390
composizione, 303-304
con i controlli per gestire le eccezioni, 442
con l'unità di controllo, 311
con multiplexer, 309
con pipeline, 389-400
conseguenze sulle prestazioni, 458
costruzione, 297-302
funzionamento della, 312-316
multi-ciclo, 327, 328, 329, 361, 364
nell'organizzazione dei calcolatori, 13, 43
parte relativa al prelievo delle istruzioni, 305
Pentium Pro, 361
per istruzioni di tipo-R, 299
per l'istruzione beq, 319
per l'istruzione jump, 322
per le istruzioni load/store, 300
per prelevare le istruzioni, 298
processore MIPS, 364

salti, 301, 302, 435
schema di controllo, 320
segnali di controllo, 310
superscalare, 447
vedi anche elementi dell'unità di elaborazione
unità di elaborazione con pipeline, 389-400
con cinque stadi, 398
con i segnali di controllo, 404
con i segnali di controllo connessi ai registri di pipeline, 408
finale, 457
modifiche per risolvere le criticità mediante propagazione, 421
significato dei segnali di controllo e, 406
spiegazione, 389, 455
vedi anche stadi della pipeline
unità di propagazione, 418, 420, 430
propagazione dei dati verso la ALU, 423
spiegazione, 428
unità di rilevamento delle criticità, 425
criticità di caricamento-uso e, 426
funzione, 427
spiegazione, 427
unità di controllo per, 425
UNIVAC I, 30
uscita, 26
ALU, 334, 357
buffer, 625
dispositivi, 8
elemento di stato, 295, 759
funzione, 773-776
indifferente (don't care), 755
memoria, 338
nell'I/O, 594-605
nell'organizzazione dei calcolatori, 13, 43
porta AND, 330
porta OR, 330
register file, 299
segnale di controllo, 796
unità di controllo, 316
unità di controllo della ALU, 304
uscite di stato futuro, 796
bit, tabelle di verità dei, 798
equazioni logiche, 796
tabella di verità, 801

V

variabili
accumulatori e, 162
automatiche, 118, 160
cicli for, 252
di lock, 641, 643
statiche, 118
velocità di trasferimento, 566
very large scale integrated circuits (VLSI), 19
componenti, 19
microprocessori, 35
very long instruction word (VLIW), 461
video CRT (tubo a raggi catodici), 9
frequenza di rinfresco, 9
pixel, 9
vedi anche video grafico
video grafico, 9-11
a scala di grigi, 9
buffer di quadro, 10
CRT, 9
LCD, 10
vincolo di allineamento, 94
virgola mobile, 235-261, 265
compilazione di procedure C, 252-254
conversione da binario a decimale, 239
coprocessore Intel 8087, 259
doppia precisione, 237, 252
esempio di rappresentazione, 239
hardware, 270
istruzioni, 325-326, 736-740
istruzioni MIPS, 236
limitazioni sulla precisione, 262
moltiplicazione, 243-248
esempio, 247
passi, 243-246
nel PowerPC, 258
numeri, 236-240
operazioni, 83, 84, 250
precisione, 242
rappresentazione, 236-240
registri separati, 249, 254
singola precisione, 237
somma, 240-243
decimale, 242
diagramma a blocchi dell'unità aritmetica, 244
errore di associatività, 261-262
esempio, 242-243
passi, 241-242
standard IEEE 754, 237, 238
unità, 254
unità decimale, 270
vedi anche aritmetica
virgolette (''), 719
von Neumann, John, 28, 29, 30, 269, 271

W

wafer, 20
costo, 23
difetti, 20, 23
divisione, 20
resa, 20, 40
vedi anche die
Wilkes, Maurice, 28, 269, 549
Wozniak, Steve, 32

X

xspim, 709
avviare, 709
breakpoint, 712
finestra di console, 711, 715
interfaccia, 710
opzioni sulla linea di comando, 712-713
pannelli, 709, 710
pulsante breakpoint, 712, 715
pulsante clear, 712, 715
pulsante continue, 711, 712
pulsante help, 715
pulsante load, 711, 715
pulsante mode, 715
pulsante assembly file, 711
pulsante print, 715
pulsante quit, 715
pulsante run, 711, 715
pulsante set value, 715
pulsante step, 711, 715
pulsante terminal, 715
vedi anche simulatore SPIM

Z

Zuse, Konrad, 29, 269

Operandi MIPS

Nome	Esempio	Commenti
32 registri	$s0-$s7, $t0-$t9, $gp, $fp, $zero, $sp, $ra, $at, Hi, Lo	Accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. Il registro $zero ha sempre valore 0. Il registro $at è riservato all'assembler per la gestione di costanti di grandi dimensioni. Hi e Lo contengono il risultato di divisione e moltiplicazione.
2^{30} parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte e quindi le parole successive differiscono di 4 unità. La memoria mantiene le strutture dati, come i vettori, i registri riversati (spilled) e i registri salvati durante le chiamate a procedura.

Linguaggio assembler MIPS

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	somma	add $s1, $s2, $s3	$s1 = $s2 + $s3	Tre operandi; dati nei registri
	sottrazione	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Tre operandi; dati nei registri
	somma immediata	addi $s1, $s2, 100	$s1 = $s2 + 100	Usata per sommare costanti
	somma senza segno	addu $s1, $s2, $s3	$s1 = $s2 + $s3	Tre operandi; nessun test sull'overflow
	sottrazione senza segno	subu $s1, $s2, $s3	$s1 = $s2 - $s3	Tre operandi; nessun test sull'overflow
	somma immediata senza segno	addiu $s1, $s2, 100	$s1 = $s2 + 100	Somma una costante; nessun test sull'overflow
	sposta dal registro del coprocessore	mfc0 $s1, $epc	$s1 = $epc	Copia l'Exception PC ed altri registri speciali
	moltiplicazione	mult $s2, $s3	Hi, Lo = $s2 × $s3	Prodotto con segno su 64 bit in Hi e Lo
	moltiplicazione senza segno	multu $s2, $s3	Hi, Lo = $s2 × $s3	Prodotto senza segno su 64 bit in Hi e Lo
	divisione	div $s2, $s3	Hi, Lo = $s2 / $s3	Lo = quoziente, Hi = resto
	divisione senza segno	divu $s2, $s3	Hi, Lo = $s2 / $s3	Quoziente e resto senza segno
	move from Hi	mfhi $s1	$s1 = Hi	Esegue una copia di Hi
	move from Lo	mflo $s1	$s1 = Lo	Esegue una copia di Lo
Logica	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	Tre operandi registri; AND bit a bit
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	Tre operandi registri; OR bit a bit
	and immediato	andi $s1, $s2, 100	$s1 = $s2 & 100	AND bit a bit tra registro e costante
	or immediato	ori $s1, $s2, 100	$s1 = $s2 ! 100	OR bit a bit tra registro e costante
	scalamento logico a sinistra	sll $s1, $s2, 10	$s1 = $s2 << 10	Scalamento a sinistra di una costante
	scalamento logico a destra	srl $s1, $s2, 10	$s1 = $s2 >> 10	Scalamento a destra di una costante
Trasferimento dati	load word	lw $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Word dalla memoria al registro
	store word	sw $s1, 100($s2)	Memoria[$s2 + 100] = $s1	Word dal registro alla memoria
	load byte unsigned	lbu $s1, 100($s2)	$s1 = Memoria[$s2 + 100]	Byte dalla memoria al registro
	store byte	sb $s1, 100($s2)	Memoria[$s2 + 100] = $s1	Byte dal registro alla memoria
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Carica una costante nei 16 bit più significativi
Salto condizionato	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Controllo dell'uguaglianza ed eventuale salto; salto PC-relative
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Controllo della disuguaglianza ed eventuale salto; salto PC-relative

(segue)

Categoria	Istruzione	Esempio	Significato	Commenti
Salto condizionato	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0;	Confronto del tipo "minore di"; completamento a due
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0;	Confronto con una costante; completamento a due
	set less than unsigned	sltu $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0;	Confronto del tipo "minore di"; numeri senza segno
	set on less than immediate unsigned	sltiu $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0;	Confronto con una costante; numeri senza segno
Salto incondizionato	jump	j 2500	go to 10000	Salto ad un indirizzo specificato
	jump register	jr $ra	go to $ra	Usato nei costrutti switch e nell'uscita dalle procedure
	jump and link	jal 2500	$ra = PC + 4; go to 10000	Per le chiamate a procedura

Operandi MIPS per virgola mobile

Nome	Esempio	Commenti
32 registri in virgola mobile	$f0, $f1, $f2, ..., $f31	I registri MIPS per la virgola mobile sono usati in coppia per i numeri in doppia precisione.
2^{30} parole di memoria	Memoria[0], Memoria[4],..., Memoria[4294967292]	Accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte e quindi le parole successive differiscono di 4 unità. La memoria mantiene le strutture dati, come i vettori, i registri riversati (spilled) e i registri salvati durante le chiamate a procedura.

Linguaggio assembler MIPS per virgola mobile

Categoria	Istruzione	Esempio	Significato	Commenti
Aritmetica	FP add single	add.s $f2, $f4, $f6	$f2=$f4+$f6	Somma in virgola mobile (singola precisione)
	FP subtract single	sub.s $f2, $f4, $f6	$f2=$f4-$f6	Sottrazione in virgola mobile (singola precisione)
	FP multiply single	mul.s $f2, $f4, $f6	$f2=$f4×$f6	Moltiplicazione in virgola mobile (singola precisione)
	FP divide single	div.s $f2, $f4, $f6	$f2=$f4/$f6	Divisione in virgola mobile (singola precisione)
	FP add double	add.d $f2, $f4, $f6	$f2=$f4+$f6	Somma in virgola mobile (doppia precisione)
	FP subtract double	sub.d $f2, $f4, $f6	$f2=$f4-$f6	Sottrazione in virgola mobile (doppia precisione)
	FP multiply double	mul.d $f2, $f4, $f6	$f2=$f4×$f6	Moltiplicazione in virgola mobile (doppia precisione)
	FP divide double	div.d $f2, $f4, $f6	$f2=$f4/$f6	Divisione in virgola mobile (doppia precisione)
Trasferimento dati	load word copr.1	lwc1 $f1, 100($s2)	$f1=Memoria[$s2+100]	Trasferimento dati su 32 bit ad un registro in virgola mobile
	store word copr.1	swc1 $f1, 100($s2)	Memoria[$s2+100]=$f1	Trasferimento dati su 32 bit in memoria
Salti condizionati	Branch on FP true	bc1t 25	se (cond==1) vai a PC+4+100	Salto relativo al PC su condizione vera in virgola mobile
	Branch on Fp false	bc1f 25	se (cond==0) vai a PC+4+100	Salto relativo al PC su condizione falsa in virgola mobile
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2, $f4	se ($f2<$f4) cond=1, altrimenti cond=0	Confronto in virgola mobile singola precisione
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2, $f4	se ($f2<$f4) cond=1, altrimenti cond=0	Confronto in virgola mobile doppia precisione

Linguaggio macchina MIPS

Nome istruzione	Formato	Esempio 6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Commenti
add	R	0	18	19	17	0	32	add $1, $2, $3
sub	R	0	18	19	17	0	34	sub $1, $2, $3
addi	I	8	2	1	100			addi $1, $2, 100
addu	R	0	2	3	1	0	33	addu $1, $2, $3
subu	R	0	2	3	1	0	35	subu $1, $2, $3
addiu	I	9	2	1	100			addiu $1, $2, 100
mfc0	R	16	0	1	14	0	0	mfc0 $1, $epc
mult	R	0	2	3	0	0	24	mult $2, $3
multu	R	0	2	3	0	0	25	multu $2, $3
div	R	0	2	3	0	0	26	div $2, $3
divu	R	0	2	3	0	0	27	divu $2, $3
mfhi	R	0	0	0	1	0	16	mfhi $1
mflo	R	0	0	0	1	0	18	mflo $1
and	R	0	2	3	1	0	36	and $1, $2, $3
or	R	0	2	3	1	0	37	or $1, $2, $3
andi	I	12	2	1	100			andi $1, $2, 100
ori	I	13	2	1	100			ori $1, $2, 100
sll	R	0	0	2	1	10	0	sll $1, $2, 10
srl	R	0	0	2	1	10	2	srl $1, $2, 10
lw	I	35	18	17	100			lw $1, 100($2)
sw	I	43	18	17	100			sw $1, 100($2)
lui	I	15	0	1	100			lui $1, 100
beq	I	4	1	2	25			beq $1, $2, 100
bne	I	5	1	2	25			bne $1, $2, 100
slt	R	0	2	3	1	0	42	slt $1, $2, $3
slti	I	10	2	1	100			slti $1, $2, 100
sltu	R	0	2	3	1	0	43	sltu $1, $2, $3
sltiu	I	11	2	1	100			sltiu $1, $2, 100
j	J	2	2500					j 10000
jr	R	0	31	0	0	0	8	jr $31
jal	J	3	2500					jal 10000

Linguaggio macchina MIPS per virgola mobile

Nome	Formato	Esempio						Commenti
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Dimensione del campo		6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Tutte le istruzioni MIPS sono lunghe 32 bit

Formati delle istruzioni MIPS

Linguaggio macchina MIPS. Sono mostrati i formati e gli esempi, con i valori riportati in ciascun campo: i campi op e funct (di 6 bit ciascuno) costituiscono il codice operativo, il campo rs (5 bit) rappresenta uno dei registri sorgente, il campo rt (5 bit) solitamente è un altro registro sorgente, il campo rd (5 bit) è il registro destinazione, ed il campo shamt (5 bit) è il numero di posizioni di scalamento. I campi valore sono espressi in notazione decimale. L'Appendice A riporta il set di istruzioni completo del linguaggio assembler MIPS.

Nome	Campo						Commenti
Dimensione del campo	6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Tutte le istruzioni MIPS sono lunghe 32 bit
Formato-R	op	rs	rt	rd	shamt	funct	Formato istruzioni aritmetiche
Formato-I	op	rs	rt	address/immediate			Formato istruzioni di trasferimento dati e istruzioni di salto condizionato
Formato-J	op	target address					Formato istruzioni di salto incondizionato