Carl Hamacher
Zvonko Vranesic
Safwat Zaky
Naraig Manjikian

UNIVERSITÀ DEGLI STUDI DI CATANIA
BIBLIOTECA DELLE SCIENZE MATEMATICHE E INFORMATICHE

Introduzione all'architettura dei calcolatori

BIBLIOTECA DELLE SCIENZE MATEMATICHE E INFORMATICHE
AC/4
110
UNIVERSITÀ DEGLI STUDI CATANIA

Terza edizione

Edizione italiana a cura di Giuseppe Scollo
Traduzione a cura di Emiliano Tramontana

UNIVERSITÀ DEGLI STUDI DI CATANIA
Biblioteca delle Scienze matematiche ed informatiche
Inventario n° 805 ia 000022

McGraw-Hill

Milano • New York • San Francisco • Washington D.C. • Auckland
Bogotá • Lisboa • London • Madrid • Mexico City • Montreal
New Delhi • San Juan • Singapore • Sydney • Tokyo • Toronto

Titolo originale: *Computer Organization and Embedded Systems, Sixth Edition*
Copyright © 2012, 2002, 1996, 1990 by The McGraw-Hill Companies, Inc.

Copyright © 2013, 2007 McGraw-Hill Education (Italy) s.r.l.
Via Ripamonti 89
20141 Milano

McGraw-Hill
A Division of The McGraw-Hill Companies

I diritti di traduzione, di riproduzione, di memorizzazione elettronica e di adattamento totale e parziale con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche) sono riservati per tutti i Paesi.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

Le fotocopie per *uso personale* del lettore possono essere effettuate nei limiti del 15% di ciascun volume/fascicolo di periodico dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941, n. 633.

Le riproduzioni effettuate per finalità di carattere professionale, economico o commerciale o comunque *per uso diverso da quello personale* possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEARedi, Corso di Porta Romana 108, 20122 Milano, e-mail segreteria@clearedi.org e sito web www.clearedi.org.

Publisher: Paolo Roncoroni
Development Editor: Filippo Aroffo
Produzione: Donatella Giuliani
Traduzione terza edizione: Emiliano Tramontana
Traduzione seconda edizione: Luca Breveglieri, Graziano Pravadelli (Appendice A, Caso di studio C2)
Realizzazione editoriale: Fotocompos s.r.l., Gussago (BS)
Grafica di copertina: FeelItalia, Milano
Immagine di copertina: *CPU Stream* @Andrew Ostrovsky
Stampa: La Litotipo - Settimo Milanese (MI)

ISBN 978-88-386-6751-0
Printed in Italy
123456789LITLIT76543
Finito di stampare nel mese di dicembre 2012

Indice breve

1 Introduzione al calcolatore 1

2 Istruzioni macchina 33

3 Operazioni di ingresso e uscita 109

4 Software 143

5 Struttura di base del processore 163

6 Introduzione al pipelining 195

7 Sistema di ingresso e uscita 221

8 Sistema di memoria 255

9 Aritmetica 321

10 Sistemi embedded 359

11 Elaborazione parallela e prestazioni 387

Caso di studio C1 Progetto di un sistema su un chip 405

Caso di studio C2 Architettura Intel IA-32 407

Appendice A Elementi di logica della commutazione 409

Eserciziario E1

Indice

Prefazione XV

Autori XX

Ringraziamenti dell'Editore XXI

Guida alla lettura XXII

1 Introduzione al calcolatore 1

1.1 Famiglie di calcolatori 1

1.2 Componenti funzionali 2
1.2.1 Unità di ingresso 4
1.2.2 Unità di memoria 4
1.2.3 Unità aritmetica-logica 5
1.2.4 Unità di uscita 6
1.2.5 Unità di controllo 6

1.3 Concetti operativi di base 7

1.4 Rappresentazione dei numeri e operazioni aritmetiche 10
1.4.1 Numeri interi 10
1.4.2 Numeri frazionari 16

1.5 Rappresentazione dei caratteri 17

Approfondimento applicativo 1.1
Codici di controllo e correzione di errore 19

Approfondimento applicativo 1.2
Rappresentazione binaria di informazione multimediale 22

Approfondimento applicativo 1.3
Compressione di dati 24

1.6 Considerazioni di efficienza 26
1.6.1 Tecnologia 26
1.6.2 Parallelismo 26

1.7 Breve storia del calcolatore 27
1.7.1 Prima generazione 29
1.7.2 Seconda generazione 29
1.7.3 Terza generazione 30
1.7.4 Quarta generazione 30

Sommario 31

Note bibliografiche 31

2 Istruzioni macchina 33

2.1 **Memoria del calcolatore** 34
2.1.1 Indirizzamento di byte 36
2.1.2 Ordinamento di byte 36
2.1.3 Allineamento di parola 37
2.1.4 Dati numerici e testuali 38

2.2 **Operazioni di memoria** 38

2.3 **Istruzioni macchina di base** 38
2.3.1 Trasferimento di registro 39
2.3.2 Notazione simbolica 40
2.3.3 Insiemi di istruzioni RISC e CISC 40
2.3.4 Introduzione agli insiemi di istruzioni RISC 41

Approfondimento applicativo 2.1
Architetture di insiemi di istruzioni di processori in commercio: NIOS II, ColdFire, ARM, IA-32 42
2.3.5 Esecuzione Sequenziale 44
2.3.6 Esecuzione con salto 45
2.3.7 Calcolo di indirizzo 48

2.4 **Modi di indirizzamento** 48
2.4.1 Costante e variabile 49
2.4.2 Modo indiretto e puntatore 50
2.4.3 Modo con indice e vettore 52

Approfondimento applicativo 2.2
Modi di indirizzamento NIOS II, ColdFire, ARM, IA-32 56

2.5 **Linguaggio assemblativo** 58
2.5.1 Direttive di assemblatore 59
2.5.2 Assemblaggio ed esecuzione 62

Approfondimento applicativo 2.3
Direttive di assemblatore e sintassi di GNU Assembler 63
2.5.3 Notazione per i numeri 67

2.6 **Gestione di pila** 67

2.7 **Gestione di sottoprogramma** 69
2.7.1 Annidamento e pila 70
2.7.2 Passaggio di parametri 71
2.7.3 Area di attivazione 75

2.8 **Altre istruzioni macchina** 80
2.8.1 Istruzioni logiche 80

Approfondimento applicativo 2.4
Collegamento di sottoprogrammi, macroistruzioni 81
2.8.2 Scorrimento e rotazione 83
2.8.3 Moltiplicazione e divisione 86

2.9 **Trattamento di valori immediati a 32 bit** 88

2.10 **Insiemi di istruzioni CISC** 89
2.10.1 Altri indirizzamenti 90
2.10.2 Bit di esito o condizione 91

2.11 **Stili RISC e CISC** 93

2.12 **Esempi di programma** 94
2.12.1 Prodotto scalare di vettori 94
2.12.2 Programma di ricerca di una stringa 95

Approfondimento applicativo 2.5
Codici operativi e sintassi delle istruzioni 96

2.13 **Codifica numerica di istruzione** 103

Approfondimento applicativo 2.6
Esempi e problemi di programmazione assemblativa 104

Sommario 107
Note bibliografiche 107

3 Operazioni di ingresso e uscita 109

3.1 **Accesso a dispositivo di I/O** 109
3.1.1 Interfaccia dei dispositivi di I/O 110
3.1.2 I/O controllati da programma 111
3.1.3 Un esempio di un programma per I/O in stile RISC 114
3.1.4 Un esempio di un programma per I/O in stile CISC 114

3.2 **Tecnica di interruzione** 116

Approfondimento applicativo 3.1
I/O nei processori NIOS II, ARM, ColdFire e IA-32 117
3.2.1 Controllo di interruzione 122
3.2.2 Gestione di dispositivi multipli 123
3.2.3 Controllo della richiesta 125
3.2.4 Registri di controllo del processore 126
3.2.5 Esempi di programmi con interruzioni 127
3.2.6 Concetto di eccezione 132

Approfondimento applicativo 3.2
I/O con interruzioni NIOS II, ARM, ColdFire, IA-32 134

Approfondimento applicativo 3.3
Eccezioni nei processori NIOS II, ARM, ColdFire, IA-32 139

Sommario 141

4 Software 143

4.1 **Processo assemblativo** 143
4.1.1 Assemblatore a due passi 144

4.2 **Caricare ed eseguire programmi oggetto** 144

4.3 **Il collegatore** 145

4.4 **Librerie** 146

4.5 **Il compilatore** 146
4.5.1 Ottimizzazioni del compilatore 147
4.5.2 Combinare programmi scritti in linguaggi diversi 147

4.6 **Il debugger** 147

4.7 **Uso di un linguaggio di alto livello per operazioni di I/O** 150

4.8 **Interazioni tra linguaggio assemblativo e linguaggio C** 152

4.9 Sistema operativo 155
4.9.1 Processo di avvio 156
4.9.2 Gestione dell'esecuzione di programmi di applicazioni 157
4.9.3 Uso delle interruzioni in un sistema operativo 158
Sommario 161
Note bibliografiche 161

5 Struttura di base del processore 163

5.1 Alcuni concetti fondamentali 163
5.2 Esecuzione di istruzioni 166
5.2.1 Istruzioni di caricamento 166
5.2.2 Istruzioni aritmetiche e logiche 167
5.2.3 Istruzioni di immagazzinamento 168
5.3 Componenti hardware 169
5.3.1 Banco di registri 169
5.3.2 ALU 169
5.3.3 Percorso dati 171
5.3.4 Sezione di prelievo delle istruzioni 174
5.4 Prelievo di istruzioni e passi di esecuzione 176
5.4.1 Salti 178
5.4.2 In attesa della memoria 181
5.5 Segnali di controllo 181
5.6 Controllo di tipo cablato 185
5.6.1 Segnali di controllo del percorso dati 186
5.6.2 Gestione dei ritardi della memoria 187
5.7 Processori in stile CISC 187
5.7.1 Interconnessione che usa i bus 189
5.7.2 Controllo microprogrammato 191
Sommario 193

6 Introduzione al pipelining 195

6.1 Concetti fondamentali 195
6.2 Organizzazione in pipeline 196
6.3 Problematiche del pipelining 198
6.4 Dipendenze di dato 198
6.4.1 Inoltro di operandi 199
6.4.2 Gestione in software di dipendenze di dato 200
6.5 Ritardi della memoria 201
6.6 Ritardi nei salti 203
6.6.1 Salti incondizionati 203
6.6.2 Salti condizionati 204
6.6.3 Il posto del ritardo di salto 205
6.6.4 Predizione di salti 206
6.7 Limiti di risorse 209

6.8 Valutazione della prestazione 210
6.8.1 Effetti di stalli, penalità e cache miss 211
6.8.2 Numero di stadi della pipeline 212
6.9 Funzionamento superscalare 212
6.9.1 Salti e dipendenze di dato 214
6.9.2 Esecuzione fuori ordine 215
6.9.3 Completamento dell'esecuzione 216
6.9.4 Funzionamento dello smistamento 217
6.10 Pipeline nei processori CISC 218
6.10.1 Pipeline nei processori ColdFire 219
6.10.2 Pipeline nei processori Intel 219
Sommario 220
Note bibliografiche 220

7 Sistema di ingresso e uscita 221

7.1 Struttura a bus 221
7.2 Funzionamento del bus 223
7.2.1 Bus sincrono 223
7.2.2 Bus asincrono 228
7.2.3 Considerazioni elettriche 232
7.3 Arbitraggio 232
7.4 Interfacce di I/O 234
7.4.1 Interfaccia parallela 234
7.4.2 Interfaccia seriale 238
7.5 Standard di interconnessione 242
7.5.1 Bus seriale universale (USB) 242
7.5.2 FireWire 246
7.5.3 Bus PCI 247
7.5.4 Bus SCSI 251
7.5.5 SATA 252
7.5.6 SAS 253
7.5.7 PCI Express 253
Sommario 254
Note bibliografiche 254

8 Sistema di memoria 255

8.1 Concetti fondamentali 255
8.2 Memoria RAM a semiconduttori 257
8.2.1 Componente di memoria 258
8.2.2 Memoria statica 259
8.2.3 Memoria dinamica asincrona 262
8.2.4 Memoria dinamica sincrona 265
8.2.5 Banco di memoria 268
8.3 Memoria a sola lettura 271
8.3.1 Memoria ROM 271
8.3.2 Memoria PROM 272

8.3.3 Memoria EPROM 273
8.3.4 Memoria EEPROM 273
8.3.5 Memoria flash 274

8.4 Tecnica di DMA 275

8.5 Gerarchia di memoria 278

8.6 Memoria cache 280
8.6.1 Schema di indirizzamento 284
8.6.2 Algoritmo di sostituzione 289
8.6.3 Esempio di indirizzamento 290

8.7 Considerazioni di prestazione 294
8.7.1 Guadagno di prestazione 295
8.7.2 Integrazione con il processore 296
8.7.3 Altre migliorie 297

8.8 Memoria virtuale 299
8.8.1 Traduzione di indirizzo 300

8.9 Gestione della memoria 306

8.10 Memoria di massa 306
8.10.1 Disco magnetico 307
8.10.2 Dischi ottici 313
8.10.3 Sistemi a nastri magnetici 317

Sommario 318
Note bibliografiche 319

9 Aritmetica 321

9.1 Addizionatore e sottrattore 321
9.1.1 Unità logica per addizione e sottrazione 323

9.2 Progettazione di addizionatori veloci 325
9.2.1 Addizione con anticipo del riporto 325

9.3 Moltiplicazione di numeri senza segno 329
9.3.1 Moltiplicatore a matrice 330
9.3.2 Circuito moltiplicatore sequenziale 331

9.4 Moltiplicazione di numeri con segno 333
9.4.1 Algoritmo di Booth 333

9.5 Moltiplicazione veloce 336
9.5.1 Ricodifica bit-pair dei moltiplicatori 336
9.5.2 Addizione di addendi con salvataggio del riporto 337
9.5.3 Albero di addizione degli addendi con riduttori 3-2 339
9.5.4 Albero di addizione degli addendi con riduttori 4-2 341
9.5.5 Sommario della moltiplicazione veloce 344

9.6 Divisione di interi 344

9.7 Numeri e operazioni in virgola mobile 347
9.7.1 Operazioni Fondamentali 351
9.7.2 Bit di guardia e troncamento 352
9.7.3 Realizzazione di operazioni in virgola mobile 354

9.8 Conversione da decimale a binario 356

Sommario 357
Note bibliografiche 358

10 Sistemi embedded 359

10.1 Esempi di sistemi embedded 359
10.1.1 Forno a microonde 360
10.1.2 Macchina fotografica digitale 361
10.1.3 Domotica 363

10.2 Chip microcontrollori per applicazioni embedded 363

10.3 Un microcontrollore semplice 364
10.3.1 Interfacce parallele di I/O 365
10.3.2 Interfaccia di I/O seriale 368
10.3.3 Contatore/Temporizzatore 369
10.3.4 Meccanismo di controllo delle interruzioni 371
10.3.5 Esempi di programmazione 372

10.4 Cronometro di riflessi, un esempio completo 372

10.5 Sensori e attuatori 375
10.5.1 Sensori 375
10.5.2 Attuatori 379
10.5.3 Esempi di applicazione 380

10.6 Famiglie di microcontrollori 381
10.6.1 Microcontrollori basati su processore Intel 8051 381
10.6.2 Microcontrollori Freescale 382
10.6.3 Microcontrollori ARM 382

10.7 Aspetti di progettazione 383
Sommario 385
Note bibliografiche 386

11 Elaborazione parallela e prestazioni 387

11.1 Multithreading hardware 387

11.2 Elaborazione vettoriale (SIMD) 388
11.2.1 Unità di elaborazione grafica (GPU) 391

11.3 Multiprocessori a memoria condivisa 391
11.3.1 Reti di interconnessione 393

11.4 Coerenza di cache 396
11.4.1 Protocollo di scrittura immediata 396
11.4.2 Protocollo di scrittura differita 396
11.4.3 Cache snoopy 397
11.4.4 Coerenza di cache basata su directory 398

11.5 Multicalcolatori a scambio di messaggi 399

11.6 Programmazione parallela per multiprocessori 399

11.7 Modellazione della prestazione 402
Sommario 403
Note bibliografiche 404

Caso di studio C1 Progetto di un sistema su un chip 405

Caso di studio C2 Architettura Intel IA-32 407

Appendice A Elementi di logica della commutazione 409

A.1 Funzioni logiche fondamentali 409

A.2 Sintesi di funzioni logiche 413
A.2.1 Sintesi intuitiva 414
A.2.2 Forma canonica 416

A.3 Sintesi in forma minima 417
A.3.1 Metodo di Karnaugh 421
A.3.2 Condizione di indifferenza 425

A.4 Sintesi con porte universali 426

A.5 Tecnologia microelettronica 430
A.5.1 Tecnologia CMOS 432
A.5.2 Ritardo di propagazione 438
A.5.3 Vincoli di fan-in e fan-out 439
A.5.4 Porta tri-state (o in alta impedenza) 440
A.5.5 Circuito integrato 441

A.6 Bistabili e flip-flop 443
A.6.1 Bistabile asincrono 443
A.6.2 Bistabile sincrono 445
A.6.3 Flip-flop master-slave 448
A.6.4 Flip-flop edge-triggered 449
A.6.5 Flip-flop di tipo T 451
A.6.6 Flip-flop di tipo JK 451
A.6.7 Flip-flop con preset e clear 453

A.7 Registro seriale-parallelo (ser.-par. register) 453

A.8 Contatore binario (binary counter) 456

A.9 Decodificatore (decoder) 457

A.10 Multiplatore (multiplexer) 458

A.11 Componenti programmabili (FPGA) 461
Sommario 462
Note bibliografiche 463

Indice analitico I1

Eserciziario E1

Prefazione

Questo testo è frutto dell'adattamento e della traduzione in lingua italiana della sesta edizione di Computer Organization, uno dei manuali più conosciuti e diffusi internazionalmente sui fondamenti della struttura – o architettura – del calcolatore. Il libro ha una storia assai lunga (la prima edizione in inglese è del 1978) ed è il risultato di una revisione continua e di un arricchimento costante per quanto riguarda gli argomenti affrontati, che partono dai fondamenti del calcolatore per arrivare ad aspetti avanzati, oggetto di ricerca corrente. Ciò spiega anche l'estensione del titolo della sesta edizione: *Computer Organization and Embedded Systems*.

Scopo del testo è presentare i fondamenti della struttura del calcolatore, esponendo e illustrando concetti, nozioni ed esempi appropriati, in qualità e quantità, a un corso introduttivo sull'argomento. Il presente volume si propone di trattare il calcolatore dal punto di vista della struttura fisica (hardware) a blocchi funzionali, raffinandola progressivamente fino al livello di logica digitale, e dal punto di vista del linguaggio macchina, dando nozioni ed esempi sufficienti anche per imparare a programmare direttamente in linguaggio macchina. Insiste tuttavia sempre sugli aspetti concettuali e su quelli tecnologici e strutturali, cioè sull'architettura hardware del calcolatore, e non è pertanto né un manuale di progettazione di circuiti digitali né uno di programmazione.

Il testo è pensato per insegnamenti nei corsi di laurea di I livello in Informatica e in Ingegneria. Con riguardo alla struttura fisica del calcolatore, i curricula di Ingegneria spesso prevedono un distinto insegnamento di introduzione ai circuiti logici digitali, pertanto questo argomento è qui collocato nell'Appendice A, utile nei percorsi formativi che non ne prevedano la trattazione altrove. Con riguardo alla gestione software delle risorse hardware, il testo accenna in più punti alla relazione tra calcolatore e sistema operativo, ma non ne espone la teoria se non con qualche nozione molto generale. Infine, con riguardo alla programmazione in linguaggio macchina, si mantiene la scelta, innovativa nell'edizione originale rispetto alle precedenti edizioni, di proporre esempi di programmi in linguaggio assemblativo generico, indipendente da una particolare architettura commerciale, ma se ne illustra anche la traduzione nei linguaggi assemblativi specifici di quattro famiglie di processori commerciali: Altera NIOS II, Freescale ColdFire, ARM e Intel IA-32.

Un'analisi comparativa delle caratteristiche di queste architetture è offerta in approfondimenti applicativi disseminati nel testo. A tale fine, l'abbondante materiale di quattro appendici dell'edizione originale è stato selezionato e parzialmente riorganizzato.

Gli unici prerequisiti richiesti per un'agevole comprensione del testo sono la conoscenza, almeno approssimativa, di un linguaggio di programmazione quale C, C++, Java, Pascal o altri. Qualche nozione sul sistema operativo può essere utile, ma non è necessaria. Per il resto il libro è sostanzialmente autocontenuto e può essere affrontato da chiunque abbia un livello di istruzione secondario (liceo, istituto professionale e simili) tale da permettere di affrontare l'inizio dell'istruzione superiore in una facoltà di orientamento scientifico o tecnologico. Naturalmente un utile presupposto è un po' di interesse per il calcolatore, necessario per un proficuo studio.

Contenuti

Il corpo principale del libro si può approssimativamente suddividere in una sequenza di quattro parti: introduzione ai concetti di base, terminologia e storia, fondamenti di aritmetica binaria

e di logica della commutazione (Capitolo 1 e Appendice A); programmazione in linguaggio macchina (Capitoli 2-4); fondamenti di struttura hardware del calcolatore (Capitoli 5-8); introduzione ad argomenti avanzati (Capitoli 9-11). A corredo del corpo principale sono presentati numerosi *Approfondimenti applicativi*, integrati nei primi tre capitoli, e in coda due *Casi di studio*, rispettivamente sulla progettazione di sistemi embedded mediante FPGA e sull'architettura Intel IA-32. Più in dettaglio, il contenuto del libro si presenta come segue.

Il **Capitolo 1 Introduzione al calcolatore** espone i concetti fondamentali in modo discorsivo, anticipa la terminologia e presenta una breve storia del calcolatore. Illustra la classica composizione del calcolatore in unità funzionali di base: processore, memoria, rete di interconnessione, unità di ingresso e di uscita, e per ciascuna di esse tratteggia problematiche essenziali e soluzioni tecnologiche disponibili.

Tratta poi la rappresentazione binaria dell'informazione: di numeri, con le operazioni aritmetiche additive di base e rappresentazione in complemento a due, di caratteri, introducendone i codici di rappresentazione, e di dati multimediali, questi ultimi negli Approfondimenti applicativi aggiunti in questa edizione ai contenuti dell'edizione orginale. Infine, introduce aspetti di prestazione e ripercorre la storia del calcolatore, soffermandosi sull'invenzione delle tecnologie essenziali. È un capitolo autonomo e sostanzialmente autocontenuto, di inquadramento degli argomenti salienti trattati più a fondo nel resto del libro.

Il **Capitolo 2 Istruzioni macchina** illustra il concetto e la realizzazione dell'istruzione macchina: codifica di informazione e organizzazione di base della memoria; modi di indirizzamento; rappresentazione simbolica delle istruzioni macchina e linguaggi assemblativi; tassonomia delle istruzioni; direttive di assemblatore e macroistruzioni; gestione di sottoprogramma, cioè chiamata, rientro, passaggio dei parametri e gestione di pila; codifica binaria dell'istruzione ovvero il formato di memoria. Introduce questi concetti dapprima per architetture RISC, in cui trovano le realizzazioni più semplici, quindi per architetture CISC.

Termina con una esposizione dei principi di progetto CISC e RISC dell'insieme di istruzioni macchina del processore. Sei Approfondimenti applicativi mostrano come i concetti genericamente esposti si concretizzano in quattro architetture reali di processori commerciali, delle quali propone un'analisi comparativa: due RISC, Altera NIOS II e ARM, e due CISC, Freescale ColdFire e Intel IA-32. Esempi di programmi assemblativi generici presentati nel corpo principale del capitolo sono sviluppati nei linguaggi assemblativi nativi di queste architetture.

Il **Capitolo 3 Operazioni di ingresso e uscita** completa l'introduzione alla programmazione in linguaggio macchina con la trattazione dei trasferimenti di I/O dal punto di vista del programmatore, nelle due principali tecniche di I/O: controllo di programma e servizio di interruzioni, successivamente esteso al più generale concetto di servizio delle eccezioni. Tre Approfondimenti applicativi ampliano alle tecniche di I/O l'analisi comparativa di quattro architetture reali condotta nel Capitolo 2.

Il **Capitolo 4 Software** introduce il processo di assemblaggio, collegamento, caricamento e controllo di esecuzione dei programmi, illustrando i compiti dei rispettivi componenti software di supporto: compilatore, assemblatore, collegatore, caricatore, debugger. Si descrivono anche routine del sistema operativo che gestiscono l'esecuzione dei programmi di utente e le loro operazioni di I/O, inclusa la gestione delle interruzioni.

Il **Capitolo 5 Struttura di base del processore** illustra la struttura interna di base di un processore RISC, esplorandone il progetto a partire dalla sequenza di passi elementari necessari per il prelievo e l'esecuzione di differenti tipi di istruzione macchina.

Quindi sviluppa un'organizzazione hardware atta a realizzarli, attraverso l'interazione fra unità di prelievo, percorso dati e unità di controllo, considera i diversi requisiti posti da processori CISC e introduce sia il controllo cablato sia il controllo microprogrammato.

Il **Capitolo 6 Introduzione al pipelining** presenta i principi del pipelining e di molteplici unità di esecuzione nel progetto di processori ad alte prestazioni. Una variante in pipeline del processore RISC progettato nel Capitolo 5 illustra il pipelining. Il capitolo tratta la problematica dei diversi tipi di conflitti che possono penalizzarne il guadagno di prestazione e le principali tecniche escogitate per alleviarne gli effetti, le quali possono influenzare il progetto di compilatori. Esplora anche la relazione fra esecuzione in pipeline e progetto del repertorio di istruzioni macchina, e discute il progetto di processori superscalari.

L'hardware per l'I/O costituisce l'oggetto del **Capitolo 7 Sistema di ingresso e uscita** che discute le reti di interconnessione, con particolare riguardo alla struttura a bus, di cui spiega sia il funzionamento sincrono sia quello asincrono, le operazioni di lettura e di scrittura e l'arbitraggio. Tratta poi le porte di I/O e i loro modi operativi e circuiti logici. Conclude con una panoramica di vari standard d'interconnessione di sistema e di periferica, fra i quali USB e PCI Express.

Il **Capitolo 8 Sistema di memoria** illustra la struttura hardware e il funzionamento del sistema di memoria del calcolatore: discute la tecnologia microelettronica delle memorie a semiconduttori, il transistore e lo schema circuitale della cella di memoria, le memorie RAM, SRAM, DRAM, SDRAM, Rambus, ROM, PROM, EPROM, EEPROM, Flash ecc. Dà cenni su moduli di memoria e schemi di indirizzamento. Tratta la gerarchia di memoria, le memorie cache e relativi schemi di corrispondenza (diretta, associativa e associativa a gruppi) e di gestione, con analisi quantitativa delle prestazioni. Introduce i sistemi di memoria virtuale, la paginazione e l'unità di gestione della memoria, presentando tecniche di traduzione rapida degli indirizzi. Tratta infine le memorie di massa, poste alla base della gerarchia di memoria, con cenni alle tecnologie di disco magnetico, dischi ottici (CD, DVD) e nastri magnetici.

Il **Capitolo 9 Aritmetica** esplora la realizzazione hardware dell'unità aritmetica-logica (ALU) del calcolatore. Descrive il progetto logico di circuiti per l'esecuzione di addizione, sottrazione, moltiplicazione e divisione in virgola fissa, operanti nella rappresentazione in complemento a due. Spiega tecniche di accelerazione della loro esecuzione, in particolare l'addizione con anticipo del riporto e la moltiplicazione con ricodifica di Booth, ricodifica bit-pair e uso di addizionatori con salvataggio del riporto. Conclude illustrando la rappresentazione dei numeri frazionari in virgola mobile e l'esecuzione di operazioni su di essi, in conformità allo standard IEEE 754.

Oggigiorno i processori, dedicati, che si trovano in sistemi embedded sono di gran lunga più numerosi di quelli impiegati in calcolatori di uso generale. Il **Capitolo 10 Sistemi embedded** presenta aspetti di base dell'integrazione di tali sistemi, le interconnessioni di componenti e il loro funzionamento in tempo reale. Discute e illustra l'uso di microcontrollori in vari esempi di applicazioni embedded.

Il **Capitolo 11 Elaborazione parallela e prestazioni** estende la trattazione dell'esecuzione parallela, introdotta nel Capitolo 6 al livello di istruzione, focalizzando il parallelismo al livello di programma e di sistema. Presenta dapprima il multithreading hardware e l'elaborazione vettoriale per il miglioramento della prestazione di un processore singolo. Quindi descrive i multiprocessori a memoria condivisa e l'associata problematica della coerenza di cache. Tratta diversi tipi di reti di interconnessione per multiprocessori e accenna ai multicalcolatori, o reti di calcolatori, a scambio di messaggi.

Il **Caso di studio C1 Progetto di un sistema su un chip** presenta un'applicazione dei concetti esposti nel Capitolo 10 a un esempio di realizzazione di un sistema-su-chip (SoC), in cui si integrano su un singolo chip le funzionalità di elaborazione, memoria, I/O e temporizzazione necessarie per soddisfare i requisiti specifici di una data applicazione.

Un esempio elaborato mostra l'uso di FPGA e di moderni strumenti di progetto in tale contesto.

Il **Caso di studio C2 Architettura Intel IA-32** espone l'architettura dell'insieme di istruzioni dei processori IA-32, dandone estensivamente sintassi, interpretazione e vincoli, e aggiungendo svariate informazioni utili come codifica numerica, particolarità, istruzioni aritmetiche in virgola mobile (delle quali si fa un raffronto con le analoghe istruzioni Freescale ColdFire), istruzioni vettoriali, istruzioni speciali e privilegiate ecc.

L'**Appendice A Elementi di logica della commutazione** richiama i fondamenti di logica Booleana, sui quali si basa il resto che consta di: cenni alla tecnologia microelettronica digitale e modello logico del transistore come interruttore; porta logica, rete combinatoria, forma canonica e forma minima a due livelli; rete sequenziale, bistabile sincrono e asincrono; blocchi funzionali logici di base quali multiplatore, decodificatore, contatore, registri ecc.

Caratteristiche della terza edizione

Il libro è pensato per un corso di base sulla struttura del calcolatore, compresa un'eventuale attività di laboratorio di programmazione in linguaggio macchina. È comunque adatto anche per la lettura individuale. Le parti che lo compongono hanno una certa autonomia e sono facilmente separabili. Tolta l'introduzione, evidentemente essenziale, e i presupposti di logica digitale, i quali si possono studiare o meno a seconda delle conoscenze pregresse, il nucleo fondamentale del libro (Capitoli 2-8) presenta il calcolatore illustrandone il linguaggio macchina, i sottosistemi principali e la struttura interna del processore. Quest'ultimo argomento è trattato dapprima presentando un modello semplificato, adatto essenzialmente per eseguire l'istruzione macchina in più cicli di clock, senza parallelismo tra istruzioni. Tale modello, incentrato sullo scomporre il processore in unità di prelievo, di calcolo e di controllo, almeno come concetto è basilare, e la sua organizzazione a stadi ben si presta al suo successivo raffinamento in modelli di supporto al parallelismo delle istruzioni, con tecniche quali il pipelining e l'esecuzione superscalare. Rispetto alla precedente edizione italiana, il materiale incluso è stato esteso, aggiornato e riorganizzato per essere più adatto alle necessità dei corsi. I tre capitoli finali (Capitoli 9-11) offrono qualche spunto su argomenti avanzati, principalmente circuiti veloci per l'aritmetica, lo standard IEEE 754 per l'aritmetica in virgola mobile, un'introduzione ai sistemi embedded, il parallelismo al livello di programma e di sistema e il guadagno di prestazione che ne deriva.

Tutti i capitoli e l'appendice presentano la materia in modo abbastanza preciso e completo da permettere di impostare e risolvere problemi, esemplificati e proposti nell'Eserciziario incluso nel libro (altra novità rispetto alla precedente edizione italiana).

Una particolarità del libro, intrinseca al testo originale, è che istruzioni e linguaggio macchina dapprima vengono trattati in generale (nel corpo dei Capitoli 2-3), mediante una notazione per le istruzioni, i simboli, le operazioni e le direttive di assemblatore, che è genericamente riferibile a qualunque processore o quasi, e poi vengono raffinati in dettaglio sulle architetture di processori commerciali. Una differenza tra l'edizione originale e il presente adattamento è che mentre nella prima la trattazione delle quattro architetture commerciali, con relativi esempi di programmazione, è presentata in altrettante appendici separate, qui si è preferito integrarla nei Capitoli 2-3 mediante una serie di Approfondimenti applicativi volti a metterne in luce similarità e differenze. A ulteriore corredo applicativo si sono aggiunti i Caso di studio C1 e C2: in particolare quest'ultimo è più completo nei dettagli sull'architettura Intel IA-32 di quanto non fosse possibile esserlo nei primi suddetti capitoli.

Con riguardo alle architetture CISC, si è confermata la scelta di due esempi didattici molto significativi e diffusi: Freescale ColdFire, processore semplice, lineare, studiabile in un tempo ragionevole, e Intel IA-32, più complesso ma studiabile comunque riutilizzando molti degli stessi concetti, e diffusissimo. L'edizione italiana precedente, pubblicata nel 2007, usa come modelli didattici i processori M68000 e IA-32, dei quali illustra sinteticamente alcune famiglie, ma comprensibilmente manca di qualche argomento più avanzato, quale i sistemi embedded e le architetture RISC.

In questo si trova la differenza forse più vistosa rispetto alla presente edizione, che oltre all'aggiornamento costituito dal trattare la famiglia di processori Freescale "successore" di Motorola, estende la trattazione alle architetture RISC dei processori Altera NIOS II e ARM, di largo impiego nella progettazione di sistemi embedded.

Ci auguriamo di avere organizzato e tradotto il presente testo in modo semplice e comprensibile. La terminologia tecnica inglese è conservata con cura tra parentesi, dove è tradotta, e nei termini più comuni è conservata o messa in parallelo a quella italiana. Gli esempi di programma e i calcoli sono quelli dell'edizione originale, benché talvolta sintetizzati o riordinati, e naturalmente tradotti dove possibile.

Ringraziamenti

Desideriamo esprimere il nostro più vivo ringraziamento a Filippo Aroffo dello staff editoriale McGraw-Hill per il prezioso, infaticabile e prolungato supporto alla produzione di questo libro. Un grato riconoscimento è anche dovuto al team di Fotocompos, per la sua collaborazione in tutte le fasi della lavorazione.

Il curatore desidera inoltre ringraziare gli studenti Luca Morreale, Fabio D'Urso, Simone Fazzio e Sebastiano Puglisi, dell'edizione 2011-2012 dell'insegnamento di Architettura degli elaboratori al Dipartimento di matematica e informatica dell'Università di Catania, per avere segnalato e corretto errori in una versione preliminare dei materiali didattici impiegati per la preparazione del libro, così contribuendo a migliorarne la qualità.

Giuseppe Scollo
Emiliano Tramontana

Autori

Cari Hamacher è Professor Emeritus presso il Department of Electrical and Computer Engineering della Queen's University, Kingston, Canada.
La sua attività di ricerca si svolge nell'ambito dei multiprocessori e multi computer, con particolare attenzione alle loro reti di interconnessione.

Zvonko Vranesic è Professor Emeritus presso il Department of Electrical and Computer Engineering della University of Toronto, Canada. Si occupa inoltre di ricerca e sviluppo presso l'Altera Toronto Technology Center.
I suoi attuali interessi di ricerca includono l'architettura dei calcolatori e la field-programmable VLSI tecnology.

Safwat Zaky è Professor Emeritus presso il Department of Electrical and Computer Engineering della University of Toronto, Canada. È inoltre Fellow della Canadian Academy of Engineering.
I suoi attuali interessi di ricerca includono l'architettura dei calcolatori, il digital-circuit design e la compatibilità elettromagnetica.

Naraig Manjikian è Associate Professor presso il Department of Electrical and Computer Engineering della Queen's University, Kingston, Canada.
La sua attività di ricerca si svolge nell'ambito dell'architettura dei calcolatori, dei multiprocessor systems, della field-programmable VLSI technology e delle applicazioni dell'elaborazione parallela.

Il Curatore

Giuseppe Scollo è professore associato di Informatica presso il Dipartimento di Matematica e Informatica dell'Università di Catania, dove tiene l'insegnamento di Architettura degli elaboratori.
I suoi attuali interessi di ricerca sono rivolti a sistemi dinamici simbolici, algoritmi paralleli di aritmetica computazionale e problemi di combinatorica, mentre nell'arco della carriera vertono sull'uso di metodi matematici nel progetto di sistemi informatici e nell'elaborazione di linguaggi, con speciale attenzione a fondamenti e linguaggi di specifica algebrica, e ad applicazioni di interesse industriale, quali il progetto di sistemi logici e di linguaggi di specifica formale, il progetto e la descrizione formale di servizi e protocolli per reti di calcolatori, il progetto di sistemi di acquisizione dati. Ha pubblicato oltre ottanta lavori nel settore.

Ringraziamenti dell'Editore

L'Editore ringrazia i docenti che hanno partecipato alla review del testo e che, con le loro preziose indicazioni, hanno contribuito alla realizzazione della terza edizione di *Introduzione all'architettura dei calcolatori*:

Roberto Canonico, *Università degli Studi di Napoli "Federico II"*

Marco Cesati, *Università degli Studi di Roma "Tor Vergata"*

Luigi Coppolino, *Università degli Studi di Napoli "Parthenope"*

Giovanni Danese, *Università degli Studi di Pavia*

Antonino Mazzeo, *Università degli Studi di Napoli "Federico II"*

Sebastiano Pizzutilo, *Università degli Studi di Bari "Aldo Moro"*

Luigi Romano, *Università degli Studi di Napoli "Parthenope"*

L'Editore ringrazia inoltre i docenti che parteciparono alla review della seconda edizione di *Introduzione all'architettura dei calcolatori*:

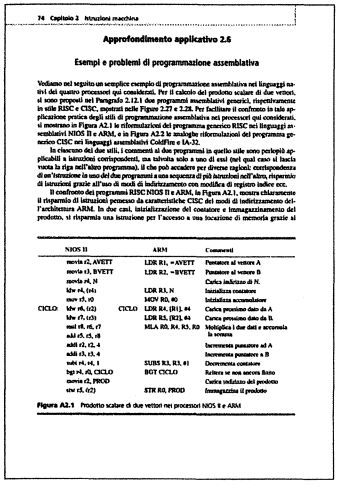
Rocco Aversa, *Seconda Università degli Studi di Napoli*

David Macii, *Università degli Studi di Trento*

Antonino Mazzeo, *Università degli Studi di Napoli "Federico II"*

Davide Quaglia, *Università degli Studi di Verona*

Guida alla lettura



74 Capitolo 2 Istruzioni macchina

Approfondimento applicativo 2.6

Esempi e problemi di programmazione assemblativa

Vediamo nel seguito un semplice esempio di programmazione assemblativa nei linguaggi nativi dei quattro processori qui considerati. Per il calcolo del prodotto scalare di due vettori, si sono proposti nel Paragrafo 2.12.1 due programmi assemblativi generici, rispettivamente in stile RISC e CISC, mostrati nelle Figure 2.27 e 2.28. Per facilitare il confronto in tale applicazione pratica degli stili di programmazione assemblativa nei processori qui considerati, si mostrano in Figura A2.1 le riformulazioni del programma generico RISC nei linguaggi assemblativi NIOS II e ARM, e in Figura A2.2 le analoghe riformulazioni del programma generico CISC nei linguaggi assemblativi ColdFire e IA-32.

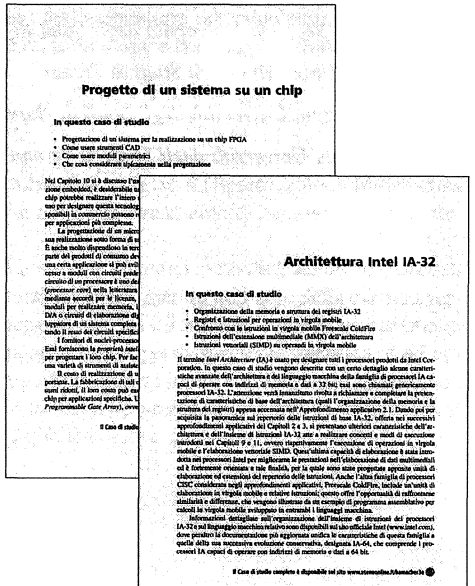
In ciascuno dei due stili, i commenti ai due programmi in quello stile sono perlopiù applicabili a istruzioni corrispondenti, ma talvolta solo a uno di essi (nel qual caso si lascia vuota la riga nell'altro programma), il che può accadere per diverse ragioni: corrispondenza di un'istruzione in uno dei due programmi a una sequenza di più istruzioni nell'altro, risparmio di istruzioni grazie all'uso di modi di indirizzamento con modifica di registro indice ecc.

Il confronto dei programmi RISC NIOS II e ARM, in Figura A2.1, mostra chiaramente il risparmio di istruzioni peneuso da caratteristiche CISC dei modi di indirizzamento dell'architettura ARM. In due casi, inizializzazione del contatore e immagazzinamento del prodotto, si risparmia una istruzione per l'accesso a una locazione di memoria grazie ai

	NIOS II		ARM	Commenti
	movia r2, AVETT		LDR R1, =AVETT	Puntatore al vettore A
	movia r3, BVETT		LDR R2, =BVETT	Puntatore al vettore B
	movia r4, N			Carica indirizzo di N.
	ldw r4, (r4)		LDR R3, N	Inizializza contatore
	mov r5, r0		MOV R0, #0	Inizializza accumulatore
CICLO:	ldw r6, (r2)	CICLO:	LDR R4, [R1], #4	Carica prossimo dato da A
	ldw r7, (r3)		LDR R5, [R2], #4	Carica prossimo dato da B
	mul r8, r6, r7		MLA R0, R4, R5, R0	Moltiplica i due dati e accumula la somma
	add r5, r5, r8			
	addi r2, r2, 4			Incrementa puntatore ad A
	addi r3, r3, 4			Incrementa puntatore a B
	subi r4, r4, 1		SUBS R3, R3, #1	Decrementa contatore
	bgt r4, r0, CICLO		BGT CICLO	Ritorna se non ancora finito
	movia r2, PROD			Carica indirizzo del prodotto
	stw r5, (r2)		STR R0, PROD	Immagazzina il prodotto

Figura A2.1 Prodotto scalare di due vettori nei processori NIOS II e ARM

Le schede *Approfondimento applicativo* integrano la teoria con la trattazione di quattro architetture commerciali: Altera NIOS II, Freescale ColdFire, ARM e Intel IA-32.



Progetto di un sistema su un chip

In questo caso di studio

- Progettazione di un sistema per la realizzazione su un chip FPGA
- Come usare strumenti CAD
- Come usare moduli parametrici
- Che cosa considerare tipicamente nella progettazione

Architettura Intel IA-32

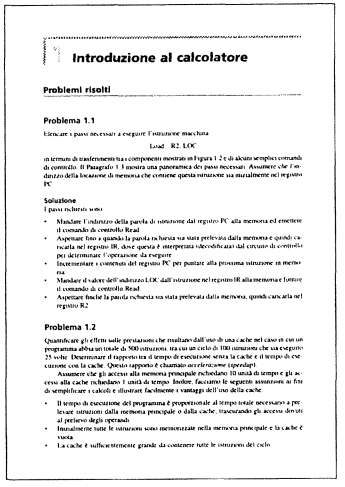
In questo caso di studio

- Organizzazione della memoria e assetto dei registri IA-32
- Registri e istruzioni per operazioni in virgola mobile
- Confronto tra le istruzioni in virgola mobile Freescale ColdFire
- Istruzioni dell'estensione multimediale (MMX) dell'architettura
- Istruzioni vettoriali (SIMD) su operandi in virgola mobile

Il *Caso di studio C1 Progetto di un sistema su un chip* presenta un'applicazione dei concetti presentati sui sistemi embedded, mentre il *Caso di studio C2 Architettura Intel IA-32* descrive nel dettaglio quella che finora è stata l'architettura più comune nei personal computer.

L'*Eserciziario* incluso nel testo comprende oltre 250 tra problemi risolti e problemi proposti per completare la preparazione in vista dell'esame.

I *Problemi risolti* aiutano lo studente ad apprendere come risolvere i problemi che potrebbe trovare all'esame.



1 **Introduzione al calcolatore**

Problemi risolti

Problema 1.1

Elencare i passi necessari a eseguire l'istruzione macchina

Load R2, LOC

in termini di trasferimenti tra componenti mostrati in Figura 1.2 e di alcuni semplici comandi di controllo. Il Paragrafo 1.3 presenta una panoramica dei passi necessari. Assumere che l'indirizzo della locazione di memoria che contiene questa istruzione sia inizialmente nel registro PC.

Soluzione

I passi richiesti sono:

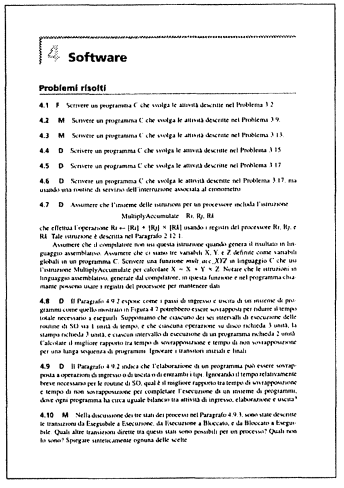
- Mandare l'indirizzo della parola di istruzione dal registro PC alla memoria ed emettere il comando di controllo Read
- Aspettare fino a quando la parola richiesta sia stata prelevata dalla memoria e quindi caricarla nel registro IR, dove questa è interpretata (decodificata) dal circuito di controllo per determinare l'operazione da eseguire
- Incrementare i contenuti del registro PC per puntare alla prossima istruzione in memoria
- Mandare il valore dell'indirizzo LOC dall'istruzione nel registro IR alla memoria e fornire il comando di controllo Read
- Aspettare finché la parola richiesta sia stata prelevata dalla memoria, quindi caricarla nel registro R2

Problema 1.2

Quantificare gli effetti sulle prestazioni che risultano dall'uso di una cache nel caso in cui un programma abbia un totale di 500 istruzioni, tra cui un ciclo di 100 istruzioni che sia eseguito 25 volte. Determinare il rapporto tra il tempo di esecuzione senza la cache e il tempo di esecuzione con la cache. Questo rapporto è chiamato *accelerazione (speedup)*.

Assumere che gli accessi alla memoria principale richiedano 10 unità di tempo e gli accessi alla cache richiedano 1 unità di tempo. Inoltre, facciamo le seguenti assunzioni, al fine di semplificare i calcoli e illustrare facilmente i vantaggi dell'uso della cache:

- Il tempo di esecuzione del programma è proporzionale al tempo totale necessario a prelevare istruzioni dalla memoria principale o dalla cache, trascurando gli accessi dovuti al prelievo degli operandi.
- Inizialmente tutte le istruzioni sono memorizzate nella memoria principale e la cache è vuota.
- La cache è sufficientemente grande da contenere tutte le istruzioni del ciclo



4 **Software**

Problemi risolti

4.1 **F** Scrivere un programma C che svolga le attività descritte nel Problema 3.2

4.2 **M** Scrivere un programma C che svolga le attività descritte nel Problema 3.9

4.3 **M** Scrivere un programma C che svolga le attività descritte nel Problema 3.13

4.4 **D** Scrivere un programma C che svolga le attività descritte nel Problema 3.15

4.5 **D** Scrivere un programma C che svolga le attività descritte nel Problema 3.17

4.6 **D** Scrivere un programma C che svolga le attività descritte nel Problema 3.17, ma usando una routine di servizio dell'interrupt associata al cronometro

4.7 **D** Assumere che l'insieme delle istruzioni per un processore includa l'istruzione

MultiplyAccumulate Rr, Rj, Rk

che effettua l'operazione Rr ← [Rr] + [Rj] × [Rk] usando i registri del processore Rr, Rj, e Rk. Tale istruzione è descritta nel Paragrafo 2.12.1.

Assumere che il compilatore non usi questa istruzione quando genera il risultato in linguaggio assemblativo. Assumere che ci siano tre variabili X, Y, e Z definite come variabili globali in un programma C. Scrivere una funzione *mult acc*, XYZ in linguaggio C che usi l'istruzione MultiplyAccumulate per calcolare X ← X + Y × Z. Notare che le istruzioni in linguaggio assemblativo generate dal compilatore, in questa funzione e nel programma chiamante possono usare i registri dei processori per mantenere dati.

4.8 **D** Il Paragrafo 4.9.2 espose come i passi di ingresso e uscita di un insieme di programmi come quello mostrato in Figura 4.7 potrebbero essere sovrapposti per ridurre il tempo totale necessario a eseguirli. Supponiamo che ciascuno dei sei intervalli di esecuzione delle routine di SO sia 1 unità di tempo, e che ciascuna operazione su disco richieda 3 unità. La stampa richiede 3 unità, e ciascun intervallo di esecuzione di un programma richiede 2 unità. Calcolare il migliore rapporto tra tempo di sovrapposizione e tempo di non sovrapposizione per una lunga sequenza di programmi. Ignorare i transitori iniziali e finali.

4.9 **D** Il Paragrafo 4.9.2 indica che l'elaborazione di un programma può essere sovrapposta a operazioni di ingresso o di uscita di ciascuno tipo. Ignorando il tempo relativamente breve necessario per le routine di SO, qual è il migliore rapporto tra tempo di sovrapposizione e tempo di non sovrapposizione per completare l'esecuzione di un insieme di programmi, dove ogni programma ha circa uguale bilancio tra attività di ingresso, elaborazione e uscita?

4.10 **M** Nella discussione dei tre stati dei processi nel Paragrafo 4.9.3, sono state descritte le transizioni da Eseguibile a Esecuzione, da Esecuzione a Bloccato, e da Bloccato a Eseguibile. Quali altre transizioni direste tra questo stato sono possibili per un processo? Quali non lo sono? Spiegare sinteticamente ognuna delle scelte

I *Problemi proposti* sono classificati secondo la difficoltà, identificata dalle lettere **F** (facile), **M** (medio) e **D** (difficile):

F La soluzione può essere ottenuta in pochi minuti applicando pedissequamente i concetti e le metodologie presentati nel testo.

M È necessario far uso dei concetti e dei metodi presentati nel libro in modi che non sono necessariamente quelli esemplificati nel testo. In alcuni casi, la soluzione può derivare dallo schema generale di uno degli esempi presentati, ma è necessario uno sviluppo più articolato rispetto ai problemi **F**.

D È necessario un'ulteriore rielaborazione dei concetti del testo per poter risolvere questi problemi. Per esempio, se la soluzione richiede di scrivere un programma, l'algoritmo da tradurre in codice può essere sensibilmente diverso da quelli presentati nel testo.

1 Introduzione al calcolatore

In questo capitolo

- Differenti tipi di calcolatori
- Struttura di base di un calcolatore e sue operazioni
- Istruzioni macchina e loro esecuzione
- Rappresentazioni di numeri e caratteri
- Addizioni e sottrazioni di numeri binari
- Principali punti riguardanti le prestazioni di un calcolatore
- Una breve storia dello sviluppo dei calcolatori

Il libro descrive e spiega come è fatto il *calcolatore* (*computer*) elettronico,[1] illustrando in primo luogo funzioni e struttura delle unità fondamentali componenti il calcolatore, in particolare il processore e la memoria, e in secondo luogo funzioni e in parte struttura delle unità che mettono il calcolatore in comunicazione con il mondo esterno, ovvero le periferiche.

Molte delle spiegazioni contenute nel libro sono dedicate al modo di funzionare dei dispositivi – il cosiddetto *hardware* – costituenti il calcolatore e a come sono organizzati – la *struttura* o *architettura* del calcolatore (*computer structure* o *architecture*). Per esempio, i dispositivi del calcolatore comprendono svariati tipi di circuiti elettronici, schermo video, unità di memoria di tipo magnetico o ottico, apparati elettromeccanici e unità di comunicazione e altro ancora. Tuttavia l'organizzazione dei numerosi componenti funzionali contenuti nel calcolatore ruota in modo essenziale attorno alla specifica del linguaggio macchina da usare per programmarlo e di conseguenza alla struttura delle unità funzionali che eseguono le istruzioni del linguaggio, primo tra tutte il processore. Parecchio spazio è dunque dedicato anche a presentare e spiegare la struttura, il significato e l'uso del linguaggio macchina caratteristico del processore.

Benché non sia lo scopo principale del libro, qui si discutono anche svariati aspetti di organizzazione e programmazione del sistema operativo, perché studiando e progettando i vari componenti del calcolatore è importante tenerne in vista sia gli aspetti tecnologici, o *hardware*, sia quelli programmativi, o *software*. Oggi il calcolatore elettronico è diventato un sistema troppo complesso perché ci si possa permettere di ignorarne alcuni aspetti quando si voglia comprenderlo bene.

1.1 Famiglie di calcolatori

Le strutture dei calcolatori si differenziano fortemente per dimensione, costo, potenza di calcolo e scopo per cui sono progettate. I calcolatori moderni possono essere suddivisi in quattro categorie generali.

---

[1] In tutto il libro si scriverà e dirà *calcolatore*, non *computer*.

- I *calcolatori embedded* (*incorporati* o *inclusi*) sono integrati in un dispositivo o sistema più grande ai fini di monitorare o controllare automaticamente un processo fisico o un ambiente. Essi sono usati per uno scopo specifico anziché per elaborazioni generali. Tipiche applicazioni includono automatismi per l'industria e per la casa, dispositivi domestici, prodotti per telecomunicazioni e veicoli. Gli utenti possono persino non sapere del ruolo che i calcolatori hanno in tali sistemi.
- I *calcolatori personali* (*personal computer*) hanno raggiunto largo uso nelle case, nelle istituzioni scolastiche e in uffici aziendali e di progettazione, principalmente dedicati all'uso individuale. Essi supportano una varietà di applicazioni, di cui alcuni esempi sono elaborazioni, preparazione documenti, progettazione assistita, intrattenimento audio e video, comunicazioni interpersonali e navigazione Internet. Varie classificazioni sono usate per i calcolatori personali. I *calcolatori da tavolo* (*desktop computer*) soddisfano bisogni generali e sono adatti al tipico ambiente di lavoro di una persona. I *calcolatori portatili* (*portable computer* o *lap-top computer*) rappresentano invece la versione compatta e leggera del calcolatore da tavolo. Possono essere alimentati da una batteria, per poter operare in condizioni di mobilità. Le *stazioni di lavoro* (*workstation*) si differenziano dal calcolatore da tavolo perché sono dotate di schermo grafico ad alta risoluzione e di potenza di calcolo sensibilmente maggiore. Esse vengono usate spesso in applicazioni di ingegneria, scientifiche e soprattutto in quelle che richiedono attività di disegno interattivo.
- I *server* e i *sistemi enterprise* (*calcolatori aziendali*) sono grandi calcolatori pensati per essere condivisi da un numero potenzialmente alto di utenti che accedono a essi da qualche tipo di calcolatore personale tramite una rete pubblica o privata. Tali calcolatori possono ospitare grandi basi di dati (*database*) e fornire capacità di elaborazione di informazioni per un'agenzia governativa o per organizzazioni commerciali.
- I *supercalcolatori* e i *calcolatori Grid* (o *griglia computazionale*) di solito offrono prestazioni più elevate. Essi sono i più costosi e fisicamente la più grande categoria di calcolatori. I supercalcolatori sono usati per le elaborazioni più difficili richieste nelle previsioni meteorologiche, per progettazioni e simulazioni di ingegneria e per lavori scientifici. Essi hanno un costo elevato. I calcolatori Grid forniscono un'alternativa più economica. Essi collegano insieme, tramite una rete ad alta velocità chiamata *griglia*, un gran numero di calcolatori personali e di unità a disco fisicamente distribuiti. Questo insieme è gestito come un'unica risorsa di elaborazione. Distribuendo equamente il carico di elaborazione sulla griglia è possibile ottenere alte prestazioni su applicazioni grandi che vanno dalle elaborazioni numeriche alla ricerca di informazioni.

C'è una tendenza emergente all'accesso a supporti di elaborazione conosciuti come *calcolatori cloud* (*cloud computing*). Gli utenti di calcolatori personali accedono a risorse di elaborazione e immagazzinamento di dati ampiamente distribuite, ai fini di portare a termine esigenze di elaborazione individuali e indipendenti. Internet fornisce la necessaria infrastruttura di comunicazione. I fornitori di hardware e software del cloud operano facendo pagare gli utenti in base all'uso che fanno dei servizi cloud.

1.2 Componenti funzionali

Il calcolatore è costituito da cinque parti, o unità funzionali principali, indipendenti l'una dall'altra: ingresso, memoria, per calcoli aritmetici e logici (in breve aritmetica-logica), uscita, controllo; sono tutte mostrate in Figura 1.1. L'unità di ingresso acquisisce informazioni in forma codificata da operatori umani, da dispositivi elettromeccanici, come la tastiera, o anche da altri calcolatori collegati tramite linee di comunicazione digitali, per esempio via rete. Le

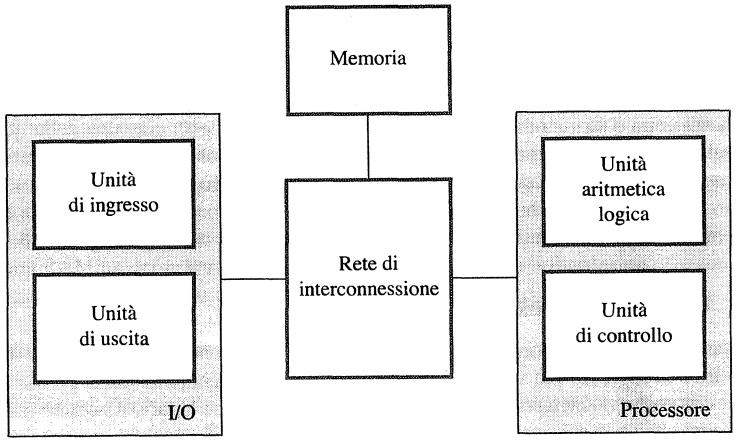


Figura 1.1 Unità funzionali di un calcolatore.

informazioni così ottenute vengono scritte nella memoria del calcolatore oppure usate subito da parte dell'unità aritmetica-logica per svolgere i calcoli desiderati, secondo una serie di passi o istruzioni determinata dal programma presente in memoria, e infine i risultati del calcolo vengono mandati al mondo esterno tramite l'unità di uscita; l'unità di controllo si incarica di coordinare il funzionamento di tutte le altre unità.

Una rete di interconnessione fornisce i mezzi alle unità funzionali per scambiare informazioni e coordinare le loro azioni. Successivi capitoli forniranno altri dettagli sulle singole unità e le loro interconnessioni. Si possono raggruppare le unità funzionali in macro-unità: l'insieme di unità aritmetica-logica e dei circuiti di controllo principali costituisce il *processore* (*processor*), mentre l'insieme delle unità di ingresso e uscita forma il sistema di I/O (*input/output system*).

Ora bisogna esaminare in dettaglio natura e flusso delle informazioni manipolate dal calcolatore; per farlo conviene categorizzarle in *istruzioni* e *dati*. Le *istruzioni macchina* (*machine instruction*), o semplicemente *istruzioni*, sono comandi dati al calcolatore in forma esplicita e da esso interpretabili direttamente; a grandi linee esse hanno le funzioni seguenti:

- guidare il flusso di informazioni internamente al calcolatore;
- guidare il trasferimento di informazioni tra calcolatore e sistema di I/O;
- specificare le operazioni logiche e aritmetiche da eseguire.

La lista delle istruzioni macchina che complessivamente svolgono un determinato compito costituisce il *programma* (*program*). Quando il programma è in esecuzione si trova sempre in memoria: il processore preleva in modo sequenziale le istruzioni elencate nel programma e le esegue una alla volta. Pertanto il funzionamento del calcolatore è interamente determinato dal programma correntemente in memoria, salvo la facoltà da parte di un operatore umano oppure dispositivo di I/O (come un sensore) di mandare al calcolatore un segnale di *interruzione* (*interrupt*) per influenzarne in parte il comportamento oppure addirittura forzarlo, con più o meno autorità, a svolgere una funzione o servizio determinati.

In genere i *dati* (*data*) sono costituiti da numeri (interi o di altro tipo) e caratteri che servono alle istruzioni macchina come operandi su cui lavorare; tuttavia il termine *dato* si applica

spesso a qualsiasi elemento di informazione rappresentato in forma digitale. Anche i dati sono immagazzinati in memoria.

Le informazioni manipolate dal calcolatore vanno rappresentate o codificate in formato adatto allo scopo. La maggior parte dei calcolatori odierni è costituita da circuiti con due stati stabili. Pertanto un numero, un carattere o anche un'istruzione macchina viene codificato come successione o stringa di *cifre binarie* o *bit* (da *binary digit*), ciascuna delle quali può assumere due soli valori, denotati con 0 e 1. Di solito si rappresenta il numero intero in *notazione posizionale binaria*, come si vedrà meglio nel Paragrafo 1.4. Anche il carattere alfanumerico (lettera, cifra, simbolo di interpunzione o simbolo speciale) viene solitamente codificato in modo binario tramite una stringa di bit, come discusso nel Paragrafo 1.5.

1.2.1 Unità di ingresso

Il calcolatore acquisisce informazioni dal mondo esterno servendosi delle unità di ingresso. Il dispositivo di ingresso più visibile all'utente è la *tastiera*: ogniqualvolta si preme un tasto la lettera o il simbolo corrispondente viene convertito nel codice binario associato e il codice così determinato viene trasmesso al calcolatore.

Beninteso esistono molti altri tipi di dispositivo di ingresso da utente: per esempio *cloche* (o anche *joystick*), che possono assumere svariate forme, *sfere direzionali* (*trackball*), *puntatori* (*mouse*) e altri ancora; si usano spesso in combinazione con schermo video dotato di capacità grafiche per formare insieme il *terminale*. Inoltre si può aggiungere al tutto il microfono, per convertire il suono in segnale digitale tramite la tecnica di campionamento per poi avviarlo all'elaborazione o tenerlo in memoria, e la telecamera, che ha funzioni analoghe ma per le immagini fisse o in movimento. Anche le infrastrutture per la comunicazione digitale, come Internet, possono fornire input a un calcolatore da altri calcolatori e server di basi di dati.

1.2.2 Unità di memoria

La memoria del calcolatore ha la funzione di contenere l'insieme dei programmi e i dati. Esistono due famiglie: memoria *centrale*, o *primaria*, e memoria *di massa*, o *secondaria*.

Memoria centrale

La *memoria primaria*, detta anche *memoria principale*, è una memoria ad accesso veloce che opera alla velocità dei circuiti elettronici. Mentre si trova in esecuzione il programma deve risiedere proprio nella memoria centrale. Secondo la tecnologia microelettronica corrente la *memoria centrale* (*main memory* o *primary memory*) è formata da un numero molto grande di celle elementari realizzate tramite *transistori a semiconduttori*, dove la singola cella è capace di contenere l'informazione di un bit. Di rado la cella da un bit viene letta o scritta individualmente, piuttosto si usa un gruppo formato da una quantità fissata di celle, chiamato *parola di memoria* (*memory word*), o semplicemente *parola*, e talvolta anche *locazione di memoria* (*memory location*). La memoria centrale è strutturata in modo da permettere il trasferimento (in lettura o scrittura) di un'intera parola di memoria tramite una sola operazione.

Spesso il numero di bit costituenti la parola viene chiamato *dimensione* o *lunghezza della parola*; è un parametro importante del calcolatore considerato: il valore tipico di n varia tra 8 e 64 bit e quasi sempre n assume come valore una potenza di 2 (per esempio 8, 16, 32 o 64).

Ogni parola di memoria è identificata in modo univoco da un numero, chiamato *indirizzo* (*address*) della parola; gli indirizzi sono assegnati alle parole andando in successione. Per lavorare su una determinata parola di memoria, collocata a un dato indirizzo, bisogna specificarne l'indirizzo e attivare un comando di controllo che dia inizio all'operazione. Ci sono due operazioni di memoria fondamentali: la *lettura* di parola, ovvero il recupero da memoria del valore della parola, e la *scrittura* di parola, ovvero la modifica in memoria del contenuto della parola, sostituendolo con un nuovo valore.

Si è già detto che il programma in esecuzione deve trovarsi in memoria centrale, perché le istruzioni e i dati su cui esse lavorano vengono letti da memoria o scritti in memoria sotto la direzione del processore. Pertanto è essenziale potere individuare e raggiungere qualsiasi parola di memoria il più rapidamente possibile. La tecnologia microelettronica che permette di farlo in un intervallo di tempo fissato e costante, indipendente dall'indirizzo della parola, si chiama *memoria ad accesso casuale* o RAM (*random access memory*); l'intervallo di tempo necessario per individuare e raggiungere la parola si chiama *tempo di accesso alla memoria* (o *tempo di lavoro della memoria*), il quale è fisso e indipendente dall'indirizzo: i tempi tipici delle unità RAM moderne variano da pochi nanosecondi (pochi miliardesimi di secondo) fino a cento nanosecondi (ovvero un decimilionesimo di secondo) o poco più.

Memoria cache Normalmente la memoria centrale del calcolatore è strutturata sotto forma di *gerarchia di memoria*, costituita da tre o quattro *strati* o *livelli* di unità di memoria RAM a semiconduttori, ciascuna delle quali è caratterizzata da capacità e velocità proprie. Gli strati contenenti unità di memoria RAM piccole ma veloci costituiscono la cosiddetta *memoria cache*: essa è strettamente legata al processore, tanto da essere spesso ospitata sullo stesso circuito integrato di quest'ultimo allo scopo di raggiungere la massima velocità di lavoro.

All'inizio dell'esecuzione di un programma, la cache è vuota. Tutte le istruzioni del programma e i dati necessari sono immagazzinati nella memoria principale. Mano a mano che l'esecuzione va avanti, le istruzioni sono portate al chip del processore e una copia di ciascuna è inserita nella cache. Quando l'esecuzione di un'istruzione richiede dati presenti nella memoria principale, i dati sono prelevati e una copia è inserita nella cache.

Ora, supponiamo che un numero di istruzioni siano eseguite ripetutamente, come avviene in un ciclo di un programma. Se queste istruzioni sono disponibili nella cache, esse possono essere prelevate velocemente durante il periodo di uso frequente. Analogamente, se le stesse locazioni dei dati vengono accedute ripetutamente mentre le copie del contenuto sono disponibili nella cache, allora queste saranno prelevate velocemente.

Memoria di massa Come si è visto la memoria centrale ha un ruolo essenziale, ma tende anche a essere costosa e, salvo casi particolari, è *volatile*, cioè il contenuto va perso interrompendo l'alimentazione elettrica. Pertanto, quando si deve memorizzare in modo permanente o per lungo tempo un volume grande di dati o un numero di programmi elevato, si ricorre all'uso della *memoria di massa* o *secondaria* (*mass* o *secondary storage*), soprattutto per quegli elementi di informazione che non vengono usati con frequenza. La memoria di massa è *persistente*, cioè il contenuto si conserva anche in assenza di alimentazione. I tempi di accesso per la memoria secondaria sono più lunghi di quelli per la memoria primaria.

C'è una vasta scelta di tecnologie adatte alla realizzazione di unità di memoria di massa: alcuni esempi sono *disco magnetico*, *disco ottico* (CD e DVD) e *dispositivi di memoria flash*.

1.2.3 Unità aritmetica-logica

La maggior parte dei calcoli o delle operazioni svolte dal calcolatore viene eseguita dall'*unità aritmetica-logica*, o ALU (*arithmetic-logic unit*), del processore. Ciascuna operazione aritmetica o logica, come addizione, sottrazione, moltiplicazione, divisione o confronto tra numeri, inizia portandone gli operandi all'interno del processore, qui li si sottopone all'operazione desiderata e infine se ne manda il risultato a destinazione. All'interno del processore l'operando (come pure il risultato) viene scritto in un elemento di memoria apposito, chiamato appunto *registro* (*register*), caratterizzato da velocità di lettura e scrittura elevatissima: i registri

di norma hanno tutti la stessa dimensione della parola e operano a velocità nettamente superiore a quella dello strato di memoria cache più rapida presente nella gerarchia di memoria del calcolatore.

1.2.4 Unità di uscita

In un certo senso l'unità di uscita è la controparte di quella di ingresso: ha infatti la funzione di mandare al mondo esterno il risultato dell'esecuzione del programma. Un esempio familiare di unità di uscita è dato dalla *stampante* (*printer*), di cui ormai si trovano in commercio numerosi e svariati modelli in tecnologie differenti: impatto meccanico, a getto di inchiostro oppure basati su tecniche di fotoimpressione, come la stampa via raggio laser; oggi si arrivano a costruire stampanti in grado di produrre fino a 20 o più pagine al minuto. A un essere umano siffatta velocità può sembrare molto elevata, ma in realtà è ancora modesta confrontandola con quella caratteristica del processore microelettronico.

Alcune unità, come gli schermi grafici, forniscono sia la funzione di output, mostrando testo e grafica, sia la funzione di input, dal contatto con lo schermo di tipo *touchscreen*. Siccome è frequente trovare unità funzionali aventi doppio ruolo di ingresso e uscita si giustifica bene l'uso del termine composito *unità di I/O*.

1.2.5 Unità di controllo

Le unità di memoria, aritmetica-logica, ingresso e uscita hanno la funzione di contenere e mantenere stabili le informazioni, elaborarle, acquisirle dal mondo esterno e inviarle, rispettivamente; è evidente come tutte queste attività vadano coordinate opportunamente. Si tratta appunto del compito dell'unità di controllo, che si può paragonare intuitivamente al sistema nervoso centrale dell'organismo: così come questo rileva le percezioni tramite i nervi e manda comandi motori ai muscoli, quella rileva lo stato delle parti del sistema e manda loro comandi operativi appropriati.

Le istruzioni macchina contenute nei programmi di I/O associati ai dispositivi di ingresso e uscita indicano le operazioni di trasferimento di I/O da svolgere, per esempio se leggere oppure scrivere dati, ma la scansione temporale del trasferimento è invece regolata dai circuiti di controllo. L'unità di controllo genera segnali che attivandosi scandiscono gli istanti di tempo in cui le varie operazioni di I/O devono avere luogo; pertanto essi governano anche la scansione temporale del trasferimento di dati tra processore e memoria.

Ragionando in modo astratto verrebbe naturale immaginarsi l'unità di controllo come se fosse un dispositivo individuale, fisicamente separato dalle altre parti del calcolatore, che interagisse con esse coordinandone le attività, ma di rado tale elegante modello concettuale viene messo in pratica: la maggior parte dei circuiti di controllo è sparpagliata qua e là all'interno del calcolatore; tuttavia c'è sempre un numero abbastanza elevato di linee di controllo diramantisi per ogni dove nel calcolatore, che veicolano segnali destinati a scandire e sincronizzare le attività delle varie unità funzionali. Pertanto si possono riassumere e schematizzare nel modo seguente le operazioni svolte dal calcolatore.

- Tramite l'unità di ingresso ordini e informazioni vengono acquisiti dal mondo esterno, espressi sotto le due forme seguenti:
  - il programma, che dirige le attività del calcolatore;
  - i dati, su cui il programma opera elaborandoli;

  e vengono scritti in memoria.
- Le istruzioni del programma vengono prelevate progressivamente dalla memoria per metterle in esecuzione.

- Sotto la guida delle istruzioni via via eseguite, i dati vengono letti dalla memoria, mandati all'unità aritmetica-logica e quivi elaborati producendo i risultati, che vengono scritti in memoria.
- I risultati finali vengono mandati al mondo esterno tramite l'unità di uscita.
- L'unità di controllo coordina le attività di tutte le unità funzionali facenti parte del calcolatore.

1.3 Concetti operativi di base

Si è già detto come le attività del calcolatore siano guidate delle istruzioni macchina componenti il programma. Per ciascun compito da svolgere esiste un programma in memoria, formato da una lista di istruzioni. Dalla memoria il processore preleva istruzioni in modo progressivo e le mette in esecuzione, attuando le operazioni specificate da ciascuna istruzione; anche i dati oggetto delle istruzioni sono contenuti in memoria e vengono presi quando l'istruzione relativa ne ha bisogno. Ecco come si presenta un'istruzione macchina tipica:

Load R2, LOC

Questa istruzione legge i contenuti di una locazione di memoria il cui indirizzo è rappresentato simbolicamente dall'etichetta LOC e carica i contenuti nel registro R2 del processore. I contenuti originali della locazione LOC sono preservati, mentre quelli del registro R2 sono sovrascritti. L'esecuzione di questa istruzione richiede vari passi. Innanzitutto, l'istruzione è prelevata dalla memoria e inserita nel processore. Successivamente, l'operazione da eseguire è determinata dall'unità di controllo. L'operando alla locazione LOC è prelevato dalla memoria e inserito nel processore. Infine, l'operando è memorizzato nel registro R2.

Dopo che gli operandi sono stati caricati dalla memoria nei registri del processore, è possibile effettuare operazioni aritmetiche o logiche su di essi. Per esempio, l'istruzione

Add R4, R2, R3

somma i contenuti dei registri R2 e R3, quindi inserisce la loro somma nel registro R4. Gli operandi in R2 e R3 non sono alterati, ma il precedente valore in R4 è sovrascritto dalla somma.

Dopo aver completato le operazioni desiderate, i risultati sono nei registri del processore. Essi possono essere trasferiti in memoria usando istruzioni come

Store R4, LOC

Questa istruzione copia l'operando del registro R4 nella locazione di memoria LOC. I contenuti originariamente presenti nella locazione LOC sono sovrascritti, ma quello di R4 sono preservati.

Per le istruzioni Load e Store, i trasferimenti tra memoria e processore sono fatti partire mandando l'indirizzo della locazione di memoria desiderato all'unità di memoria e asserendo i segnali di controllo appropriati. I dati sono quindi trasferiti alla memoria o viceversa al processore.

La Figura 1.2 mostra i collegamenti tra processore e memoria, e il processore è schematizzato evidenziandone anche alcuni elementi strutturali interni non ancora menzionati ma fondamentali. I rispettivi collegamenti sono omessi perché qui interessa discuterne solo le funzioni; comunque saranno illustrati nel Capitolo 5 nel contesto della trattazione completa sulla struttura interna del processore.

Ecco una breve rassegna degli elementi interni del processore mostrati in Figura 1.2. Naturalmente si ritrovano le unità aritmetica-logica e di controllo, già illustrate e discusse (Figura 1.1), ma si trovano anche numerosi registri che il processore usa per una varietà di scopi differenti. Il *registro di istruzione*, o IR (*instruction register*), ha la funzione speciale di contenere

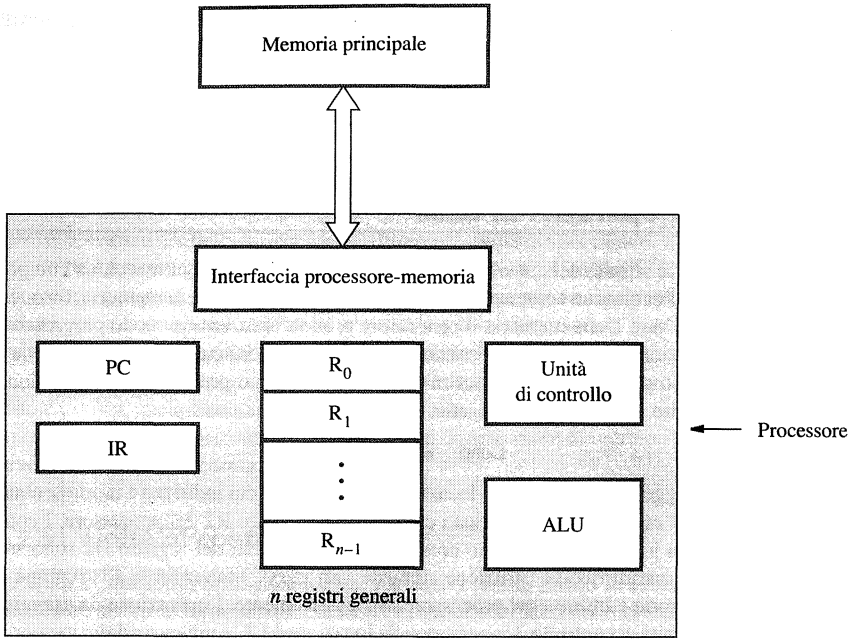


Figura 1.2 Connessione tra il processore e la memoria principale.

l'istruzione macchina prelevata dalla memoria e di mantenerla stabile per la durata dell'esecuzione. Il registro IR manda il proprio contenuto ai circuiti di controllo, che lo esaminano (o come si dice lo *decodificano*) e generano i segnali di controllo per le varie unità funzionali coinvolte nell'esecuzione dell'istruzione.

Il *contatore di programma*, o PC (*program counter*), è un altro registro specializzato: contiene l'indirizzo della parola di memoria dove si trova l'istruzione macchina da prelevare ed eseguire non appena l'esecuzione dell'istruzione corrente sarà terminata; durante l'esecuzione dell'istruzione corrente il valore contenuto in PC viene aggiornato in modo da farlo corrispondere all'indirizzo della parola contenente la prossima istruzione da eseguire. Si dice anche che il registro PC *punta* alla parola di memoria contenente la prossima istruzione da eseguire.

Oltre ai registri IR e PC, chiamati anche *registri di uso speciale* (*special purpose register*), nel processore si trovano $n \geq 1$ *registri di uso generale* (*general purpose register*), denominati R_0, R_1, ..., R_{n-1}; la loro funzione verrà illustrata e discussa nel Capitolo 2, ma per ora basta sapere che servono principalmente come deposito di dati di uso immediato e di puntatori a parole di memoria contenenti dati (o altri puntatori) su cui lavorare.

L'interfaccia processore-memoria è un circuito che gestisce il trasferimento dei dati tra la memoria principale e il processore. Se una parola deve essere letta dalla memoria, l'interfaccia manda l'indirizzo di quella parola alla memoria insieme al segnale di controllo Read. L'interfaccia aspetta che la parola sia estratta, quindi la trasferisce al registro del processore appropriato. Se una parola deve essere scritta in memoria, l'interfaccia trasferisce sia l'indirizzo sia la parola alla memoria insieme al segnale di controllo Write.

Ecco di seguito qualche esempio di successione di passi operativi tipica del processore. Si comincia mostrando come prelevare l'istruzione macchina e avviarla all'esecuzione.

Il programma da eseguire deve risiedere nella memoria centrale e di solito viene prelevato dalla memoria secondaria tramite l'unità di ingresso.

L'esecuzione del programma comincia quando il registro PC viene fatto puntare alla parola di memoria contenente l'istruzione iniziale.

I contenuti del PC sono trasferiti alla memoria insieme al segnale di controllo Read. Quando la parola indirizzata (in questo caso, la prima istruzione del programma) è stata prelevata dalla memoria, essa è caricata nel registro IR. A questo punto, l'istruzione è pronta per essere interpretata ed eseguita.

Istruzioni quali Load, Store e Add eseguono trasferimenti di dati e operazioni aritmetiche. Se un'istruzione richiede un operando che risiede in memoria, questo è prelevato mandando il suo indirizzo alla memoria e iniziando un'operazione Read. Quando l'operando è stato prelevato dalla memoria, esso è trasferito al registro del processore. Dopo che gli operandi sono stati prelevati in questo modo, l'ALU può eseguire l'operazione aritmetica desiderata, per esempio Add, sui valori nei registri del processore. Il risultato è mandato al registro del processore. Se il risultato deve essere scritto in memoria con un'istruzione Store, esso è trasferito dal registro del processore alla memoria insieme all'indirizzo della locazione dove il risultato deve essere memorizzato, quindi un'operazione Write è fatta partire.

In un qualche momento intermedio durante i passi di esecuzione così tratteggiati, il contenuto del registro PC viene incrementato in modo da farlo puntare alla parola di memoria contenente la prossima istruzione da eseguire; così non appena l'esecuzione dell'istruzione corrente sarà terminata, sarà già possibile prelevarne subito un'altra e metterla a sua volta in esecuzione.

Oltre a trasferire parole (dati e istruzioni) tra processore e memoria il calcolatore acquisisce anche dati nuovi dai dispositivi di ingresso e invia dati elaborati ai dispositivi di uscita. Tutto ciò comporta l'esistenza di istruzioni macchina apposite per guidare i trasferimenti di I/O.

Se un dispositivo di ingresso o uscita ha bisogno di un servizio urgente è consentito arrestare l'esecuzione del programma in corso e passare rapidamente a eseguire il servizio desiderato; per esempio, quando il sistema automatico di sorveglianza di un processo industriale rileva un evento pericoloso deve essere in grado di provvedere in tempo certo e breve. Il meccanismo di *interruzione* (*interrupt*) permette di sospendere su richiesta l'esecuzione del programma corrente e a grandi linee funziona nel modo seguente.

- Il dispositivo di I/O attiva e invia al processore il segnale di *richiesta di interruzione* (*interrupt request*).
- Il processore ascolta e serve la richiesta sospendendo l'esecuzione del programma corrente e passando a eseguire la relativa *routine di servizio di interruzione* (*interrupt service routine*); la routine è un programma a parte, specifico per il servizio in questione.
- Siccome la richiesta di interruzione è un evento imprevisto che potrebbe alterare lo stato interno del processore, prima di passare a servirla esso va salvato scrivendolo in una serie di parole di memoria riservate per tale uso. Di solito lo stato da salvare in memoria comprende i contenuti del registro PC, dei registri di uso generale e qualche altro elemento di informazione avente funzioni di controllo (e che varia un po' a seconda del processore).
- Quando la routine di servizio dell'interruzione è terminata lo stato interno del processore viene ripristinato ricaricandolo dalla memoria dove era stato salvato, cosicché si può riprendere normalmente l'esecuzione del programma sospeso in precedenza.

Questo paragrafo ha fornito una panoramica del funzionamento di un calcolatore. Questi concetti saranno discussi in dettaglio nei capitoli seguenti, prima dal punto di vista del programmatore nei Capitoli 2, 3 e 4, e quindi dal punto di vista del progettista dell'hardware nei capitoli successivi.

1.4 Rappresentazione dei numeri e operazioni aritmetiche

Il modo più naturale per rappresentare un numero in un calcolatore è tramite una stringa di bit, chiamata numero binario. Descriveremo prima le rappresentazioni in numeri binari per gli interi e le operazioni aritmetiche su esse. Successivamente forniremo una breve introduzione della rappresentazione dei numeri in virgola mobile.

1.4.1 Numeri interi

La stringa di bit B è un vettore costituito da ≥ 1 bit b_i $(0 \leq i \leq n-1)$, ciascuno dei quali può assumere valore 0 o 1, vale a dire $bi \in \{0, 1\}$, e si denota nel modo seguente:

$$B = \underbrace{b_{n-1}b_{n-2}\ldots b_1 b_0}_{n \text{ bit}}$$

La stringa B può rappresentare un *valore numerico intero naturale* val (B) compreso nell'intervallo $[0, 2n)$ (estremo inferiore incluso, superiore escluso), che si calcola mediante la formula seguente:

$$\text{val}(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_1 \times 2^1 + b_0 \times 2^0 = \sum_{i=0}^{n-1} b_i 2^i \qquad (1.1)$$

È necessario lavorare con numeri positivi e negativi, vale a dire in generale con i *numeri interi relativi* (o semplicemente *numeri interi*). Le tre tecniche di rappresentazione più importanti per il numero intero relativo sono le seguenti:

- *segno e valore assoluto* (*sign and magnitude*);
- *complemento a uno* (*one's complement*);
- *complemento a due* (*two's complement*).

Esse hanno una proprietà in comune: se il bit più a sinistra, cioè b_{n-1}, vale 0 il valore rappresentato val (B) è positivo o nullo, mentre se vale 1 è negativo (o al più nullo, in due di esse). Avendo un ruolo così speciale il bit b_{n-1} viene chiamato *bit di segno* (*sign bit*), appunto perché rivela il segno del valore. La distribuzione dei valori è però un po' diversa nei tre casi, come mostra la Figura 1.3 ponendo $n = 4$ cioè considerando stringhe di quattro bit.

In tutti e tre i casi in esame i valori positivi si distribuiscono nello stesso modo rispetto alle stringhe di bit che li codificano, mentre i valori negativi si differenziano. Ecco le regole per passare da B a $-B$, ovvero da val (B) all'opposto val $(-B) = -$val (B).

- Segno e valore assoluto: si commuta il bit di segno b_{n-1} da 0 a 1 oppure da 1 a 0, mentre tutti gli altri bit restano invariati. Per esempio, +5 è rappresentato da 0101, −5 è rappresentato da 1101.
- Complemento a uno: si commuta ciascun bit b_i $(0 \leq i \leq n \leq n-1)$ da 0 a 1 oppure da 1 a 0; così facendo si dice che il numero B viene *complementato a 1*.

Quindi, la rappresentazione di −3 è ottenuta complementando ciascun bit del vettore 0011 e diventa 1100. La stessa operazione, complemento dei bit, è effettuata per convertire un numero negativo nel corrispondente valore positivo. La conversione fra i due modi è intesa come la formazione del complemento a uno di un certo numero. Per numeri a n bit, questa operazione

B				Valore rappresentato		
b_3	b_2	b_1	b_0	Segno e val. ass.	Comp. a 1	Comp. a 2
0	1	1	1	+7	+7	+7
0	1	1	0	+6	+6	+6
0	1	0	1	+5	+5	+5
0	1	0	0	+4	+4	+4
0	0	1	1	+3	+3	+3
0	0	1	0	+2	+2	+2
0	0	0	1	+1	+1	+1
0	0	0	0	+0	+0	+0
1	0	0	0	−0	−7	−8
1	0	0	1	−1	−6	−7
1	0	1	0	−2	−5	−6
1	0	1	1	−3	−4	−5
1	1	0	0	−4	−3	−4
1	1	0	1	−5	−2	−3
1	1	1	0	−6	−1	−2
1	1	1	1	−7	−0	−1

Figura 1.3 Distribuzione del valore in rappresentazione, in segno e valore assoluto, complemento a uno e a due, per numeri da $n = 4$ bit.

è equivalente alla sottrazione del numero da $2^n - 1$. Nel caso dei numeri a 4 bit in Figura 1.3, sottraiamo da $2^4 - 1 = 15$, o 1111 in binario.

Infine, nel sistema del *complemento a due*, l'effettuazione del complemento a due di un numero a n bit consiste nella sottrazione del numero da $2n$. Quindi, il complemento a due di un numero è ottenuto aggiungendo 1 al complemento a uno di quel numero.

Si noti che ci sono rappresentazioni diverse per +0 e −0 in entrambi i sistemi del segno e valore assoluto e del complemento a uno, ma il sistema del complemento a due ha solo una rappresentazione per 0. Per numeri a 4 bit, come mostrato in Figura 1.3, il valore −8 è rappresentabile nel sistema del complemento a due ma non negli altri sistemi.

Parrebbe che la tecnica più naturale di rappresentazione del numero intero relativo sia quella in segno e valore assoluto, perché il segno viene espresso in forma separabile dal valore assoluto esattamente come si fa con i numeri decimali calcolando a mano. Osservando la Figura 1.3 si capisce subito quale sia la relazione intercorrente tra la tecnica in segno e valore assoluto e quella in complemento a uno, mentre la tecnica in complemento a due sembra alquanto bizzarra e senza chiara giustificazione. Invece nel seguito si mostrerà come proprio quest'ultima risulti essere quella più efficiente nel calcolo di addizione e sottrazione. In effetti la tecnica in complemento a due è quella usata universalmente in tutti i calcolatori, mentre le altre due, quand'anche siano disponibili, in genere sono facoltative, in particolare il complemento a uno.

Addizione di numeri naturali

La Figura 1.4 mostra l'addizione di numeri naturali da un bit, vale a dire ponendo $n = 1$. La somma di 1 e 1 è il vettore di 2 bit 10, che rappresenta il valore 2. Diciamo che la *somma* è 0 e che il *riporto in uscita* (*carry-out*) è 1.

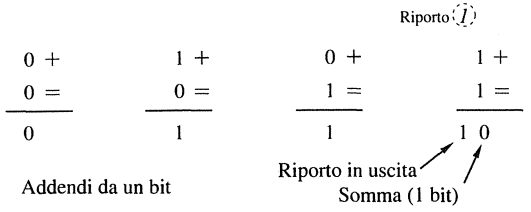


Figura 1.4 Addizione di numeri a un bit.

Per addizionare due numeri naturali rappresentati con $n > 1$ bit si usa lo stesso algoritmo di addizione manuale in colonna dei numeri decimali, salvo tenere conto che qui la base è 2. Prima si incolonnano gli addendi allineandone gli n bit da destra verso sinistra e poi si addizionano in colonna i bit procedendo da quelli meno significativi verso quelli più significativi, propagando l'eventuale riporto verso sinistra. Il riporto in uscita da una coppia di bit diventa il *riporto in entrata* (*carry-in*) alla prossima coppia di bit alla sinistra. Il riporto in entrata deve essere aggiunto alla coppia di bit per ottenere la somma e il riporto in uscita in quella posizione. Per esempio, se entrambi i bit di una coppia sono 1 e il riporto in entrata è 1, allora la somma è 1 e il riporto in uscita è 1, il che rappresenta il valore 3.

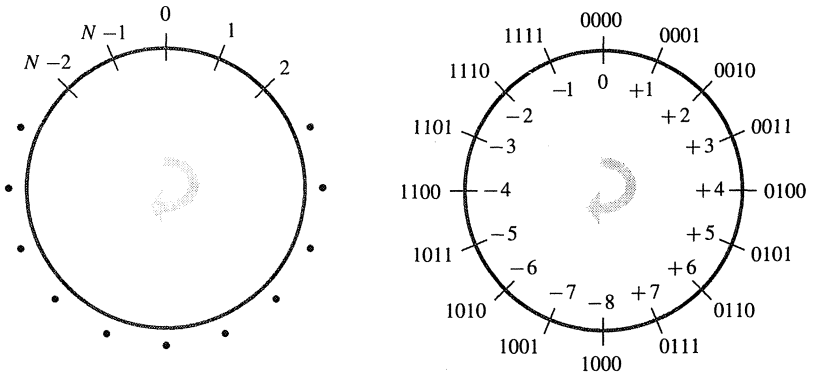
Addizione e sottrazione di interi con segno

Finora si sono viste tre tecniche per la rappresentazione del numero relativo (cioè positivo e negativo), che si differenziano solo per come rappresentano i valori negativi. Dal punto di vista dell'efficienza di calcolo delle operazioni aritmetiche di addizione e sottrazione esse si dispongono nel modo seguente: la tecnica in segno e valore assoluto è la più semplice concettualmente, ma è anche quella meno efficiente; quella in complemento a uno si comporta un po' meglio, però quella in complemento a due è senza dubbio la migliore.

Per comprendere l'aritmetica in complemento a due è opportuno ricorrere a un modello geometrico. Fissato un intero positivo $N \geq 1$ si consideri l'addizione modulo N; i numeri interi su cui si lavora sono quelli positivi o nulli compresi nell'intervallo $[0, n)$; per definizione si pone $(a + b) \bmod N$ = resto della divisione di $a + b$ rispetto a N. Si ottiene la rappresentazione grafica prendendo una circonferenza orientata in senso orario e marcata con esattamente N punti disposti regolarmente ed etichettati con gli interi da 0 a $N - 1$; la Figura 1.5a mostra il modello in generale.

Per calcolare l'addizione $(a + b) \bmod N$ ci si posiziona sulla circonferenza al punto a e poi si gira in senso orario di b passi; il punto di arrivo è il risultato. Per esempio fissando $N = 16$ e girando sulla circonferenza si hanno i risultati seguenti: $(7 + 4) \bmod 16 = 11$ e $(9 + 14) \bmod 16 = 7$ (infatti si ha che il resto di $11 \div 16$ vale 11 e quello di $26 \div 16$ vale 7); nel primo caso non si completa il giro, mentre nel secondo si ricircola. L'addizione modulo N così definita funziona per qualunque coppia di addendi $a, b \in [0, N)$ e ha tutte le proprietà consuete (per esempio è associativa, commutativa, esiste l'elemento neutro ecc).

Per stabilire il nesso con la tecnica in complemento a due basta interpretare la circonferenza in modo diverso. Sempre fissando $N = 16$ si etichettino i punti con le 16 sequenze di $n = 4$ bit 0000, 0001, ... , 1111, secondo il sistema di numerazione binario naturale. Ciò fatto, si interpretino le sequenze in complemento a due in modo da farle corrispondere agli interi relativi da -8 a $+7$ (estremi inclusi), facendo riferimento alla Figura 1.3; la circonferenza così rietichettata è mostrata in Figura 1.5b.



(a) Circonferenza rappresentante il sistema numerico modulo N

(b) Corrispondenza tra i sistemi numerici mod 16 e in complemento a due con $n = 4$ bit

Figura 1.5 Rappresentazione geometrica del sistema numerico in comp. a due.

Si mostra ora come applicare la tecnica di addizione modulo 16 al caso di addizione algebrica $7 + (-3) = +4$ in complemento a due. Le rappresentazioni in complemento a due di $+7$ e -3 sono 0111 e 1101, rispettivamente. Per addizionare i due numeri ci si posiziona prima al punto 0111 sulla circonferenza di Figura 1.5b e poi si gira in senso orario di 1101 passi interpretando tale sequenza di bit come se fosse un numero naturale; vale a dire si gira di 13 passi arrivando infine al punto 0100, che interpretato in complemento a due vale $+4$, la risposta corretta. Interpretare il secondo addendo come numero naturale e girare di altrettanti passi lungo la circonferenza è analogo ad addizionare in naturale trascurando il riporto in uscita, come mostra l'esempio seguente, dove si vede anche la catena di propagazione del riporto:

1	1	1	1			Catena di propagazione del riporto
	0	1	1	1	+	Primo addendo = +7 in comp. a 2
	1	1	0	1	=	Secondo addendo = −3 in comp. a 2
1	0	1	0	0		Somma completa a 5 bit

↑ Riporto in uscita (da trascurare)

Somma a 4 bit = +4 in comp. a 2

Si vede chiaramente come trascurando il riporto in uscita e considerando solo i quattro bit di somma, quelli meno significativi, si ottenga il risultato corretto dell'addizione in complemento a due. Tale proprietà è sempre verificata: *se* $N = 2^n$ (qui vale $N = 16 = 24$), *addizionare in naturale trascurando il riporto in uscita è equivalente ad addizionare modulo N ed è equivalente ad addizionare in complemento a due.* Infatti in naturale il valore successivo a 1111 è 10000, ma girando lungo la circonferenza si torna a 0000, e in effetti 10000 si riduce a 0000 trascurando il bit più significativo. La conclusione sintetica è la seguente: *l'algoritmo di addizione in complemento a due è identico a quello in naturale, pur di trascurare il riporto in uscita.*

Ecco le regole esplicite per calcolare addizione e sottrazione algebrica di numeri relativi codificati in complemento a due (con n bit).

1. Per calcolare l'addizione algebrica si applica l'algoritmo di addizione in aritmetica binaria naturale, ma si trascura il riporto in uscita dalla posizione dei bit più significativi degli addendi. La somma è corretta se è compresa nell'intervallo $[=2^{n-1}, 2n^{-1})$.
2. Supponendo che A e B siano minuendo e sottraendo, rispettivamente, si calcola la sottrazione algebrica $A - B$ nel modo seguente:
   - prima si complementa a 2 il sottraendo B, così ottenendo $B' = -B$;
   - poi si calcola l'addizione algebrica $A + B'$, secondo la regola (1).

La differenza è corretta se è compresa nell'intervallo $[-2^{n-1}, 2^{n-1})$.

In Figura 1.6 sono mostrati svariati esempi di addizione e sottrazione algebrica in complemento a due, per numeri da $n = 4$ bit che assumono valori compresi nell'intervallo $[-8, +8)$. Quando la somma o la differenza non cade nell'intervallo si dice che si è verificato evento di *trabocco* (*overflow*), e come rilevarlo è spiegato nel prossimo paragrafo. Le prime quattro

(a)	0010 +	(+2)		(b)	0100 +	(+4)
	0011 =	(+3)			1010 =	(−6)
	0101	(+5)			1110	(−2)
(c)	1011 +	(−5)		(d)	0111 +	(+7)
	1110 =	(−2)			1101 =	(−3)
	1001	(7)			0100	(+4)
(e)	1101 −	(−3)	⇒		1101 +	
	1001 =	(−7)			0111 =	
					0100	(+4)
(f)	0010 −	(+2)	⇒		0010 +	
	0100 =	(+4)			1100 =	
					1110	(−2)
(g)	0110 −	(+6)	⇒		0110 +	
	0011 =	(+3)			1101 =	
					0011	(+3)
(h)	1001 −	(−7)	⇒		1001 +	
	1011 =	(−5)			0101 =	
					1110	(−2)
(i)	1001 −	(−7)	⇒		1001 +	
	0001 =	(+1)			1111 =	
					1000	(−8)
(l)	0010 −	(+2)	⇒		0010 +	
	1101 =	(−3)			0011 =	
					0101	(+5)

Figura 1.6 Operazioni di addizione e sottrazione in complemento a due.

operazioni in Figura 1.6, da (a) a (d), sono di addizione algebrica e seguono la regola (1), mentre le sei seguenti, da (e) a (l), sono di sottrazione algebrica e seguono la regola (2), che prevede di complementare a 2 il sottraendo senza distinguere tra numeri positivi o negativi.

La semplicità estrema degli algoritmi di addizione, sottrazione ed estensione o riduzione del segno di numeri relativi in rappresentazione in complemento a due è il motivo che spiega perché questa sia la tecnica adottata in tutti i calcolatori odierni. Potrebbe sembrare che la tecnica in complemento a uno sia altrettanto valida, ma c'è una difficoltà: se nell'addizione il riporto in uscita vale 1 non lo si può trascurare, bensì la somma va incrementata di un'unità; tale ciclo di correzione, dipendente dal riporto in uscita (donde il nome di *riporto circolare* o *carry around*), rallenta l'operazione e rende il tutto meno vantaggioso del complemento a due. Infine la tecnica in segno e valore assoluto richiede, quando si addiziona, sottrae ecc, di esaminare prima i segni degli operandi per decidere come procedere, dando luogo a un rallentamento che rende tale tecnica decisamente svantaggiosa rispetto a entrambe le altre e segnatamente al complemento a due.

Estensione e riduzione del segno

Accade spesso che si debba aumentare o diminuire il numero di bit usati per codificare un numero relativo in complemento a due, magari per adattarlo a una parola di memoria o a un registro di processore avente dimensione differente; le regole per farlo sono semplicissime.

1. Aumento o *estensione*:
   - se il numero è positivo inizia con 0 e si devono aggiungere altri bit 0 a sinistra:

     $$+3 = 2 + 1 = 011 = 0011 = 00011 = \ldots$$

   - se è negativo inizia con 1 e si devono aggiungere altri bit 1 a sinistra:

     $$-3 = -4 + 1 = 101 = 1101 = 11101 = \ldots$$

2. Diminuzione o *riduzione*:
   - se il numero è positivo inizia con 0 e si possono togliere bit 0 a sinistra, ma bisogna smettere prima che emerga in testa il bit 1:

     $$+3 = 00011 = 0011 = 011 = \text{alt!} \qquad (+3 = 011 \neq 11 = -1)$$

   - se è negativo inizia con 1 e si possono togliere bit 1 a sinistra, ma bisogna smettere prima che emerga in testa il bit 0:

     $$-3 = 11101 = 1101 = 101 = \text{alt!} \qquad (-3 = 101 \neq 01 = +1)$$

Per capire perché la regola (1) funzioni con numeri negativi si provi a ridisegnare la circonferenza di Figura 1.5b per $N = -32$ o $N = -64$, constatando che le codifiche di $-1, -2, \ldots$ sono identiche, solo si aggiungono bit 1 a sinistra. Nella regola (2) il motivo dell'arresto è che se emergesse in testa il bit 1 (o 0) il segno del numero cambierebbe; non si può ridurre il numero di bit oltre un certo limite. Le regole (1) e (2) vanno sotto il nome di *estensione* e *riduzione* del segno: in pratica affermano che il numero non muta replicandone il bit di segno, oppure togliendoglielo purché non si inverta.

Evento di trabocco

Si è visto come in complemento a due con $n > 1$ bit si rappresentino in modo univoco tutti i numeri interi relativi compresi nell'intervallo $[-2^{n-1}, 2^{n-1})$; per esempio con $n = 4$ bit l'intervallo si estende da -8 a $+7$, estremi inclusi, come mostra anche la Figura 1.3. Se succede che l'operazione aritmetica (addizione o altra) produce un risultato fuori dall'intervallo si dice che si è verificato un evento di *trabocco* (*overflow*).

Addizionando due numeri rappresentati in binario naturale si verifica trabocco se e solo se il riporto in uscita dalla posizione più significativa dei due addendi vale 1. Però tale condizione

non funziona addizionando numeri relativi rappresentati in complemento a due. Per esempio addizionando i due numeri +7 e +4 in complemento a due con $n = 4$ bit si ottiene la somma 1011, che in complemento a due rappresenta −5, risultato certamente errato e che dunque comporta la presenza di trabocco, ma il riporto in uscita dalla quarta posizione più a sinistra, vale a dire in uscita dal bit più significativo, vale 0, apparentemente non segnalando nulla di anomalo; d'altra parte addizionando in complemento a due −4 e −6 si ottiene la somma 0110 = +6, di nuovo errato, ma questa volta il riporto in uscita vale 1, segnalando errore. Pertanto in complemento a due il riporto in uscita non costituisce indicazione affidabile di trabocco. Si osserva però che se i due addendi sono *discordi*, vale a dire hanno segno differente, il trabocco non si può verificare, giacché l'addizione (algebrica) di fatto si traduce in sottrazione; in valore assoluto la differenza non supera mai il maggiore degli addendi e dunque se n bit bastano per gli addendi, di certo bastano anche per la somma; invece se gli addendi sono *concordi*, cioè hanno lo stesso segno, il trabocco può verificarsi, benché non necessariamente. Riassumendo:

- si può verificare trabocco solo se gli addendi sono concordi in segno;
- il riporto in uscita al bit più significativo in generale non indica trabocco.

La regola per determinare la presenza di trabocco in complemento a due è invece la seguente: *si verifica trabocco se e solo se i due addendi sono concordi in segno e il bit di segno della somma è diverso da quello degli addendi* (che lo hanno identico). Per capire perché la regola funzioni basta rendersi conto che se gli addendi sono concordi anche la somma deve concordare con essi.

Quando si sottraggono due numeri, il metodo di controllo richiesto per valutare il trabocco deve essere modificato in qualche modo; ma è ancora abbastanza facile (si veda il Problema 1.10).

1.4.2 Numeri frazionari

Finora si sono considerati solo numeri interi, naturali e relativi. Qui si prenderanno in esame i formati di rappresentazione per numeri frazionari, cioè in generale dotati di parte intera e frazionaria entrambe diverse da zero.

Se usiamo un'intera parola in un calcolatore con parole a 32 bit di lunghezza per rappresentare un intero con segno in complemento a due, l'intervallo di valori che può essere rappresentato è da -2^{31} a $+2^{31}$. In decimale, questo intervallo è un po' più piccolo dell'intervallo da -10^{10} a $+10^{10}$.

Le stesse sequenze di 32 bit possono essere interpretate come frazioni nell'intervallo da -1 a $+1 - 2^{-31}$ se assumiamo che la virgola binaria implicita sia appena alla destra del bit di segno; ovvero, tra il bit b_{31} e il bit b_{30} all'estremo sinistro della rappresentazione a 32 bit. In questo caso, il valore assoluto della più piccola frazione rappresentabile è circa 10^{-10}.

Nessuna di queste due rappresentazioni di numeri *in virgola fissa* ha un intervallo che è sufficiente per molti calcoli scientifici o ingegneristici. Per convenienza, vorremmo una rappresentazione di numeri binari che possa facilmente contenere sia grandi numeri interi sia frazioni molto piccole. Per far questo, un calcolatore deve essere in grado di rappresentare numeri e operare su essi in modo che la posizione della virgola binaria sia variabile e automaticamente aggiustata con il procedere delle elaborazioni. In questo caso, la virgola binaria si dice *mobile*, e i numeri sono chiamati *numeri in virgola mobile*.

Poiché la posizione della virgola in un numero in virgola mobile varia, essa deve essere indicata esplicitamente nella rappresentazione. Per esempio, nell'usuale notazione scientifica decimale, i numeri possono essere scritti come $6{,}0247 \times 10^{23}$, $3{,}7291 \times 10^{-27}$, $-1{,}0341 \times 10^{2}$, $-7{,}3000 \times 10^{-14}$ ecc. Diciamo che questi numeri sono espressi con 5 *cifre significative* di precisione. I *fattori di scala* 10^{23}, 10^{-27}, 10^{2} e 10^{-14} indicano la posizione reale della virgola decimale rispetto alle cifre significative. Lo stesso approccio può essere

usato per rappresentare numeri binari in virgola mobile in un calcolatore, eccetto che è più appropriato usare 2 come base per il fattore di scala. Poiché la base è fissata, non occorre che sia data nella rappresentazione. L'esponente può essere positivo o negativo.

In conclusione un numero binario in virgola mobile può essere rappresentato tramite:

- un segno per il numero;
- alcuni bit significativi;
- un esponente con segno per un fattore di scala con implicita base 2.

Uno standard internazionale ampiamente riconosciuto dell'IEEE (Institute of Electrical and Electronics Engineers) per la rappresentazione di numeri in virgola mobile a 32 bit usa un bit per il segno, 23 bit significativi e 8 bit per un esponente con segno del fattore di scala, che ha base implicita 2. In termini decimali, l'intervallo di numeri rappresentati va all'incirca da $\pm 10^{-38}$ a $\pm 10^{38}$, che è un intervallo adeguato per la maggior parte dei calcoli scientifici e ingegneristici. Lo stesso standard IEEE definisce anche una rappresentazione a 64 bit per contenere più bit significativi e più bit per l'esponente con segno; ciò risulta in una precisione più alta e un più largo intervallo di valori.

La rappresentazione di numeri in virgola mobile e le operazioni aritmetiche su numeri in virgola mobile sono considerate in dettaglio nel Capitolo 9. Alcuni dei processori in commercio descritti nelle schede dei prossimi capitoli includono operazioni su numeri in virgola mobile nel loro insieme di istruzioni e hanno registri del processore dedicati a contenere numeri in virgola mobile.

1.5 Rappresentazione dei caratteri

Lo schema più comune per la codifica dei caratteri è l'ASCII (American Standard Code for Information Interchange). I caratteri alfanumerici, gli operatori, i simboli di punteggiatura e i caratteri di controllo sono rappresentati da codici a 7 bit come mostrato in Tabella 1.1 e 1.2.

Tabella 1.1 Codice ASCII a 7 bit

	Bit 654							
Bit 3210	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SPACE	0	@	P	'	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	/	l	vert
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Tabella 1.2 Caratteri speciali del codice ASCII

Carattere	Nome (ing.)	Nome (ita.)
NUL	null/idle	terminatore di stringa/segnale di inattività
SOH	start of header	carattere di inizio di intestazione
STX	start of text	carattere di inizio di testo
ETX	end of text	carattere di fine di testo
EOT	end of transmission	carattere di fine di trasmissione
ENQ	enquiry	carattere di interrogazione
ACK	acknowledgment	carattere di riscontro
BEL	audible signal	segnale di cicalino di terminale
BS	back space	cursore indietro di un carattere
HT	horizontal tab	cursore o carrello in pos. di tab. orizzontale
LF	line feed	cursore o carrello sotto di una riga
VT	vertical tab	cursore o carrello in pos. di tab. verticale
FF	form feed	segnale di emissione di foglio bianco
CR	carriage return	cursore o carrello in prima colonna
SO	shift out	segnale di emissione di bit
SI	shift in	segnale di introduzione di bit
DLE	data link escape	carattere di fuga per linea dato
DC1-DC4	device control	caratteri di controllo dispositivo
NACK	negative acknowledgment	carattere di riscontro negativo
SYN	synchronous idle	segnale sincrono di inattività
ETB	end of transmission block	carattere di fine di blocco di dati trasmesso
CAN	cancel (error in data)	segnale di cancellazione (dato errato)
EM	end of medium	carattere di fine dispositivo
SUB	special sequence	segnale di sequenza speciale
ESC	escape	carattere di fuga
FS	file separator	carattere separatore di file
GS	group separator	carattere separatore di gruppo
RS	record separator	carattere separatore di blocco (record)
US	unit separator	carattere separatore di unità
DEL	delete/idle	carattere di cancellazione/segnale di inattività

È conveniente usare un *byte* di 8 bit per rappresentare e memorizzare un carattere. Il codice occupa i sette bit della parte più bassa. Il bit di ordine più alto è di solito posto a 0. Si noti che i codici per caratteri alfabetici e numerici sono in ordine sequenziale crescente quando interpretati come numeri binari senza segno. Questo facilita le operazioni di ordinamento su dati alfabetici e numerici.

I quattro bit di ordine più basso dei codici ASCII per le cifre decimali da 0 a 9 sono i primi dieci valori del sistema di numeri binario. Questa codifica a quattro bit è intesa come il codice di *codifica binaria dei decimali* o BCD (*binary-coded decimal*).

La diffusione dell'informatica ha fatto sorgere l'esigenza di codici più ricchi per rappresentare testi in tutte le lingue, dunque per esempio con caratteri decorati da segni diacritici (accenti, dieresi, cedilla ecc.), testi in alfabeti diversi da quello inglese, testi in linguaggi di scrittura ideografica (molto frequenti nei paesi asiatici). Gli standard più rilevanti di *internazionalizzazione* della rappresentazione binaria dell'informazione testuale sono:

- famiglia di codici ISO 8859-x: a 8 bit, estensioni del codice ASCII;

- codice ISO/IEC 10646 (UCS: *Universal Character Set*): rappresentazione universale di caratteri, che estende la suddetta famiglia;
- standard di codifica basati sul codice UCS, quali Unicode e UTF-8, dove si impiega un numero variabile di byte per la codifica di ciascun carattere; UTF-8 è un'estensione compatibile del codice ASCII, ed è lo standard maggioritario nel World-Wide Web.

Da quanto visto sopra, un codice binario è essenzialmente una funzione che associa una sequenza univoca di bit a ciascun simbolo di un insieme finito, detto *alfabeto*. Tuttavia, non è sufficiente rappresentare un testo tramite la sequenza binaria determinata dal codice di rappresentazione, ma occorre aggiungervi dell'*informazione di controllo*, anch'essa in forma binaria, per far fronte alla possibilità che l'informazione binaria codificata sia affetta da errori, quali quelli dovuti a:

- disturbi fisici, interferenze ecc., nella trasmissione in canali di comunicazione;
- alterazione, accidentale o dolosa, dell'informazione binaria in dispositivi di memorizzazione.

L'Approfondimento applicativo 1.1 mostra esempi pratici di meccanismi di protezione dell'informazione a tal fine.

Molti di questi si applicano alla codifica binaria di informazione qualsiasi, non necessariamente testuale, quale quella costituita da suoni, immagini, sequenze video. L'Approfondimento applicativo 1.2 introduce e illustra i concetti alla base di tali rappresentazioni. Queste, soprattutto quando se ne desidera un'alta fedeltà, impegnano molta più memoria nei supporti, e richiedono maggiore capacità dei canali di comunicazione, rispetto alle codifiche binarie di informazione testuale. Tecniche di *compressione dei dati* hanno perciò acquisito crescente interesse con il diffondersi dell'informazione multimediale nei sistemi di calcolo e nelle reti informatiche. L'Approfondimento applicativo 1.3 ne spiega alcuni principi di funzionamento.

Approfondimento applicativo 1.1

Codici di controllo e correzione di errore

L'aggiunta di bit di controllo a una sequenza binaria può servire, entro certi limiti, a rivelare errori nella sequenza e, con tecniche opportune (ed entro limiti più ristretti), anche a correggerli. Ecco alcuni esempi.

Bit di parità

Si aggiunge un bit di controllo, il cui valore è determinato dalla parità del numero di bit di valore 1 nella sequenza data, per esempio 0 se pari, 1 se dispari. In tal modo si ottiene una sequenza estesa in cui il numero di bit di valore 1 ha parità fissa (pari nell'esempio). Se si verifica un errore su un singolo bit della sequenza (estesa), ciò viene rivelato dal fatto che la parità del numero di bit 1 nella sequenza mutata non è quella prescritta. Questa tecnica, di uso frequente per un *controllo di errore locale* su ciascun carattere di una sequenza in codice ASCII, non può rivelare errori su più bit del codice di un carattere, né correggere gli errori rivelati.

Byte di controllo

A sequenze binarie più lunghe di qualche byte spesso si aggiunge una sequenza di bit di parità, che effettua un *controllo di errore trasversale*. Per esempio, ciascun bit di un byte di controllo può controllare la parità di una sottosequenza (non contigua) di quella originaria, costituita dalle posizioni a distanza 8, ciascuna dalla successiva, a partire dalla posizione corrispondente a quella del bit in questione nel byte di controllo (cioè a partire dalla prima per il primo bit di controllo, dalla seconda per il secondo ecc.). Questa tecnica permette di rivelare anche più errori in uno stesso byte, poiché le loro posizioni sono controllate da posizioni distinte.

Somma di controllo, CRC

Una migliore capacità di rivelazione di errori multipli naturalmente costa una maggiore ridondanza di informazione di controllo. Il concetto di *somma di controllo* (*checksum*) generalizza quello di byte di controllo a sequenze di controllo più lunghe di un byte, ed è ulteriormente generalizzato da quello di *codice a ridondanza ciclica* (CRC), calcolato mediante algoritmi di algebra dei polinomi. Nonostante la relativa complessità della teoria matematica che li sottende, gli algoritmi di rivelazione di errore basati su di essa hanno realizzazioni hardware semplici ed efficienti, sono in grado di rivelare errori su sequenze contigue di posizioni anche piuttosto lunghe e sono perciò molto diffusi nelle reti di comunicazione di dati e nei dispositivi di memoria di massa.

Codici correttori

Per *sequenze binarie di uguale lunghezza*, si definisce come segue la *distanza di Hamming*:

- di due sequenze: numero di posizioni corrispondenti in cui differiscono;
- di un insieme di sequenze: la minima delle distanze di Hamming di coppie di sequenze distinte nell'insieme.

Un codice a n bit di un alfabeto A di al più 2^n simboli è una funzione iniettiva $c : a \rightarrow \{0,1\}^n$. La *distanza di Hamming di un codice* è la distanza di Hamming della sua immagine $c(A)$. Per esempio, il codice ASCII a 7 bit esteso con un bit di parità è un codice a 8 bit con distanza di Hamming 2. Si noti che l'immagine di tale codice contiene solo 2^7 delle 2^8 sequenze binarie del codominio. In generale, valgono i seguenti fatti. Un codice che abbia:

- distanza di Hamming h può rivelare fino a $h - 1$ errori;
- distanza di Hamming $h = 2k + 1$ può correggere fino a k errori.

Per rendersi conto del primo di questi fatti, basta considerare che una sequenza binaria affetta da m errori differisce in m posizioni dalla sequenza originaria; se questa è nell'immagine di un codice con distanza di Hamming $h > m$, la sequenza erronea non potrà appartenere a tale immagine (per definizione di distanza di Hamming di un codice), pertanto viene riconosciuta come erronea.

La correzione di errore ha una spiegazione analoga, ma un po' più complessa. Se una data sequenza binaria s muta nella sequenza s' per effetto di m errori e $m < h/2$, allora s' avrà distanza di Hamming $m < h/2$ da s, mentre la sua distanza di Hamming da ogni altra sequenza binaria nell'immagine del codice sarà maggiore di $h/2$, dunque è possibile determinare la corretta sequenza originale come quella, nell'immagine del codice, a distanza di Hamming minima dalla sequenza erronea.

A titolo di esempio, si consideri l'alfabeto X costituito dalle 16 cifre esadecimali 0,...,9,A,...,F (con tali cifre, che designano i numeri da 0 a 15, si costituisce una rappresentazione posizionale dei numeri in base 16, si veda il Paragrafo 2.5.3). Un naturale codice binario a 4 bit di rappresentazione dell'alfabeto X assegna a ogni cifra esadecimale la rappresentazione binaria senza segno del suo valore. Detto codice è mostrato nella seconda colonna della Tabella A1.1, la cui terza colonna mostra un codice a 7 bit per lo stesso alfabeto. Come si può vedere, questo codice ha distanza di Hamming 3 e dunque permette la correzione di errore su singolo bit. Per esempio, si supponga che il codice a 7 bit della cifra B venga mutato da un errore nella sequenza 0100011. La quarta colonna della tabella mostra la sua distanza di Hamming $h_{0100011}$ da ciascuno dei codici a 7 bit delle cifre esadecimali, da cui si vede che quello a distanza di Hamming minima è il codice della cifra B.

Tabella A1.1 Esempio di correzione di errore

X	4 bit	7	$h_{0100011}$
0	0000	0000000	3
1	0001	1101001	3
2	0010	0101010	2
3	0011	1000011	2
4	0100	1001100	6
5	0101	0100101	2
6	0110	1100110	3
7	0111	0001111	3
8	1000	1110000	4
9	1001	0011001	4
A	1010	1011010	5
B	1011	0110011	1
C	1100	0111100	5
D	1101	1010101	5
E	1110	0010110	4
F	1111	1111111	4

Ci si può chiedere come sia stato ottenuto il codice a 7 bit nella terza colonna della tabella, sì da garantirne una distanza di Hamming 3. Numerando da sinistra a destra le posizioni dei suoi bit a partire da 1 fino a 7, si può osservare che per ogni cifra dell'alfabeto X la sequenza dei bit nelle posizioni 3,5,6,7 coincide con quella del codice binario a 4 bit della stessa cifra. In altre parole, il codice a 7 bit è ottenuto da quello a 4 bit estendendolo con tre bit di controllo nelle posizioni 1,2,4 del codice esteso. La collocazione dei bit di controllo in tali posizioni, ovvero le potenze di 2 fra quelle disponibili, non è casuale, ma risulta da un algoritmo, ideato da Richard Hamming, che realizza la costruzione di codici a correzione di errore su singolo bit per sequenze binarie di lunghezza arbitraria.*

Problema

È possibile trasformare il codice a 7 bit in tabella in uno che abbia la stessa distanza di Hamming ma in cui i bit di controllo occupino posizioni contigue? Se sì, come? Se no, perché?

---

* Un simulatore basato su questo algoritmo è liberamente disponibile sul sito web http://www.dmi.unict.it/scollo/slidy/share/simulators/HammingCode10

Approfondimento applicativo 1.2

Rappresentazione binaria di informazione multimediale

I suoni e le immagini sono in origine informazioni analogiche: un segnale audio analogico rappresenta l'andamento di una pressione acustica nel tempo, mentre un'immagine è rappresentata da una distribuzione, in uno spazio bidimensionale, di valori di luminanza o di colore. Un segnale video analogico rappresenta una sequenza temporale di immagini. Tali segnali, per essere suscettibili di elaborazione numerica, vanno trasformati in sequenze discrete di campioni discreti, ciascuno rappresentabile da un numero finito di bit. La prima discretizzazione è detta *campionamento*, mentre la seconda rappresenta una *quantizzazione* nel codominio dei numeri reali. La Figura A1.1 mostra l'effetto di entrambe, con il campionamento operante sulle ascisse (dominio del tempo in questo caso) e la quantizzazione sulle ordinate (codominio dei valori del segnale). Nell'esempio mostrato in figura, il segnale può assumere valori negativi, ma i suoi valori quantizzati sono limitati fra $-4q$ e $3q$, dove q è l'unità di quantizzazione. Poiché ciascun valore analogico campionato viene approssimato al multiplo di q più vicino, i valori quantizzati sono tutti multipli di q, perciò rappresentabili dai rispettivi coefficienti di q. I valori quantizzati mostrati in figura sono così rappresentabili da numeri interi con segno nell'intervallo $[-4,3]$, dunque con 3 bit nella rappresentazione in complemento a due. Il segnale campionato e quantizzato è perciò rappresentabile con 21 bit.

Il reciproco $1/T$ del periodo di campionamento è la *frequenza di campionamento* (espressa in Hz) e la qualità della riproduzione dell'informazione dipende da questa, dall'unità di quantizzazione e dalla fisiologia della percezione dell'informazione multimediale. Per esempio, il segnale mostrato in figura potrebbe essere un frammento di *segnale audio*; in tal caso è da distinguere se si tratti di parlato o di musica:

- 8 kHz (ovvero un campione ogni ottavo di millesimo di secondo) è una frequenza adeguata alla riproduzione del parlato;
- per riproduzione musicale di alta fedeltà la frequenza sale a 44,1 kHz.

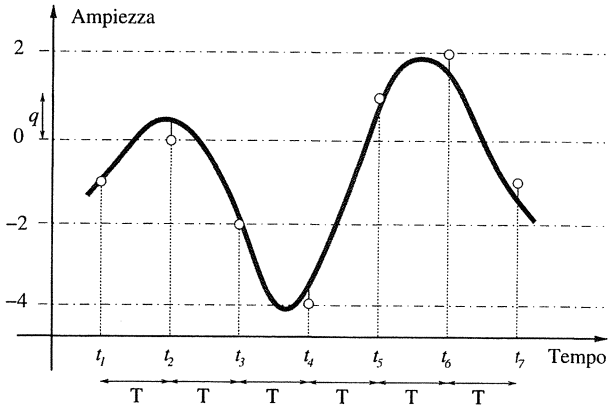


Figura A1.1 Campionamento e quantizzazione di un segnale analogico.

L'unità di quantizzazione è determinata dalla massima escursione di ampiezza ammessa e dal fatto che il valore di ampiezza di ogni campione è codificato con 16 bit (32 se stereo). Con questi dati, è facile rendersi conto di quanto sia impegnativa in termini di capacità di memoria la rappresentazione numerica dell'audio; per esempio, per un brano di 4 minuti di musica stereo d'alta fedeltà occorrono $44.100 \times 240 \times 32$ bit $\approx$ 40 Mbyte.

La *codifica MIDI* (*Musical Instruments Digital Interface*) è molto più parsimoniosa, grazie al fatto che non rappresenta il suono bensì l'informazione per la sua produzione da un sintetizzatore, per esempio: strumento emulato, altezza della nota (o del gruppo di note costituenti un accordo), durata. In alternativa, quando la qualità del suono sintetizzato non è soddisfacente, si può fare ricorso a tecniche di compressione audio (si veda l'Approfondimento applicativo 1.3).

La rappresentazione numerica di immagini è basata su concetti simili, ma il campionamento è nello spazio invece che nel tempo. Più precisamente, viene campionata una regione limitata bidimensionale, di norma rettangolare, applicandovi una griglia regolare per la selezione dei punti del dominio, ciascuno dei quali determina un elemento, detto *pixel* (*picture element*), dell'immagine discretizzata. Il campionamento spaziale può essere visto come un campionamento temporale sequenzializzando la matrice dei pixel, per esempio da sinistra verso destra e dal basso verso l'alto. La quantizzazione dipende dal tipo cromatico dell'immagine, poiché se ne quantizza la luminanza in un insieme finito di livelli di grigio oppure, nel caso di immagine a colori, una simile quantizzazione viene effettuata per ciascuno di tre componenti fondamentali del colore, per esempio RGB (*Red, Green, Blue*). La *risoluzione* genericamente indica la precisione del campionamento dell'immagine, ma è espressa da grandezze diverse in contesti diversi, per esempio:

- dimensioni della matrice, in numero di pixel (o colonne $\times$ righe, per esempio 1.024×768);
- densità dei pixel, per esempio pixel/mm, in ciascuna direzione spaziale del campionamento.

La precisione della quantizzazione è determinata dal numero di bit impiegati per il livello di grigio o per il colore di ciascun pixel; nel secondo caso è anche detta *profondità di colore*. Se non si applicano tecniche di compressione (si veda l'Approfondimento applicativo 1.3), l'immagine discreta è detta in formato *bitmap* e impegna molta memoria. Per esempio, un'immagine a colori con risoluzione 1.024×768 e 24 bit di profondità di colore richiede 2,25 Mbyte. Per immagini con caratteristiche particolari esistono però tecniche molto più parsimoniose. Per la grafica vettoriale, per esempio, la rappresentazione SVG (*Scalable Vector Graphics*), che descrive come riprodurre l'immagine usando componenti vettoriali, è non solo molto più compatta della rappresentazione bitmap ma anche, come dice il nome, più conveniente per il suo cambiamento di scala, ovvero per il ridimensionamento proporzionale dell'immagine.

Un'immagine animata o una sequenza video sono sequenze temporali di immagini. Concettualmente, la loro rappresentazione binaria è una sequenza temporale di rappresentazioni binarie di immagini. In pratica però è assolutamente necessario impiegare opportune tecniche di compressione, che sfruttino utili specificità di tali sequenze, quali il fatto che immagini contigue nella sequenza differiscono poco (tranne nel caso, relativamente raro, di un cambio di scena), per abbattere di ordini di grandezza la lunghezza della codifica binaria. Infatti, se per esempio si considera una sequenza video costituita da immagini bitmap quali quella indicata sopra, con 24 immagini al secondo (frequenza richiesta per la percezione visiva di continuità del movimento), occorrerebbero 54 Mbyte per ogni secondo di durata del video. Pur abbassandone risoluzione e profondità di colore, a scapito della qualità di riproduzione, la larghezza di banda necessaria resterebbe proibitiva. Nell'Approfondi-

mento applicativo 1.3 si introducono principi alla base di tecniche di compressione generiche, applicabili a qualsiasi sequenza binaria, come pure di tecniche specifiche per la compressione di sequenze video.

Problema

Si abbia un archivio di immagini fotografiche digitali bitmap da 16 milioni di pixel ciascuna, a 3 byte/pixel di profondità di colore. Quante ne può ospitare una memoria da 2 GB senza compressione? (1 GB = 1.024 MB = 1.024 × 1.024 KB = 1.024 × 1.024 × 1.024 byte).

Approfondimento applicativo 1.3

Compressione di dati

La rappresentazione dell'informazione con sequenze binarie più brevi (si spera: si veda il problema sotto) è l'obiettivo delle tecniche di compressione di dati, per risparmiare spazio nella memorizzazione dei dati e tempo nella loro trasmissione. La compressione costa essa stessa spazio e tempo, tuttavia è spesso vantaggiosa.

Una prima classificazione distingue fra compressione *senza perdita* di informazione (*lossless*) e *con perdita* (*lossy*), di solito entro limiti accettabili. In generale, una tecnica di compresssione consta di due algoritmi correlati, uno per la *codifica* compressa $c(s)$ di una data sequenza binaria s, l'altro per la *decodifica* $d(t)$ di una sequenza compressa t, ovvero per il recupero dell'informazione originale, o di una sua approssimazione nel caso di compressione con perdita. La compressione senza perdita è dunque caratterizzata dall'identità $d(c(s)) = s$, mentre per quella con perdita l'eguaglianza va sostituita da una relazione di approssimazione, eventualmente dotata di opportuna misura.

Una seconda classificazione distingue fra tecniche di compressione *generiche*, applicabili a sequenze binarie qualsiasi, e tecniche *specifiche*, progettate per particolari tipi di informazione, per esempio dati audio, immagini, sequenze audiovisive (AV). Nel seguito si focalizzerà l'attenzione su principi di funzionamento di tecniche generiche e si indicheranno i principali standard, basati su tecniche specifiche, per la compressione di dati multimediali.

In tutti i casi, si definisce il *rapporto di compressione* di una codifica c per una data sequenza binaria s come il rapporto fra la lunghezza di s e quella di $c(s)$, pertanto maggiore di 1 se e solo se la compressione della sequenza s effettivamente produce una sequenza binaria più breve.

Tecniche di compressione generiche

Sebbene siano applicabili a qualsiasi sequenza binaria, la loro efficacia dipende molto dal tipo di informazione rappresentata. Si distinguono i seguenti tipi di *codifica di compressione*.

Run-length

L'idea consiste nel codificare il numero di occorrenze consecutive di ciascun simbolo, risparmiando così la codifica delle sue eventuali successive occorrenze consecutive. Ecco alcuni esempi di casi in cui risulta vantaggiosa: compressione di matrici numeriche *sparse* (cioè, per definizione, a valori non nulli in pochi elementi), di audio con lunghe pause di silenzio, di immagini con ampie zone cromaticamente uniformi.

Differenziale, o relativa

Si spezza la sequenza da comprimere in blocchi e, codificato il primo, per i successivi si codificano le differenze del blocco relativamente al precedente. È particolarmente efficace su dati AV.

Variable-length

Si adotta un codice a lunghezza variabile per la rappresentazione binaria dei simboli, dove la lunghezza del codice di ciascun simbolo è inversamente proporzionale alla sua probabilità, stimata dalla sua frequenza in un *corpus* di testi rappresentativo. L'idea è antica, per esempio fu applicata nel progetto del codice Morse per la trasmissione telegrafica, ottimizzato per la lingua inglese. I codici di Huffman sono esempi più moderni.

Basata su dizionario, adattativa

Si codificano riferimenti (numerici) a un dizionario predefinito (quale quello per il controllo ortografico in un elaboratore di testi) o, in quella adattativa, dinamicamente costruito sulla stessa sequenza da codificare. Per esempio, l'algoritmo LZW (così detto dalle iniziali dei suoi ideatori: Lempel, Ziv e Welsh) di compressione generica adattativa, pubblicato nel 1984, è un raffinamento di algoritmi simili precedenti. Fu adottato in programmi di utilità, fra i quali compress in sistemi Unix di vecchia data, ma nel corso del tempo questi hanno ceduto il posto ad altre tecniche dello stesso tipo, fra cui gzip, bzip2, rzip, che vantano rapporti di compressione mediamente migliori.

Compressione di dati multimediali

Le suddette tecniche generiche sono tutte senza perdita e possono venire combinate fra loro e con altri metodi per ottenere tecniche di compressione specifiche, possibilmente con perdita, appositamente progettate (e standardizzate) per dati multimediali. Nel seguito sono brevemente indicati alcuni fra gli *standard* di compressione più diffusi, per tre tipologie di informazione multimediale.

Compressione audio

- MP3: compressione di dati musicali, vedi oltre;
- OGG/VORBIS: più efficace di MP3 a pari qualità, ma meno diffuso;
- *Speex*: molto efficace per la compressione del parlato.

Compressione di immagini

- GIF: *Graphical Interchange Format*;
- PNG: *Portable Network Graphics*;
- JPEG: *Joint Photographic Experts Group*.

Compressione AV

Il *Motion Picture Experts Group* (MPEG) è l'organismo di normativa tecnica responsabile degli standard più diffusi a tal fine. Questi sono definiti su più *livelli* (*layer*) di dati multimediali. Le tecniche attualmente in uso combinano:

- per la compressione video: ~JPEG + codifica relativa;
- per la compressione audio: MP3 = MPEG-1 Layer 3.

Problema

Sarebbe vantaggioso disporre di una tecnica di compressione generica senza perdita che offrisse un rapporto di compressione maggiore di 1 per qualsiasi sequenza binaria da comprimere. Una siffatta tecnica non può esistere: perché?

1.6 Considerazioni di efficienza

La misura più significativa per valutare l'efficienza o prestazione del calcolatore è espressa dalla velocità con cui esso è in grado di eseguire il programma; i due fattori principali che nel calcolatore determinano la velocità di esecuzione sono dati dalla struttura dei circuiti componenti e dall'articolazione del linguaggio macchina. Inoltre, siccome di solito il programma viene scritto in linguaggio di alto livello, la prestazione del calcolatore è condizionata anche dal compilatore impiegato per tradurlo in linguaggio macchina. Ne consegue che per aumentare l'efficienza del calcolatore, ovvero per accelerarne le operazioni, bisogna progettare in modo coordinato l'*insieme* (o *repertorio*) delle istruzioni macchina, i relativi compilatori e la struttura del calcolatore. Non è lo scopo del libro spiegare come si progetta un buon compilatore; qui invece si illustrerà il progetto dell'insieme di istruzioni macchina e dei circuiti in grado di eseguirle.

Comunque il Capitolo 4 fornisce una panoramica del software, includendo una discussione sul ruolo di compilatori e sistemi operativi. Questo libro si concentra sulla progettazione dell'insieme di istruzioni, assieme a memoria, processore e hardware di I/O, e sull'organizzazione di calcolatori sia piccoli sia grandi. Il Paragrafo 1.2.2 descrive come le cache possono migliorare le prestazioni della memoria. Alcuni aspetti delle prestazioni dell'insieme di istruzioni sono discussi nel Capitolo 2. Il paragrafo che segue fornisce una panoramica di come le prestazioni siano influenzate dalla tecnologia, come pure dal processore e dall'organizzazione del sistema.

1.6.1 Tecnologia

La tecnologia di integrazione a scala molto grande, o VLSI (*Very Large Scale of Integration*), che è usata per costruire i circuiti elettronici per un processore su un singolo chip, è un fattore critico per la velocità di esecuzione delle istruzioni macchina. La velocità per commutare tra gli stati 0 e 1 nei circuiti logici è largamente determinata dalla dimensione dei transistori che realizzano i circuiti. Più piccoli sono i transistori, più veloce è la commutazione. I progressi della tecnologia di costruzione attraverso varie decadi hanno ridotto notevolmente la dimensione del transistore. Questo ha due vantaggi: le istruzioni possono essere eseguite più velocemente e più transistori possono essere inseriti in un chip, portando ad avere più funzionalità logica e più capacità di memoria.

1.6.2 Parallelismo

Le prestazioni possono essere aumentate attraverso l'esecuzione parallela di un certo numero di operazioni. Il parallelismo può essere realizzato su vari livelli differenti.

Parallelismo a livello di istruzione

Il modo più semplice per eseguire una sequenza di istruzioni in un processore è di completare tutti i passi per l'istruzione corrente prima di iniziare i passi per la prossima istruzione. Se noi sovrapponiamo l'esecuzione dei passi di istruzioni successive, il tempo totale di esecuzione sarà ridotto. Per esempio, la prossima istruzione potrebbe essere prelevata dalla memoria e allo stesso tempo un'operazione aritmetica potrebbe essere eseguita sui registri operandi dell'istruzione corrente. Questa forma di parallelismo è chiamata *pipelining* (*elaborazione a stadi*). Essa è discussa in dettaglio nel Capitolo 6.

Processori multicore

Unità di elaborazione multiple possono essere costruite su un singolo chip. Nella letteratura tecnica, il termine *core* è usato per ciascuno di questi processori. Quindi il termine processore

è usato per il chip complessivo. Di conseguenza avremo la terminologia processori *dual-core*, *quad-core* e *octo-core* per chip che hanno due, quattro e otto core, rispettivamente.

Multiprocessori

I calcolatori possono contenere molti processori, in cui ognuno possibilmente contiene più core. Tali sistemi sono chiamati *multiprocessori*. Questi sistemi eseguono un numero di differenti applicazioni in parallelo, o eseguono parti di una singola applicazione in parallelo. Tutti i processori in genere hanno accesso a tutta la memoria in tali sistemi e il termine *multiprocessore a memoria condivisa* (*shared-memory multiprocessor*) è spesso usato per esplicitare questo fatto. Le alte prestazioni di questi sistemi sono accompagnate da complessità e costi molto alti, dovuti all'uso di tanti processori e unità di memoria, insieme a una rete di interconnessione più complessa.

In contrasto con i sistemi multiprocessore, è possibile usare un gruppo di calcolatori completi collegati per ottenere un'alta potenza totale di elaborazione. I calcolatori di solito hanno accesso solo alle loro unità di memoria. Quando i compiti che essi eseguono necessitano di condividere dati, i calcolatori scambiano *messaggi* su una rete di comunicazione. Questa proprietà li distingue dai multiprocessori a memoria condivisa, portando al nome di *multicalcolatori a scambio di messaggi* (*message-passing multicomputer*).

I multiprocessori e i multicalcolatori sono descritti nel Capitolo 12.

1.7 Breve storia del calcolatore

Il calcolatore elettronico attuale è il risultato del rapido progresso tecnologico verificatosi negli ultimi 60 anni. Tuttavia la comparsa del calcolatore elettronico è stata preceduta dallo sviluppo di dispositivi di calcolo meccanici, molto più lungo e lento. Per la storia del calcolatore, con datazione, presupposti scientifici e tecnologici, biografia degli inventori, storia dei progetti di ricerca e dei produttori, si rimanda alla bibliografia e in particolare a [1]; qui se ne dà solo una sintesi brevissima, che illustra i passi fondamentali nello sviluppo della tecnologia delle macchine capaci di calcolare in modo automatico. Per la storia del concetto di calcolo e dell'informatica teorica si rimanda interamente alla letteratura relativa.

Premesse e precursori

Dispositivi meccanici atti a facilitare il calcolo, specialmente in applicazioni di astronomia, esistettero già a partire dall'antichità classica, ma erano rari e oggi di essi si sa poco, pertanto si rimanda alla letteratura specializzata. Essi comunque non diedero origine a progressi tecnologici rilevanti, destinati a diffondersi e durare; non si ebbero mutamenti sostanziali nemmeno nel medioevo e nel rinascimento.

Nell'arco dei circa tre secoli intercorsi prima della metà del XX secolo (vale a dire circa dal 1650 al 1950), dapprima in Europa e in seguito anche in America venne ideata ed entrò lentamente in uso una serie di dispositivi meccanici di complessità crescente, costruiti partendo da elementi di base come ingranaggio, leva, puleggia e svariati altri, che si usavano per calcolare operazioni aritmetiche come addizione, sottrazione, moltiplicazione e divisione. Inizialmente essi non erano programmabili, ma in seguito come dispositivo di ingresso si usò la scheda o il nastro di carta perforato: i fori venivano rilevati meccanicamente e la loro disposizione codificava il programma, cioè la lista delle operazioni aritmetiche da eseguire, e i dati su cui lavorare; il dispositivo di calcolo possedeva dunque una sia pur limitata programmabilità. I risultati venivano emessi tramite lo stesso sistema, scheda o nastro perforato, oppure stampati su carta. Calcolatori meccanici (o calcolatrici) di tale genere vennero impiegati a più riprese per calcolare, con successo, la tavola dei logaritmi ($\log_{10}$ e $\log_e$), quella delle funzioni goniometriche circolari e iperboliche (sin, cos, tan, sinh, cosh, tanh ecc.) e di altre funzioni ancora, ottenendole tramite approssimazione poli-

nomiale. Tra i precursori coinvolti in questi sviluppi vale la pena di citare i filosofi e matematici francese Blaise Pascal e tedesco Gottfried Wilhelm von Leibniz, attivi proprio all'inizio del periodo, ma ce ne sono parecchi altri ancora e di alcuni di loro si dirà più avanti.

Nell'ultima parte di tale periodo (circa negli ultimi 50 anni, dal 1900 al 1950) si cominciarono ad aggiungere elementi elettrici al calcolatore, ottenendo così calcolatori elettromeccanici: motore elettrico per azionare il dispositivo, *relais* elettrico per calcolare funzioni logiche, dello stesso tipo già in uso nelle centrali telefoniche elettromeccaniche dell'epoca e altri elementi ancora. La memoria rimaneva però realizzata in modo puramente meccanico, tramite il posizionamento di aghi, levette o ingranaggi, e l'esecuzione di calcoli ne comportava la messa in movimento, ciò che qualificava tali dispositivi come ancora essenzialmente di tipo meccanico. Verso la fine della sua storia il calcolatore elettromeccanico era ormai diventato una macchina programmabile in modo abbastanza flessibile, sia pure con difficoltà e limitazioni, e capace di eseguire parecchie operazioni aritmetiche al secondo (nell'ordine della decina al secondo nei casi migliori), ma costoso, ingombrante, rumoroso e poco affidabile. Nondimeno tali macchine conobbero un certo successo scientifico e commerciale; vale la pena di citare il calcolatore elettromeccanico (*Harward machine*) progettato dallo statunitense Howard Aiken, come anche quelli progettati dal tedesco Konrad Zuse poco prima della II Guerra Mondiale (la quale mise fine ai suoi progetti), ma ne esistono svariati altri.

Nascita e primi passi

La storia del calcolatore elettronico iniziò durante la II Guerra Mondiale (1939-1945), quando il primo di essi pienamente operativo venne ideato e costruito presso l'Università della Pennsylvania, USA, da parte del matematico John von Neumann e degli ingegneri Presper Eckert e John Mauchly[2] (tutti e tre statunitensi): si basava sull'uso della tecnologia elettronica della *valvola a effetto termoionico* o *tubo a vuoto* (*vacuum tube*), già esistente e impiegata, per esempio, nella tecnologia delle radio militari e in quella del radar. La valvola termoionica è un dispositivo puramente elettrico (non ha parti in movimento) e si può usare sia per calcolare operazioni logiche e aritmetiche sia per realizzare la memoria; essa ha dunque dato inizio allo sviluppo del calcolatore elettronico odierno, in cui la meccanica è assente o ha un ruolo ausiliario, anche se in seguito in tale uso la valvola termoionica è stata sostituita da altre tecnologie, segnatamente dal transistore.

Il complesso di idee costituenti questo primo modello di calcolatore elettronico è detto "*macchina di von Neumann*", ma tale termine attribuisce il merito della scoperta in modo esclusivo a J. von Neumann, oscurando i contributi pure decisivi degli altri inventori e trascurando il fatto che precursori concettuali del modello sono già concepiti nei decenni precedenti, dal lavoro dei britannici Charles Babbage[3] e Lady Ada Lovelace[4] (figlia del poeta Lord George Byron) compiuto intorno alla metà del XIX secolo alla fondamentale *Macchina di Turing* degli anni 30 del XX secolo, un modello teorico di calcolatore universale. Essenzialmente il modello di von Neumann prevede la presenza di un solo processore, un po' di memoria centrale e un solo bus.

Si suole dividere la storia dello sviluppo del calcolatore elettronico odierno in quattro fasi o *generazioni*, caratterizzate dalla comparsa di idee e tecnologie importanti per la realizzazione di processore, memoria e unità di I/O. La prima generazione va dal 1945 al 1955, la seconda dal 1955 al 1965, la terza dal 1965 al 1975 e la quarta dal 1975 a oggi; naturalmente è una suddivisione un po' approssimativa, soprattutto rispetto alla quarta generazione, al cui interno si potrebbero identificare nuove generazioni ogni 5-10 anni in base al ritmo di sviluppo delle innovazioni più significative.

1.7.1 Prima generazione

L'idea di programma da tenere in memoria ed eseguire tramite il processore venne introdotta compiutamente e resa nota dal matematico statunitense John von Neumann negli anni 40 del XX secolo.

Secondo il modello illustrato allora da von Neumann (e frutto della collaborazione con diversi altri ricercatori), il programma e i dati su cui lavorare erano contenuti nella medesima unità di memoria, come del resto si fa ancora oggi. Il programma veniva scritto in *linguaggio macchina simbolico* o *linguaggio assemblativo* (*assembly language*) direttamente da parte del programmatore e poi tradotto tramite l'*assemblatore* (*assembler*) un altro programma dedicato a tale funzione, in *linguaggio macchina binario*, eseguibile direttamente da parte del calcolatore.

La tecnologia impiegata era quella della valvola termoionica, che permette di calcolare le operazioni logiche e aritmetiche di base in un tempo dell'ordine del millisecondo ($1\ \text{ms} = 10^{-3}\ \text{s}$). In tale modo si conseguì un aumento di velocità pari a un fattore che andava da 100 a 1.000 rispetto alla velocità del calcolatore elettromeccanico. I registri venivano anch'essi realizzati tramite la valvola termoionica, ma quasi da subito la memoria centrale venne realizzata anche tramite la tecnologia della *linea di ritardo a mercurio*,[5] mentre le unità di I/O si basavano sulla tecnologia della *scheda* e del *nastro perforati*, o qualcosa di simile. Vennero ideate e verso la fine del periodo cominciarono a entrare in uso la memoria centrale a *nuclei magnetici* e la memoria di massa a *tamburo magnetico*.[6]

1.7.2 Seconda generazione

L'effetto rettificante della giunzione bipolare a semiconduttori venne scoperto (in modo fortuito, a seguito di esperimenti con altri fini) presso i Laboratori Bell verso la fine del 1940

---

[2] Analoghi sviluppi, seppure un po' più limitati a causa della guerra, si ebbero nello stesso periodo anche in Gran Bretagna per opera di Alan Turing e in seguito di Maurice Wilkes; vi sono peraltro parecchi altri ricercatori statunitensi e di diversi paesi coinvolti a vario titolo nell'invenzione del calcolatore elettronico moderno o di parti funzionanti di esso, in quel periodo o poco prima (per esempio ancora gli statunitensi John Atanasoff e Clifford Berry); si veda comunque [1] al riguardo.

[3] Progettista della "*Macchina Analitica*" (*Analytical Engine*), un calcolatore meccanico programmabile che egli definì in modo completo ma che nel suo tempo non potè costruire per mancanza di tecnologie adeguate a costo accessibile; un prototipo venne infine costruito dopo la sua morte (grazie agli appunti e articoli della contessa di Lovelace, pure già scomparsa) agli inizi del XX secolo, quando era ormai un modello superato, per dimostrare retrospettivamente la validità delle sue idee: funziona perfettamente e oggi è esposto al British Museum.

[4] Ella tradusse dal francese in inglese (e poi pubblicò), una descrizione della macchina analitica di C. Babbage (del quale era collaboratrice e grande amica) aggiungendovi di suo pugno svariati commenti e note oggi riconosciuti come la primissima definizione chiara e consapevole (benché non completa) del concetto, essenziale, di macchina di calcolo *programmabile*.

[5] Oggi del tutto scomparsa: l'idea è di fare circolare in successione i bit della parola di memoria sotto forma di un treno di impulsi ultrasonici lungo un tubo (o più di uno) di circa un metro riempito con mercurio liquido; gli impulsi sono molto corti e si propagano nel mercurio con un certo ritardo, pertanto il tubo ne può contenere parecchi circolanti in rapida successione, fino a qualche migliaio; si fa ricircolare il treno di impulsi e si leggono e scrivono impulsi mediante un circuito elettrico ausiliario contenente un generatore e un rilevatore di ultrasuoni; tutto ciò oggi appare bizzarro, ingombrante e costoso, ma allora no e comunque funziona; la memoria a tubo di mercurio scomparve quando apparve quella a nuclei magnetici.

[6] Pure scomparsa, sostituita dal disco magnetico; era costituita da un cilindro rotante (o più d'uno), di altezza alquanto superiore al raggio (cioè lungo e sottile), ricoperto di materiale ferromagnetico, equipaggiato di schiera di testine di lettura-scrittura disposte in posizione fissa lungo una o più generatrici del cilindro; lettura e scrittura funzionavano già come poi nel disco magnetico.

(da parte di Shottky e collaboratori) e nello stesso luogo subito dopo venne ideato il *transistore a giunzione bipolare* (*bipolar junction transistor*, BJT), che si basa su tale effetto. La tecnologia del transistore BJT sostituì rapidamente (in effetti, in circa 15 anni) quella della valvola termoionica ed è proprio tale mutamento tecnologico quello che delimita il passaggio tra la prima e la seconda generazione. La tecnologia di memoria centrale a nuclei magnetici e quella di memoria di massa a tamburo magnetico divennero di uso generale e si perfezionarono sensibilmente. Furono ideati i *linguaggi di alto livello* (il primo fu il Fortran, 1956), che semplificavano e potenziavano enormemente la programmazione; naturalmente comparvero anche i *compilatori*, che traducevano il programma scritto in linguaggio di alto livello in linguaggio macchina simbolico, che a sua volta veniva tradotto in linguaggio macchina binario da parte del programma assemblatore. Si introdussero processori di I/O separati che funzionavano in parallelo al processore principale dedicato all'esecuzione del programma, sollevandolo dal compito di gestire i trasferimenti di I/O con effetto positivo sull'efficienza del calcolatore. In quel periodo la IBM (*International Business Machine*) divenne il maggiore produttore mondiale di calcolatori.

1.7.3 Terza generazione

Verso i primi anni '60 fu sviluppato il *transistore laterale* (cioè realizzato su una superfice 2D di silicio, invece che in un blocco 3D) e in tale modo divenne possibile integrare numerosi transistori sulla stessa piastrina di silicio, o *chip*: comparve così il *circuito integrato* (*integrated circuit*, IC) e la relativa tecnologia, che consentì di produrre processori e memorie più potenti, capaci e veloci. La *memoria centrale elettronica* a circuiti integrati cominciò a sostituire quella a nuclei magnetici ed è proprio questo il mutamento tecnologico che delimita il passaggio dalla seconda alla terza generazione.[7]

Altri progressi compresero l'ideazione della *microprogrammazione*, del *parallelismo* e della metodologia di *pipelining*, di cui si è detto nel Paragrafo 1.6. Il sistema operativo era già comparso in forma rudimentale monoprogrammata, ma divenne multiprogrammato e ciò permise di eseguire più programmi sul medesimo calcolatore condividendone le risorse in modo efficiente. Comparvero anche la tecnologia di *memoria cache* e quella di *memoria virtuale*: la prima ha lo scopo di mostrare all'osservatore esterno una memoria centrale più veloce di quanto il tipo di transistore impiegato non consenta direttamente, mentre la seconda ha lo scopo di mostrarne una più grande di quanto non sia realmente. I calcolatori che dominavano il mercato erano i grandi *mainframe* System 360 e 370 di IBM e i minicalcolatori PDP di Digital Equipment Corporation (DEC, compagnia oggi assorbita da Hewlett Packard).

1.7.4 Quarta generazione

A partire dai primi anni '70 la tecnologia dei circuiti integrati progredì al punto da potere mettere l'intero processore e una parte significativa della memoria di un calcolatore non troppo grande su un solo circuito integrato. Si poterono integrare sul circuito alcune decine di migliaia di transistori, dando origine alla tecnologia VLSI. L'odierno livello raggiunto dalla densità di integrazione è dell'ordine dei miliardi di transistori su un chip. La legge di Moore asserisce che la densità di integrazione raddoppia all'incirca ogni 18 mesi. Si tratta solo di un'osservazione empirica, che prima o poi cesserà di valere, ma è stata sorprendentemente vera nei quattro decenni trascorsi. La tecnologia VLSI permetteva insomma di realizzare il processore

[7] La memoria a nuclei magnetici, con il suo peso enorme, fu sempre la "palla al piede" del calcolatore, finché esistette.

completo su circuito integrato singolo: così nacque il *microprocessore* (*microprocessor*). Furono fondate o si ingrandirono molte nuove compagnie trainanti in questo settore tecnologico innovativo: tra le principali statunitensi, Intel, National Semiconductor, Motorola, Texas Instruments (TI) e Advanced Micro Devices (AMD); in Italia la Società Generale Semiconduttori (SGS), poi divenuta ST Microelectronics (STM, in unione con la Thomson francese) e tuttora esistente. Oggi naturalmente diverse compagnie europee, recenti o storiche, e asiatiche producono anch'esse processori.

Un tipo particolare di tecnologia VLSI, chiamato FPGA (*field programmable gate array*, ovvero matrici di porte logiche programmabili sul campo), ha permesso agli sviluppatori di sistemi di progettare e realizzare processore, memoria e circuiti di I/O su un singolo chip per soddisfare i requisiti di specifiche applicazioni, specialmente nei sistemi incorporati. Strumenti avanzati di progettazione assistita dal calcolatore permettono di sviluppare velocemente prodotti basati su FPGA. Aziende come Altera e Xilins forniscono questa tecnologia, insieme ai sistemi necessari allo sviluppo del software.

Oggigiorno sistemi embedded, calcolatori portatili e telefoni cellulari versatili sono largamente diffusi. Calcolatori personali da tavolo e stazioni di lavoro, interconnessi da reti locali con o senza fili e da Internet, con accesso a server di basi di dati e motori di ricerca, forniscono una varietà di potenti piattaforme di calcolo.

Concetti organizzativi come parallelismo e memorie gerarchiche si sono evoluti per produrre i sistemi di calcolo ad alte prestazioni di oggi attraverso la maturazione della quarta generazione. I supercalcolatori e i calcolatori Grid, all'estremo superiore delle alte prestazioni, sono usati per le previsioni meteorologiche, le elaborazioni scientifiche e ingegneristiche, le simulazioni.

Sommario

La storia del calcolatore elettronico è ricca di sviluppi inattesi e interessanti, dove si alternano l'ideazione di concetti nuovi e la scoperta di tecnologie prima inesistenti. Nel capitolo sono stati considerati molti aspetti inerenti la struttura e il funzionamento del calcolatore elettronico ed è stata introdotta gran parte della terminologia scientifica e tecnica necessaria. Le istruzioni macchina e i programmi sono stati brevemente descritti. Sono state spiegate l'addizione e la sottrazione di numeri binari.

Nei capitoli successivi del libro si daranno le spiegazioni dettagliate intorno a tali termini e concetti, collocando così le varie parti del presente capitolo nella prospettiva giusta.

Note bibliografiche

1. Williams M., *A History of Computing Technology*, Wiley-IEEE Computer Society Press, 1997.

2 Istruzioni macchina

In questo capitolo

- Istruzioni macchina ed esecuzione di programmi
- Metodi di indirizzamento per accedere a operandi in registri e in memoria
- Linguaggio assemblativo per rappresentare istruzioni macchina, dati e programmi
- Pile e sottoprogrammi

Questo capitolo considera il modo in cui i programmi vengono eseguiti in un calcolatore dal punto di vista dell'insieme di istruzioni macchina. Nel Capitolo 1 si è introdotto il concetto generale che le istruzioni e i dati da elaborare si trovano in memoria centrale. Qui, invece, si illustrerà in termini generali il concetto di istruzione macchina: come è definita e come è acquisita e interpretata da parte del processore. Verranno pertanto descritte le tipologie fondamentali di istruzione e le modalità principali e più comuni per manipolare i dati, vale a dire come l'istruzione possa caricare o leggere dai registri i dati da elaborare e memorizzare o scrivere nei registri il risultato calcolato.

La trattazione qui svolta discute i concetti fondamentali inerenti le istruzioni macchina del processore. Non si fa riferimento ad alcun linguaggio macchina specifico o di uso commerciale, bensì a un linguaggio macchina generico, tuttavia contenente le tipologie di istruzione macchina e le modalità di manipolazione dei dati comuni alla maggior parte dei processori commerciali. Si introdurrà un numero sufficiente di istruzioni macchina generiche e di modalità di manipolazione dei dati, bastanti a scrivere programmi semplici ma completi.

Comunemente il complesso di tutte le istruzioni macchina riconosciute come proprie dal processore, il quale pertanto è capace di eseguirle, viene chiamato *insieme* o *repertorio di istruzioni macchina*, o ISA (*Instruction Set Architecture*). L'ISA non è soltanto un elenco di nomi di istruzioni macchina associati alle rispettive operazioni da svolgere, ma specifica anche le modalità di manipolazione dei dati e come esse si possano combinare con le varie istruzioni. Dato che il capitolo ha lo scopo di illustrare le capacità funzionali generali del processore qui non si descrive un'ISA completa e dettagliata, quale si potrebbe trovare in un processore commerciale. Piuttosto si darà un elenco di istruzioni e di esempi su come usarle in modo appropriato, abbastanza esteso da trattare tutte le situazioni rilevanti per fini pratici. Poiché è interessante e importante vedere come i concetti discussi siano implementati in un vero calcolatore, si aggiungeranno in questo e nel prossimo capitolo quattro esempi di processori commerciali diffusi. Questi processori, presentati in approfondimenti applicativi, sono: il processore NIOS II della Altera Corporation, il processore ColdFire della Freescale Semiconductor Inc., la famiglia di processori ARM della Advanced RISC Machines Ltd. e l'architettura di base IA-32 della famiglia di processori costruiti dalla Intel Corporation. Negli stessi approfondimenti applicativi si proporranno problemi di stesura di programmi generici dei due capitoli sulla base degli insiemi di istruzioni specifici di tali processori.

Il lettore può scegliere solo un processore e studiare il materiale corrispondente degli approfondimenti applicativi, per capire appieno la progettazione di un'ISA commerciale. Co-

munque la conoscenza del materiale di questi approfondimenti applicativi non è essenziale per la comprensione del materiale nella parte principale del libro.

La grande maggioranza dei programmi sono scritti in linguaggi di alto livello quali C, C++ o Java. Per eseguire un programma scritto in un linguaggio di alto livello su un processore, il programma deve essere tradotto nel linguaggio macchina per quel processore; questo è il lavoro svolto dal compilatore. Il linguaggio assemblativo è una rappresentazione simbolica leggibile del linguaggio macchina. In questo libro si fa un uso estensivo del linguaggio assemblativo, poiché questo è il modo migliore per descrivere il funzionamento dei calcolatori.

La discussione in questo capitolo inizia considerando come le istruzioni e i dati siano immagazzinati in memoria e come vi si acceda per la loro elaborazione.

2.1 Memoria del calcolatore

Durante l'esecuzione del programma, numeri, caratteri e istruzioni sono contenuti nella memoria del calcolatore, pertanto è necessario esaminarne in generale la struttura. Essa è costituita da milioni o miliardi di *celle* elementari, ciascuna delle quali è capace di contenere e mantenere l'informazione di un bit, cioè il valore 0 o 1. Siccome il bit rappresenta una quantità di informazione minima, di rado la cella da un bit viene usata individualmente; di solito una quantità prefissata di celle forma un gruppo e tutte le celle del gruppo vanno lette o scritte in parallelo come una sola operazione.

Il gruppo di $n \geq 1$ celle da un bit si chiama *parola di memoria* (*memory word*); il numero n di bit che la costituiscono si chiama *dimensione* o *lunghezza* della parola. La memoria del calcolatore è a sua volta formata da una *successione* o *vettore* di parole, come mostra la Figura 2.1.

I calcolatori odierni hanno dimensione di parola variabile a seconda del modello, tuttavia per lo più si usano lunghezze comprese nell'intervallo da 16 a 64 bit. Per esempio, un calcolatore caratterizzato da lunghezza di parola pari a 32 bit può contenere in una sola parola di memoria un numero naturale (o un numero in complemento a due) da $n = 32$ bit, oppure 4 caratteri ASCII, ciascuno da 8 bit (ovvero da un byte), come mostra la Figura 2.2. La rappresentazione del numero frazionario e dell'istruzione macchina è più complessa, ed essi possono occupare due o più parole; per quella del numero frazionario, introdotta nel Paragrafo 1.4.2, si veda il Paragrafo 9.7 per ulteriori dettagli, mentre nel Paragrafo 2.13 si discuterà come codificare in memoria l'istruzione, ma non prima di avere illustrato la struttura generale del linguaggio macchina.

Per leggere o scrivere in memoria un elemento di informazione, parola intera o byte, è necessario identificare la parola di memoria su cui si vuole lavorare, il che si ottiene associando a ciascuna parola un *indirizzo* (*address*) univoco. Per definizione l'indirizzo è un numero naturale (sempre positivo o al più nullo) e convenzionalmente si usano gli indirizzi compresi nell'intervallo $[0,2^m)$ per un valore fissato $m \geq 1$; pensando l'indirizzo come numero binario naturale la quantità m è dunque il numero di bit necessari per rappresentarlo. La corrispondenza tra indirizzo e parola di memoria è stabilita assegnando indirizzi consecutivi a parole consecutive, partendo dall'indirizzo 0. La quantità totale di indirizzi 2^m, vale a dire la quantità di parole che la memoria contiene, si chiama *spazio di indirizzamento* (*addressing space*). Per esempio, usando $m = 24$ bit per l'indirizzo si ha uno spazio di indirizzamento pari a $2^{24} = 16.777.216$ parole di memoria.

Giacché si deve lavorare con le potenze di 2, per sveltire la notazione conviene introdurre multipli opportuni. Ecco quelli di uso più frequente:

- *Kilo* = K = $2^{10} = 1024 \approx 10^3$ = mille;[1]
- *Mega* = M = $2^{20} = 1.048.576 \approx 10^6$ = un milione;

[1] Qui *Kilo* si scrive con iniziale maiuscola e si abbrevia in *K*, giacché differisce da *kilo* (sistema metrico decimale), seppure di poco; i multipli successivi si scrivono come in MKS.

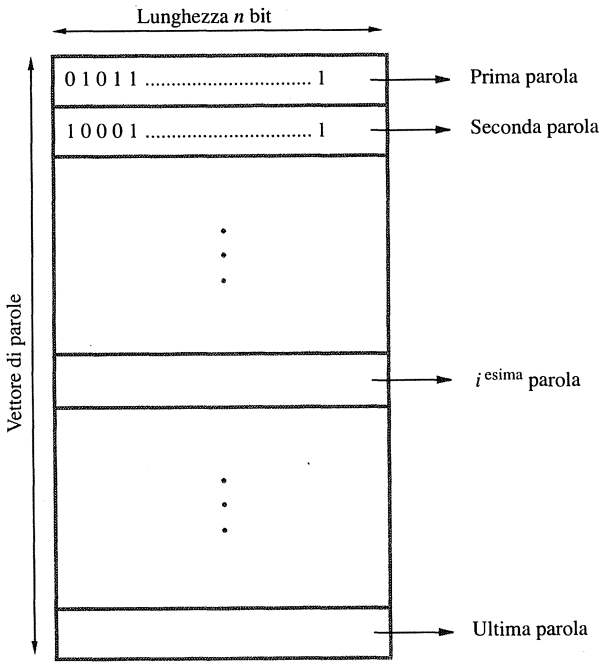


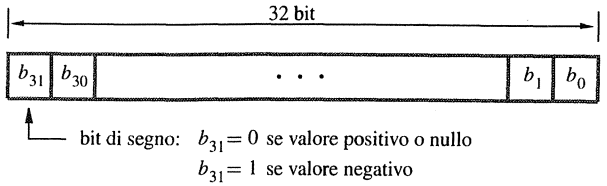
Figura 2.1 Modello generico di processore a registri.

- *Giga* = G = $2^{30} = 1.073.741.824 \approx 10^9$ = un miliardo;
- *Tera* = T = $2^{40} = 1.099.511.627.776 \approx 10^{12}$ = mille miliardi.

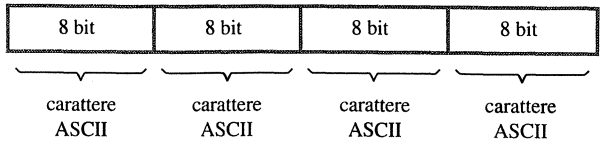
Questi termini sono usati come prefissi, per esempio si ha 1 Kilobyte = 2^{10} byte, 1 Megabyte = 2^{20} byte ecc. Si tenga presente che Kilobyte, Megabyte, Gigabyte e Terabyte si abbreviano in KB, MB, GB e TB (con "B" maiuscola), rispettivamente, mentre Kb (con "b" minuscola) è abbreviazione di Kilobit (cioè 1024 bit, non byte) e similmente per i multipli successivi. Tali multipli hanno gli stessi nomi in uso nel sistema metrico decimale (MKS) e anche quasi gli stessi valori (sono leggermente superiori ma l'errore è inferiore a 10% almeno fino al multiplo Tera, poi aumenta; basta osservare che la seconda cifra più significativa in tutti i casi è 0), permettendo di valutare in modo rapido l'ordine di grandezza di potenze di 2 con esponente elevato, con approssimazione per difetto.

Viceversa, per convertire in base 2 una potenza di 10 basta riportarla ai multipli di prima. Per esempio, $10^{15} = 10^{3+12} = 10^3 \times 10^{12} \approx \text{K T} = 2^{10} \times 2^{40} = 2^{10+40} = 2^{50}$ e similmente. Oppure si può ricordare che $10^3 = 1000 \approx 1024 = 2^{10}$, da dove si ha $10 \approx 2^{10/3}$ e generalizzando per $k \geq 1$ si ha $10^k \approx 2^{\lceil 10k/3 \rceil}$.[2] Dunque con $k = 15$ si ha $10^{15} \approx 2^{\lceil 10 \times 15/3 \rceil}$ e similmente; l'arrotondamento di $10k/3$ introduce un errore per difetto.

[2] I simboli "⌈" e "⌉" si chiamano *parentesi di Gauss*; $\lceil a \rceil$ lascia immutato l'argomento a se è già intero, mentre se non lo è lo arrotonda all'intero immediatamente superiore: $\lceil 2 \rceil = 2$ e $\lceil 2,1 \rceil = 3$, ma attenzione a $\lceil -2,1 \rceil = -2$, perché $-2 > -2,1$. Volendo completare: sono parentesi di Gauss anche "⌊" e "⌋", così



(a) parola di memoria da 32 bit contenente un numero intero relativo rappresentato in complemento a due con $n = 32$ bit



(b) parola di memoria da 32 bit contenente quattro caratteri ASCII

Figura 2.2 Esempio di elemento di informazione codificato in una parola di memoria da 32 bit.

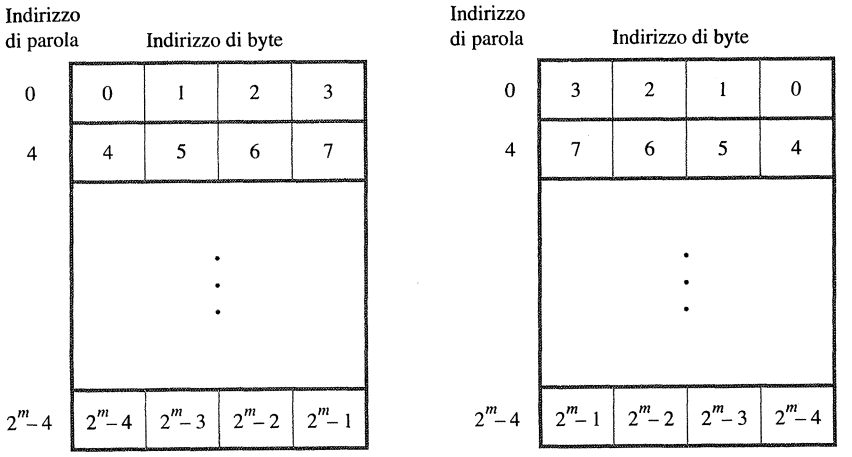
2.1.1 Indirizzamento di byte

Si è visto come il calcolatore lavori con tre elementi base di informazione: il bit, il byte (ovvero la stringa di 8 bit), e la parola, che varia da 16 a 64 bit secondo il modello di calcolatore. È poco pratico assegnare un indirizzo a ciascun bit della memoria, pertanto di norma si assegnano indirizzi consecutivi a byte consecutivi. Pressoché tutti i calcolatori odierni adottano il byte come *unità minima di informazione indirizzabile* in memoria, e tale è anche la convenzione che verrà usata nel resto del libro: si dice allora che la memoria è *indirizzabile per byte* (*byte addressable*). Dunque a byte consecutivi si assegnano indirizzi consecutivi 0, 1, 2, 3, ..., e ne consegue che la parola di memoria, che è formata da un numero intero di byte, debba avere *indirizzo multiplo della rispettiva lunghezza in byte*. Per esempio, se la parola ha lunghezza pari a 32 bit, ovvero 4 byte, a parole consecutive corrispondono indirizzi multipli di 4, vale a dire 0, 4, 8, 12, ...

2.1.2 Ordinamento di byte

Vi sono due modi diversi per assegnare indirizzi a byte appartenenti alla stessa parola di memoria: l'indirizzamento *crescente* (*big-endian*) e quello *decrescente* (*little-endian*); la Figura 2.3 li mo-

esemplificate: $\lfloor 2 \rfloor = 2$ e $\lfloor 2{,}1 \rfloor = 2$, ma di nuovo attenzione a $\lfloor -2 \rfloor = -3$ perché $-3 < -2{,}1$; tali coppie di parentesi vengono anche chiamate funzione *tetto* ("$\lceil$", "$\rceil$") e *base* ("$\lfloor$", "$\rfloor$"), rispettivamente; con $[a]$ si indica la *parte intera* di a, così definita: $[a] = \lfloor a \rfloor$ se $a \geq 0$ e $[a] = \lceil a \rceil$ se $a < 0$; esempi: $[2] = \lfloor 2 \rfloor = 2$, $[-2] = \lceil -2 \rceil = -2$, $[2{,}1] = \lfloor 2{,}1 \rfloor = 2$ e $[-2{,}1] = \lceil -2{,}1 \rceil = -2$; insomma la parte intera di a è la base se a è positivo (o nullo), se negativo è il tetto; infine $pf(a) = |a - [a]|$ è la *parte frazionaria* di a (mai negativa). Ecco alcuni esempi: $pf(2) = 0$, $pf(-2) = 0$, $pf(2{,}1) = 0{,}1$ e $pf(-2{,}1) = 0{,}1$ (il lettore faccia il conto da sé).



(a) schema di indirizzamento di byte *crescente* o *big-endian*

(b) schema di indirizzamento di byte *decrescente* o *little-endian*

Figura 2.3 Schema di indirizzamento di byte e di parola.

stra entrambi. Nell'indirizzamento crescente l'indirizzo aumenta al diminuire del peso aritmetico del byte, ovvero spostandosi verso il bit meno significativo, mentre in quello decrescente la corrispondenza tra indirizzo e byte segue l'ordine inverso. Entrambi gli schemi di indirizzamento di byte si usano nei calcolatori commerciali, a seconda del modello. Lo schema di indirizzamento di parola non dipende però da quello adottato per il byte; per esempio, se la parola è da 32 bit l'indirizzo di parola è comunque un multiplo di 4: 0, 4, 8, 12, ..., $2^m - 4$, indipendentemente dall'indirizzo di byte.

Oltre a specificare l'ordine di indirizzamento del byte all'interno della parola, è necessario precisare anche l'ordine di enumerazione dei bit all'interno sia di byte sia di parola. Per la parola la convenzione adottata più frequentemente è quella mostrata in Figura 2.2a, e verrà usata anche nel resto del libro; si tratta, del resto, dell'ordinamento più naturale per la maggior parte degli esseri umani, perché coincide con l'attribuzione del peso aritmetico. Si usa lo stesso schema di enumerazione anche per i bit all'interno del byte, ovvero b_7, b_6, ..., b_1, b_0 procedendo da sinistra verso destra. Comunque esistono anche modelli di calcolatore dove si adottano le convenzioni opposte.

2.1.3 Allineamento di parola

Considerando parole di memoria da 32 bit, caso molto frequente, i bordi di parole consecutive sono collocati agli indirizzi 0, 4, 8, 12, ..., come mostra la Figura 2.3. Quando la transizione tra una parola e quella consecutiva cade a un indirizzo multiplo della lunghezza di parola misurata in byte, si dice che lo schema di indirizzamento di parola è di tipo *allineato* (*aligned addressing*). Come si è già accennato, per ragioni pratiche la lunghezza di parola misurata in bit è sempre, o quasi, una potenza di 2; pertanto, se per esempio la parola è da $2^4 = 16$ bit lo schema di indirizzamento di parola allineato prevede che gli indirizzi di parola siano 0, 2, 4, 6, ..., mentre se fosse da $2^6 = 64$ bit $= 8$ byte dovrebbero essere 0, 8, 16, 24, ...

Tuttavia non ci sono motivi di fondo per cui la parola non debba iniziare a un byte con indirizzo scelto in modo arbitrario; si dice allora che lo schema di indirizzamento di parola è di tipo *disallineato* (*unaligned addressing*). Per esempio, considerando la Figura 2.3a, la parola da $2^5 = 32$ bit formata dai 4 byte di indirizzo 2, 3, 4, 5 avrebbe indirizzo 2 e dunque disallineato, giacché non sarebbe un multiplo di 4, la dimensione della parola in byte. Tutti i calcolatori dispongono dello schema di indirizzamento allineato e alcuni, facoltativamente, anche di quello disallineato, più complesso circuitalmente senza però presentare vantaggi decisivi. Nel resto del libro non si avrà occasione di farne uso.

2.1.4 Dati numerici e testuali

Il numero intero (naturale o relativo) di solito occupa esattamente una parola di memoria e se ne specifica la posizione dandone l'indirizzo di parola; similmente, si specifica la posizione in memoria del carattere dandone l'indirizzo di byte. Per convenienza durante la programmazione è utile avere modi diversi nelle istruzioni del programma per specificare gli indirizzi. Il Paragrafo 2.4 tratterà questo argomento.

2.2 Operazioni di memoria

Durante l'esecuzione sia le istruzioni componenti il programma sia i dati su cui esse lavorano sono contenuti in memoria codificati sotto forma di parola (una o più) e i circuiti di controllo del processore pilotano il trasferimento della parola dalla memoria al processore, o viceversa se si tratta del risultato. Pertanto le due operazioni base di memoria sono le seguenti: *prelievo* (*fetch*) o *caricamento* (*load*); e *memorizzazione* (*store*). I primi corrispondono alla *lettura* (*read*), la seconda alla *scrittura* (*write*).

L'operazione di *caricamento* prende il dato contenuto nella parola di memoria collocata a un indirizzo scelto in modo opportuno e lo scrive all'interno del processore; il contenuto della parola di memoria resta inalterato. Per dare inizio all'operazione di caricamento il processore manda alla memoria l'indirizzo della parola interessata e i comandi che attivano la lettura del contenuto di memoria; la memoria reagisce mandando al processore il dato contenuto nella parola indirizzata.

L'operazione di *memorizzazione* manda il dato dal processore alla parola di memoria collocata a un indirizzo scelto in modo opportuno e lo scrive all'interno della parola; il contenuto precedente della parola di memoria viene sovrascritto. Per dare inizio all'operazione di memorizzazione il processore manda alla memoria l'indirizzo della parola interessata, il dato da scrivere e i comandi che attivano la scrittura del contenuto di memoria; la memoria reagisce sostituendo il nuovo dato a quello precedentemente contenuto nella parola indirizzata.

Come progettare e costruire i circuiti che eseguono le operazioni di memoria appena elencate è spiegato nei Capitoli 5 e 6. Qui invece si vuole illustrare la struttura dell'insieme delle istruzioni del linguaggio macchina, insomma il cosiddetto "livello ISA" dell'architettura del calcolatore, dove tale livello funge da interfaccia tra l'hardware e il software, ma spiegandone il funzionamento dal punto di vista logico, ovvero quali operazioni siano disponibili e come si manipolino operandi e risultati.

2.3 Istruzioni macchina di base

Il programma eseguito dal calcolatore si compone di una sequenza di piccoli passi, chiamati istruzioni macchina, o più brevemente istruzioni, ciascuno dei quali svolge un'operazione

semplice, per esempio l'addizione di due numeri, il confronto di due numeri o caratteri e la lettura o la scrittura di un carattere prendendolo dalla tastiera o mandandolo al video, rispettivamente. L'insieme delle istruzioni riconosciute da parte del calcolatore deve comprendere almeno le quattro tipologie elencate di seguito.

- Trasferimento di dati tra memoria e registri interni al processore.
- Operazioni aritmetiche e logiche su dati.
- Sequenziamento delle istruzioni nel programma e altre operazioni di controllo del loro ordine di esecuzione, che può differire dal loro ordine di memorizzazione.
- Trasferimento di dati tra unità di I/O e registri interni al processore.

Si comincia ora discutendo le prime due tipologie di istruzione, ma per facilitare e sveltire la spiegazione è necessario prima introdurre qualche notazione simbolica opportuna.

2.3.1 Trasferimento di registro

Molta dell'attività del calcolatore si incentra sul trasferimento di istruzioni e dati tra processore e memoria, e secondariamente con le unità di I/O. È dunque necessario definire una notazione formale che permetta di descrivere in modo preciso e sintetico il trasferimento di informazione tra parole di memoria, registri interni al processore e registri interni alle unità di I/O (simili a quelli di cui è dotato il processore). La parola di memoria è identificata mediante l'indirizzo, che può essere una costante numerica o anche simbolica; quest'ultima però deve essere già stata dichiarata in precedenza assegnandole un valore numerico. È bene dare alla costante simbolica di tipo indirizzo un nome espressivo, come IND, POSTO, VAR2 e simili, che ne ricordi l'uso o il contenuto: spesso è il nome di una variabile dichiarata nel programma sorgente scritto in linguaggio di alto livello. Di solito i registri di uso generale del processore sono numerati, ma si usano mediante nomi come R0, R1 o R5. Anche i registri di I/O comunemente sono identificati da nomi, ma meno uniformi di quelli dei registri interni al processore purché altrettanto espressivi, come INGRESSO_DATO o USCITA_DATO e simili, per precisarne in modo chiaro funzione e uso.

Ed ecco gli elementi fondamentali della notazione: racchiudendo un indirizzo (numerico o simbolico) tra parentesi quadre [•] si indica il contenuto della parola di memoria collocata all'indirizzo in questione; racchiudendo un nome di registro (di processore o di I/O) tra parentesi quadre [•] si indica il contenuto del registro con quel nome la freccia diretta a sinistra ← indica trasferimento di valore. Si prenda come esempio l'espressione seguente:

$$R1 \leftarrow [IND]$$

Essa indica che il contenuto della parola di memoria collocata all'indirizzo simbolico IND viene scritto nel registro di nome R1, ovvero il registro numero 1, e che si tratta di un registro di uso generale interno al processore.

Ecco un altro esempio: l'addizione del contenuto dei due registri R1 e R2 e la scrittura nel registro R3 della somma così calcolata; sono tutti e tre registri di uso generale interni al processore. Nella notazione proposta l'operazione si denota nel modo seguente:

$$R3 \leftarrow [R1] + [R2]$$

La simbologia così definita costituisce un mini-linguaggio di specifica chiamato *notazione a trasferimento di registro*, o RTN (*Register Transfer Notation*), ed è molto comoda, pertanto viene usata spesso, seppure con qualche variante esteriore facile da comprendere.

Si badi bene alla distinzione seguente: a sinistra dell'operatore freccia ← deve figurare un elemento (e uno solo) capace di contenere un valore, vale a dire deve figurare una parola

di memoria (identificata dall'indirizzo) o un registro (identificato dal nome); a destra invece ci deve essere un valore semplice o un'espressione che, calcolata, produca un valore. Il contenuto dell'elemento a sinistra viene sempre sovrascritto, sostituendolo con il valore proveniente da destra. Nel gergo tecnico dei calcolatori, le parole "trasferire" e "spostare" (*move*) sono comunemente impiegate per indicare "copia". Trasferire dati da una locazione *sorgente* A a una locazione di *destinazione* B significa che il contenuto della locazione A è letto e quindi scritto nella locazione B. In questa operazione, solo il contenuto della destinazione sarà cambiato. Il contenuto della sorgente resterà immutato.

2.3.2 Notazione simbolica

Occorre anche una notazione adatta a rappresentare in modo simbolico l'istruzione macchina e il programma. La notazione in uso costituisce un vero e proprio linguaggio artificiale, chiamato *linguaggio macchina simbolico* o più brevemente *linguaggio assemblativo* (*assembly language*). Per esempio, in linguaggio assemblativo l'istruzione macchina che esegue l'operazione illustrata nel paragrafo precedente, ovvero il caricamento nel registro R1 della parola di memoria di indirizzo IND, si denota nel modo seguente:

Load R1, IND

Il contenuto delle parola di memoria di indirizzo IND resta inalterato, mentre quello del registro R1 viene sovrascritto.

Il secondo esempio di operazione dato nel paragrafo precedente, ovvero l'addizione dei due registri R1, R2 e la scrittura della somma nel registro R3, si denota invece nel modo seguente:

Add R3, R1, R2

Anche in questo caso R1 e R2 restano inalterati, mentre R3 viene sovrascritto.

Un'*istruzione* specifica un'operazione da eseguire e gli operandi coinvolti. Negli esempi sopra, si usano le parole inglesi Load e Add per indicare le operazioni richieste. Nelle istruzioni assemblative di veri processori (in commercio), tali operazioni sono definite usando dei *codici mnemonici*, che sono generalmente abbreviazioni delle parole che descrivono le operazioni. Per esempio, l'operazione Load può essere scritta come LD, mentre l'operazione Store, che trasferisce una parola da un registro del processore alla memoria, può essere scritta come STR o ST. I linguaggi assemblativi di processori diversi usano spesso codici mnemonici differenti per una data operazione. Per evitare di introdurre i dettagli di un particolare linguaggio assemblativo in questa fase iniziale, si continuerà la presentazione in questo capitolo usando le parole inglesi con iniziale maiuscola anziché i codici mnemonici specifici di un processore. Tali dettagli sono invece introdotti negli approfondimenti applicativi, per i linguaggi assemblativi dei quattro processori ivi considerati.

2.3.3 Insiemi di istruzioni RISC e CISC

Una delle più importanti caratteristiche che distingue calcolatori diversi è la natura delle loro istruzioni. Nella progettazione dell'insieme di istruzioni dei calcolatori moderni ci sono due approcci fondamentali diversi. Un approccio diffuso è basato sulla premessa che prestazioni più alte possono essere ottenute se ogni istruzione occupa esattamente una parola di memoria e se tutti gli operandi necessari per eseguire una certa operazione aritmetica o logica specificata da un'istruzione sono già presenti nei registri del processore. Questo approccio porta a una realizzazione dell'unità di elaborazione in cui le varie operazioni necessarie a gestire una sequenza di istruzioni sono elaborate "a stadi" (*pipeline*), così da sovrapporre le attività e ridurre il tempo totale di esecuzione di un programma, come sarà discusso nel Capitolo 6. Il vincolo

per cui ogni istruzione deve occupare una singola parola riduce la complessità e il numero di tipi diversi di istruzioni che possono essere incluse nell'insieme di istruzioni di un calcolatore. Tali calcolatori sono chiamati *calcolatori con insieme di istruzioni ridotto* o RISC (*Reduced Instruction Set Computer*).

Un'alternativa all'approccio RISC è fare uso di istruzioni più complesse che possono estendersi per più di una parola di memoria e che possono specificare operazioni più complesse. Questo approccio è stato quello prevalente prima dell'introduzione dell'approccio RISC negli anni '70. Sebbene l'uso di istruzioni complesse non fosse originariamente identificato da un termine, i calcolatori basati su questa idea sono stati successivamente chiamati *calcolatori con insieme di istruzioni complesso* o CISC (*Complex Instruction Set Computer*).

Inizieremo la nostra presentazione concentrandoci sugli insiemi di istruzioni in stile RISC poiché sono più semplici e quindi più facili da comprendere. Successivamente affronteremo gli insiemi di istruzioni in stile CISC e spiegheremo le differenze fondamentali tra i due approcci.

2.3.4 Introduzione agli insiemi di istruzioni RISC

Due caratteristiche significative degli insiemi di istruzioni RISC sono presentate di seguito.

- Ogni istruzione occupa una sola parola.
- Si usa un'*architettura carica/immagazzina* (*load/store architecture*) per:
  - accedere agli operandi in memoria usando solo istruzioni Load e Store;
  - usare i registri del processore per fornire gli operandi coinvolti in un'operazione aritmetica o logica, oppure fornire uno degli operandi in maniera immediata dentro la parola dell'istruzione.

All'inizio dell'esecuzione di un programma, tutte le istruzioni e i dati usati dal programma sono memorizzati nella memoria del calcolatore. In quel momento i registri del processore non contengono operandi validi. Se si vuole che gli operandi siano nei registri del processore prima che essi possano essere usati da un'istruzione, allora è necessario innanzitutto portare tali operandi nei registri. Questo compito è svolto da istruzioni Load che copiano il contenuto di una locazione di memoria in un registro del processore. Le istruzioni Load sono del tipo:

Load destinazione, sorgente

o più specificamente:

Load registro_del_processore, locazione_di_memoria

La locazione di memoria può essere indicata in vari modi. Il termine *modo di indirizzamento* è usato per riferirsi alle diverse modalità di farlo, come sarà discusso nel Paragrafo 2.4.

Si consideri adesso una tipica operazione aritmetica. L'operazione di addizione di due numeri è nel repertorio fondamentale di qualsiasi calcolatore. L'asserzione:

$$C = A + B$$

in un linguaggio di alto livello specifica al calcolatore di addizionare i valori correnti di due variabili, A e B, e di assegnare la somma a una terza variabile, C. Quando il programma contenente tale asserzione è compilato, le tre variabili A, B e C sono assegnate a locazioni distinte della memoria. Per semplicità, ci si riferirà agli indirizzi di queste locazioni mediante A, B e C, rispettivamente. I contenuti di queste locazioni rappresentano i valori delle tre variabili. Quindi la suddetta asserzione in linguaggio di alto livello richiede che l'azione:

$$C \leftarrow [A] + [B]$$

Approfondimento applicativo 2.1

Architetture di insiemi di istruzioni di processori in commercio: NIOS II, ColdFire, ARM, IA-32

Le architetture di processori commerciali esaminate qui appresso, come pure negli altri approfondimenti applicativi di questo capitolo e del Capitolo 3, ben si prestano a illustrare come i principi di progettazione RISC e CISC operino in pratica, sia in ambiti applicativi simili sia diversi. I processori NIOS II, prodotti da Altera Corp., e quelli della famiglia ARM, progettati da Advanced RISC Machines Ltd., hanno architetture RISC, mentre i processori ColdFire, prodotti da Freescale Semiconductor Inc., hanno un'architettura CISC di facile apprendimento, al contrario di quella, anch'essa CISC, dei processori della famiglia IA-32, prodotti da Intel Corp., di largo successo commerciale.

I processori NIOS II e ColdFire sono accompagnati da una varietà di strumenti di sviluppo che li rende interessanti anche per finalità formative. In entrambi i casi sono definiti un insieme di istruzioni di base e sue estensioni opzionali, mentre la realizzazione hardware può avvalersi di FPGA (si veda Paragrafo 1.7.4), la cui complessità dipende da quella dell'ISA da realizzare. Questa tecnica di progettazione hardware è la sola prevista per i processori NIOS II, detti anche processori *soft* per le ampie possibilità di specializzazione del progetto hardware disponibili all'utente. Il Caso di studio C1 presenta concetti e metodi di progettazione con FPGA e propone un esempio di progettazione di sistema embedded, una sveglia digitale, basato su processore NIOS II e in cui ci si avvale degli strumenti di sviluppo di cui è equipaggiato.

Anche i processori della famiglia ARM hanno caratteristiche di progettazione che ne favoriscono l'impiego nello sviluppo di sistemi embedded, mentre i processori Intel IA-32, fra i quali la serie Pentium, sono di largo uso nei calcolatori personali. In prevalenza, si basano su processori ARM molti sistemi embedded di basso costo e a basso consumo energetico, quali telefoni cellulari, modem/router per telecomunicazioni, sistemi di controllo di autoveicoli. Al contrario, i processori IA-32 presentano una maggiore dissipazione di energia, a causa della complessità della loro architettura, che supera quella degli altri processori considerati per almeno un ordine di grandezza. Ulteriori dettagli sull'architettura IA-32, che trascendono le finalità dei presenti approfondimenti e che perlopiù non sono trattabili a uno stadio introduttivo di esposizione dei concetti, sono raccolti nel Caso di studio C2. Più in generale, dettagli esaustivi sulle architetture qui considerate sono reperibili nella documentazione ufficiale fornita dai rispettivi produttori [1, 7, 3, 4, 10].

Nel resto di questo approfondimento e nei successivi in questo capitolo si presenterà una sintesi comparativa di alcune caratteristiche delle ISA dei suddetti processori. In particolare, qui di seguito se ne riassumono e confrontano caratteristiche generali, organizzazione della memoria e registri. Nei successivi approfondimenti se ne mettono a confronto modi di indirizzamento, alcune direttive di assemblatore, collegamento dei sottoprogrammi, principali codici operativi e sintassi delle istruzioni. Viene infine fornito un esempio di ristesura di un programma, sviluppato in linguaggio assemblativo generico nel Paragrafo 2.12.1, nei linguaggi assemblativi dei processori considerati e si propongono problemi simili riferiti ad altri esempi di programmi assemblativi generici. Altre caratteristiche, con particolare riguardo alle istruzioni per l'I/O, saranno raffrontate negli approfondimenti applicativi del Capitolo 3.

Caratteristiche generali, memoria e registri

Le principali caratteristiche dei processori in considerazione sono riassunte in Tabella A2.1. Le successive note esplicative sono intese a favorirne il confronto.

I processori RISC hanno solo accesso Load/Store alla memoria e per le istruzioni aritmetiche e logiche esigono operandi in registri. Non hanno tali vincoli i processori CISC, ma le istruzioni

Tabella A2.1 Memoria e registri nei processori NIOS II, ColdFire, ARM e IA-32

Caratteristica	NIOS II	ColdFire	ARM	IA-32
Architettura	RISC	CISC	RISC	CISC
Lunghezza di parola	32 bit	16 bit	32 bit	32 bit
Lunghezza d'istruzione	1 parola	1÷3 parole	1 parola	1÷12 byte
Spazio degli indirizzi	2^{32} byte	2^{32} byte	2^{32} byte	2^{32} byte
Lunghezze di dati (byte)	1,2,4	1,2,4	1,2,4	1,2,4,8,16
Ordinamento dei byte	decresc.	crescente	opzione	decresc.
Registri	r0-r31	A0-A7,PC, D0-D7,SR	R0-R15, CPSR	si veda Caso di Studio C2

IA-32 ammettono al più un operando in memoria. I processori ARM presentano anche alcuni aspetti CISC, ma sono comunque essenzialmente RISC. Le istruzioni ColdFire hanno lunghezza variabile, ma una struttura molto regolare del formato binario, al contrario delle istruzioni IA-32, la cui struttura è cresciuta a dismisura nell'evoluzione dai primi processori Intel a 8 bit e 16 bit fino agli odierni Pentium; l'irregolarità della crescita è stata anche conseguenza del vincolo di retrocompatibilità delle estensioni con i formati delle precedenti ISA dello stesso produttore.

Solo i processori ARM, fra quelli qui visti, ammettono entrambi gli ordinamenti dei byte nelle parole; l'opzione è controllata da un ingresso esterno. I nomi dei registri in tabella sono come dalla documentazione ufficiale dei rispettivi processori. Generalmente gli assemblatori ignorano le differenze tra forma maiuscola e minuscola. L'ordine degli operandi richiesto dagli assemblatori nativi è destinazione-sorgente, tranne per il processore ColdFire, che segue la convenzione opposta, e le istruzioni Store di processori RISC. Altri assemblatori possono seguire un ordine diverso, per esempio GNU *assembler* (GAS) segue sempre l'ordine sorgente-destinazione.

Dei 32 registri NIOS II, 22 sono di uso generale, gli altri 10 hanno usi specifici e come tali anche altri nomi validi, riconosciuti dall'assemblatore, più indicativi della loro funzione. Eccone alcuni (le funzioni di questi registri sono trattate nei successivi paragrafi di questo capitolo):

r0	zero	contiene sempre 0, caricarlo non ha effetti
r1	at	*assembler temporary*, riservato all'assemblatore, non va usato
r27	sp	*stack pointer*, puntatore alla pila
r28	fp	*frame pointer*, puntatore all'area di attivazione
r31	ra	*return address*, registro di collegamento

I registri ColdFire A0-A7 vanno caricati con indirizzi, mentre D0-D7 vanno caricati con dati. Il registro A7 funge da puntatore alla pila.

Nei processori ARM, i registri R12-R15 hanno usi dedicati, in tale ordine, come: puntatore all'area di attivazione (FP), puntatore alla pila (SP), registro di collegamento (LR), contatore di programma (PC). I nomi simbolici in parentesi sono riconosciuti dall'assemblatore, ma i primi due richiedono l'uso di una direttiva di assemblatore a tal fine.

Eccetto che nel processore NIOS II, i bit di esito (si veda il Paragrafo 2.10.2) e altra informazione di stato del processore sono disponibili in apposito registro, variamente denominato:

- SR (*status register*) nei processori ColdFire;
- CPSR (*current program status register*) nei processori ARM;
- EFLAGS (*status register*) nei processori IA-32.

All'indisponibilità hardware dei codici di condizione nei processori NIOS II si supplisce mediante un uso equivalente di altre istruzioni.

L'architettura dei registri IA-32 è molto più complessa delle altre qui considerate, quindi si rinvia al Caso di studio C2 per i suoi particolari.

sia eseguita nel calcolatore. Per effettuare questa azione, i contenuti delle locazioni di memoria A e B sono prelevati dalla memoria e trasferiti al processore dove viene calcolata la loro somma. Questo risultato è quindi inviato alla memoria e immagazzinato nella locazione C.

L'azione richiesta può essere effettuata da una sequenza di semplici istruzioni macchina. Si immagini di usare i registri R2, R3 e R4 per svolgere il compito con quattro istruzioni:

```
Load   R2, A
Load   R3, B
Add    R4, R2, R3
Store  R4, C
```

Si dice che Add è un'istruzione a *tre operandi*, o a *tre indirizzi*, della forma:

```
Add   destinazione, sorgente1, sorgente2
```

L'istruzione Store è della forma:

```
Store   sorgente, destinazione
```

dove la sorgente è un registro del processore e la destinazione è una locazione di memoria. Si osservi che nell'istruzione Store la sorgente e la destinazione sono specificati in ordine inverso rispetto all'istruzione Load; questa è una convenzione di uso comune.

Si noti che possiamo ottenere l'addizione voluta usando solo due registri, R2 e R3, se usiamo uno dei registri sorgenti anche per la destinazione del risultato. In questo caso l'addizione sarebbe effettuata mediante:

```
Add   R3, R2, R3
```

e l'ultima istruzione diventerebbe:

```
Store   R3, C
```

2.3.5 Esecuzione Sequenziale

La Figura 2.4 mostra un breve programma che esegue l'operazione C ← [A] + [B], contenuto in memoria insieme ai dati su cui lavora; l'operazione è la stessa considerata nel paragrafo precedente, la lunghezza della parola è di 32 bit e la memoria è indirizzabile a byte. Le quattro istruzioni macchina costituenti il programma occupano ciascuna esattamente una parola in memoria e sono contenute in quattro parole consecutive, la prima collocata all'indirizzo i. Ne consegue che, siccome ciascuna istruzione occupa 4 byte, la seconda e la terza istruzione si trovano agli indirizzi $i + 4$, $i + 8$ e $i + 12$, rispettivamente. Infine per semplicità si fa l'ipotesi che sia possibile incorporare nell'istruzione l'indirizzo in memoria di un operando, quantunque ciò non sia realistico considerando la lunghezza di parola nei processori correnti. Risolveremo questo punto successivamente nel Paragrafo 2.4.

Ecco come si svolge l'esecuzione del programma. Nel registro PC, *contatore di programma* (*Program Counter*) del processore, si trova l'indirizzo della parola di memoria contenente la prossima istruzione da eseguire. Per dare inizio all'esecuzione del programma bisogna scrivere in PC l'indirizzo della prima istruzione del programma, che nel caso presente vale i. In seguito i circuiti di controllo del processore si servono di PC per prelevare ed eseguire le istruzioni, una alla volta, in ordine crescente di indirizzo, o come si dice in ordine di *esecuzione sequenziale*. Ciò avviene aumentando di 4 unità l'indirizzo ogni volta durante l'esecuzione dell'istruzione. Pertanto, quando l'istruzione di memorizzazione all'indirizzo $i + 12$ sarà stata

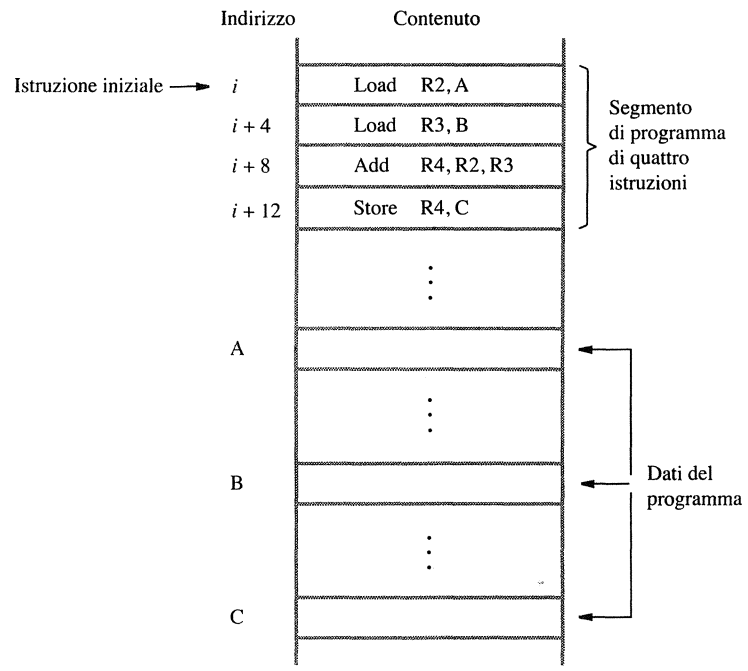


Figura 2.4 Programma che esegue l'operazione C ← [A] + [B].

eseguita il registro PC conterrà il valore $i + 16$, che è l'indirizzo dell'istruzione iniziale del segmento di programma successivo.

L'esecuzione dell'istruzione macchina è un processo a due fasi. Nella prima, chiamata *fase di prelievo* (*instruction fetch*), l'istruzione viene presa dalla parola di memoria il cui indirizzo è contenuto nel registro PC del processore e viene scritta nel *registro di istruzione* IR, che fa anch'esso parte del processore. All'inizio della seconda fase, chiamata *fase di esecuzione* (*instruction execute*), il contenuto di IR viene esaminato, o come altrimenti detto *decodificato*, per stabilire quale sia l'operazione da eseguire, la quale subito dopo viene messa in atto da parte del processore. L'esecuzione dell'operazione può comportare parecchi compiti ausiliari, come caricamento di operandi, calcoli vari di tipo aritmetico o logico e memorizzazione del risultato. In un qualche momento intermedio alla fase di esecuzione il registro PC viene incrementato facendolo puntare alla prossima istruzione da eseguire, così quando l'istruzione corrente sarà terminata il registro PC conterrà l'indirizzo della prossima e si potrà pertanto dare inizio a una nuova fase di prelievo.

2.3.6 Esecuzione con salto

Si consideri il problema di addizione di una lista di $n \geq 1$ numeri interi. Si può generalizzare il programma già visto in Figura 2.4 ottenendo quello mostrato in Figura 2.5. Gli indirizzi delle parole di memoria contenenti i numeri da addizionare sono indicati in modo simbolico come NUM1, NUM2, ..., NUMn; n istruzioni di addizione indipendenti ne calcolano progressivamente la somma accumulandola nel registro R2 e al termine del calcolo la somma viene memorizzata

i	Load R2, NUM1
$i+4$	Load R3, NUM2
$i+8$	Add R2, R2, R3
$i+12$	Load R3, NUM3
$i+16$	Add R2, R2, R3
	⋮
$i+8n-12$	Load R3, NUMn
$i+8n-8$	Add R2, R2, R3
$i+8n-4$	Store R2, SOMMA
	⋮
SOMMA	
NUM1	
NUM2	
	⋮
NUMn	

Figura 2.5 Programma sequenziale che calcola la somma di n numeri interi.

nella parola all'indirizzo simbolico SOMMA. Solo un programmatore alle prime armi, però, procederebbe in modo così ingenuo. Invece di ricorrere a una lunga serie di istruzioni di addizione ne basta una sola racchiusa in un ciclo, come mostra la Figura 2.6. Il corpo del ciclo, che inizia alla parola di indirizzo simbolico CICLO e termina subito prima dell'istruzione macchina Branch_if_[R0]>0, è formato da una serie di istruzioni eseguita in modo sequenziale e iterata per tutte le volte che occorre. A ogni giro il corpo del ciclo provvede a calcolare l'indirizzo del prossimo elemento della lista e a caricarlo dalla memoria addizionandolo al registro R3. Si può rappresentare e calcolare l'indirizzo dell'elemento in parecchi modi differenti, come si vedrà nel Paragrafo 2.4, ma per ora basta spiegare come costruire il ciclo e come controllarlo.

Si facciano le ipotesi seguenti: il numero n di elementi della lista è tenuto nella parola di memoria all'indirizzo simbolico N; il registro R2 è usato come contatore per tenere traccia del numero di giri del ciclo; il contenuto della parola di indirizzo N viene caricato in R2 all'inizio del programma. Dentro il ciclo, l'istruzione:

Subtract R2, R2, #1

sottrae un'unità al contenuto di R2 per ogni ripetizione del ciclo (sarà spiegato il significato del simbolo "#", ovvero dichiarazione di costante numerica, nel Paragrafo 2.4.1). L'esecuzione

		Load R2, N
		Clear R3
Ciclo del programma	CICLO	Determina l'indirizzo del prossimo numero, carica il prossimo numero in R5 e aggiungilo a R3
		Subtract R2, R2, #1
		Branch_if_[R2]>0 CICLO
		Store R3, SOMMA
		⋮
	SOMMA	
	N	n
	NUM1	
	NUM2	
		⋮
	NUMn	

Figura 2.6 Programma ciclico che calcola la somma di n numeri interi.

del corpo del ciclo viene ripetuta ogniqualvolta il risultato dell'istruzione di decremento riesce maggiore di zero.

Tutto è pronto per definire una nuova tipologia di istruzione macchina, il *salto* (*branch*, che alla lettera significa *diramazione*). Essa scrive un nuovo valore nel registro contatore di programma PC, di conseguenza il processore preleva ed esegue l'istruzione collocata in memoria a tale nuovo indirizzo, detto *di destinazione di salto* (*branch target*), invece di proseguire in modo puramente sequenziale prelevando ed eseguendo l'istruzione che si trova all'indirizzo consecutivo a quello dell'istruzione di salto stessa. L'istruzione di salto può essere incondizionata oppure condizionata, a seconda che il salto avvenga sempre oppure solo a seguito del verificarsi di una condizione specificata dall'istruzione stessa. Se la condizione di salto riesce falsa si procede in ordine di esecuzione sequenziale, se riesce vera si effettua il salto nel modo detto sopra. Nel programma in Figura 2.6, l'istruzione:

Branch_if_[R2]>0 CICLO

compie il salto all'istruzione collocata all'indirizzo CICLO (qui inteso come costante simbolica, dunque senza racchiuderlo tra parentesi quadre [•]) se il risultato dell'operazione immediatamente precedente al salto, cioè quella di decremento del registro R2, è maggiore di zero.

In altre parole, il ciclo viene ripetuto ogniqualvolta ci siano ancora elementi della lista da addizionare al registro R3. Alla fine dell'n^{esimo} giro del ciclo l'istruzione di decremento dà valore 0 e il salto non viene più effettuato, bensì si va in sequenza passando a eseguire l'istruzione consecutiva al salto, la quale memorizza il risultato contenuto in R3 nella parola all'indirizzo simbolico SOMMA.

La capacità di verificare condizioni e fare scelte in un insieme di alternative su come proseguire l'esecuzione è fondamentale e ha molte altre applicazioni oltre al controllo di ciclo; infatti qualsiasi linguaggio macchina contiene un certo numero di istruzioni di salto differenti per coprire tutti gli usi significativi.

Un modo per realizzare un'istruzione condizionale è confrontare i contenuti di due registri e saltare (*branch*) verso l'istruzione di destinazione se il confronto soddisfa il requisito specificato. Per esempio, l'istruzione:

Branch_if_[R4]>[R5] CICLO

può essere scritta in un generico linguaggio assemblativo come:

Branch_greater_than R4, R5, CICLO

o usando un codice mnemonico in uno specifico linguaggio assemblativo per mezzo di:

BGT R4, R5, CICLO

Tale istruzione confronta il contenuto dei registri R4 e R5, senza cambiare il contenuto di alcuno dei registri. Quindi salta verso CICLO se il contenuto di R4 è maggiore del contenuto di R5.

Un diverso modo di realizzare le istruzioni di salto usa il concetto di codici o bit di condizione (*condition code*) di cui si discuterà nel Paragrafo 2.10.2.

2.3.7 Calcolo di indirizzo

Tornando alla Figura 2.6 si nota come il compito principale del corpo del ciclo che inizia all'indirizzo simbolico CICLO sia quello di addizionare un nuovo elemento della lista per ogni iterazione. Senza dubbio dunque il ciclo contiene un'istruzione di addizione, la quale a ogni giro deve fare riferimento a un indirizzo diverso, ma per quanto noto finora, una volta che l'indirizzo dell'operando sia stato specificato nell'istruzione, non è più possibile modificarlo. Una soluzione consiste nel supporre che invece di dare direttamente l'indirizzo come parte dell'istruzione, si possa dare un registro R_i a sua volta contenente l'indirizzo. Il registro R_i va caricato all'inizio con l'indirizzo NUM1, prima di avviare il ciclo, poi va aumentato di 4 unità a ogni giro del ciclo.

La situazione così descritta non è per nulla infrequente, si verifica anche in molti altri contesti e pertanto rende necessario disporre di modi più flessibili per specificare l'indirizzo di un operando. Essi vanno sotto il nome di *modi di indirizzamento* (*addressing mode*) e l'insieme di istruzioni macchina del processore di norma ne fornisce un certo numero. I dettagli variano a seconda del processore, ma i concetti sono gli stessi e verranno appunto illustrati nel prossimo paragrafo.

2.4 Modi di indirizzamento

Nei paragrafi precedenti sono stati illustrati alcuni esempi di programmi semplici in linguaggio macchina che lavorano su dati contenuti in memoria. I dati si possono organizzare in molti modi diversi che riflettono la natura dell'informazione e come essa è usata. Per organizzare

i dati il programmatore si serve delle *strutture dati* messegli a disposizione dal linguaggio di alto livello, che comprendono vettore, tabella, lista con indice o concatenata, insieme, coda, albero, grafo e tante altre.

Siccome all'inizio il programma è solitamente scritto in linguaggio di alto livello, il programmatore ha a disposizione costanti, variabili locali e globali, puntatori, vettori (*array*) e strutture (*struct*) o record. Ma quando traduce in linguaggio macchina il programma, il compilatore deve ridefinire tutti questi oggetti servendosi solo delle funzioni offerte dalle istruzioni del calcolatore dove il programma è destinato a girare. I vari metodi mediante cui nell'istruzione macchina si può specificare la posizione di operandi e risultato si chiamano *modi di indirizzamento* (*addressing mode*). In questo paragrafo sono presentati i modi di indirizzamento fondamentali dei processori RISC. Un riassunto è fornito in Tabella 2.1, che include inoltre la sintassi assemblativa generica che si userà per ciascun modo di indirizzamento. Tale sintassi (discussa nel Paragrafo 2.5) definisce come specificare le istruzioni e i modi di indirizzamento dei loro operandi.

2.4.1 Costante e variabile

Le variabili si trovano in quasi tutti i programmi. In linguaggio macchina si rappresenta la variabile assegnandole un registro o una parola di memoria per contenerne il valore che istruzioni macchina appropriate possono modificare.

Il programma nella Figura 2.5 si limita a fare uso di due soli modi di indirizzamento per lavorare sulle variabili, vale a dire utilizza il nome del registro oppure l'indirizzo della parola di memoria contenente la variabile. Eccone le definizioni precise.

- *Modo di registro*: l'operando o il risultato è contenuto in un *registro* di processore, il cui *nome* (che è anche indirizzo) è dato nell'istruzione.
- *Modo assoluto* (o *diretto*): l'operando o il risultato è contenuto in una *parola di memoria*, il cui *indirizzo* è dato nell'istruzione.

Poiché in un processore RISC un'istruzione deve occupare esattamente una singola parola, c'è un limite al numero di bit che possono essere usati per un indirizzo assoluto. Tipicamente tale limite è 16 bit se la lunghezza della parola è 32 bit. In genere, per avere un indirizzo di 32 bit, il valore a 16 bit è esteso a 32 bit replicando il bit b_{15} nelle posizioni dei bit b_{31-16} (come nell'estensione del segno). Questo significa che un indirizzo assoluto può essere specificato in questo modo solo per un intervallo limitato dell'intero spazio degli indirizzi. Si affronterà il problema di come specificare tutti gli indirizzi a 32 bit nel Paragrafo 2.9. Per mantenere semplici gli esempi, si assumerà per adesso che tutti gli indirizzi delle locazioni di memoria coinvolti in un programma possano essere indicati usando 16 bit.

Tabella 2.1 Modi di indirizzamento di tipo RISC

Nome	Sintassi assemblativa	Funzione di indirizzamento (in RTN)
Immediato	#Valore	Operando = Valore
Registro	R*i*	EA = R*i*
Assoluto (o diretto)	LOC	EA = LOC
Indiretto da registro	(R*i*)	EA = [R*i*]
Con indice e spiazzamento	X(R*i*)	EA = [R*i*] + X
Con base e indice	(R*i*,R*j*)	EA = [R*i*] + [R*j*]

EA = indirizzo effettivo; Valore = un numero con segno; X = valore dello spiazzamento

L'istruzione:

Add R4, R2, R3

usa il modo di registro per tutti e tre gli operandi. I registri R2 e R3 contengono i due operandi sorgenti, mentre R4 è la destinazione.

Il modo assoluto può rappresentare variabili globali in un programma. Una dichiarazione del tipo:

Integer NUM1, NUM2, SOMMA

in un linguaggio di alto livello farà in modo che il compilatore allochi una locazione di memoria per ognuna delle variabili NUM1, NUM2 e SOMMA. Quando successivamente si hanno riferimenti a esse nel programma, il compilatore può generare istruzioni assemblative che usano il modo assoluto per accedere a queste variabili.

Il modo assoluto è usato nell'istruzione:

Load R2, NUM1

che carica il valore alla locazione di memoria NUM1 nel registro R2.

Anche costanti, che rappresentano dati o indirizzi, si trovano in quasi tutti i programmi per calcolatore. In linguaggio macchina si rappresentano mediante il modo immediato. Eccone la definizione:

- *Modo immediato* (o di *costante*): l'operando è *dato esplicitamente* nell'istruzione.

Ed ecco un esempio di istruzione che ne fa uso:

Add R4, R6, $200_{\text{immediato}}$

aggiunge il valore 200 al contenuto del registro R6 e pone il risultato nel registro R4. L'uso del pedice per denotare il modo immediato non è appropriato nei linguaggi assemblativi. Una convenzione comune è quella di usare il segno di dichiarazione di costante numerica (#) davanti al valore per indicare che questo valore è usato come operando immediato. Quindi, scriviamo l'istruzione sopra come:

Add R4, R6, #200

Nel modo di indirizzamento seguente l'istruzione non dà esplicitamente l'indirizzo di operando o risultato, piuttosto specifica elementi di informazione partendo dai quali si può ricavarlo. Quando l'indirizzo stesso non è subito noto ma è oggetto di un qualche tipo di trasferimento o calcolo, viene indicato come *indirizzo effettivo* o EA (*Effective Address*). Questo è quindi utilizzato per accedere all'operando.

2.4.2 Modo indiretto e puntatore

Il programma in Figura 2.6 richiede la possibilità di modificare l'indirizzo dell'operando in memoria a ogni passata del ciclo. Un buon modo per realizzare questa possibilità è usare un registro del processore per tenere l'indirizzo dell'operando. Il contenuto del registro viene quindi modificato (incrementato) durante ogni passata per fornire l'indirizzo del prossimo numero della lista a cui accedere. Il registro agisce come un *puntatore* alla lista, e diciamo che si accede *indirettamente* a un elemento della lista usando l'indirizzo nel registro. La desiderata possibilità è fornita dal modo di indirizzamento indiretto (si veda la Figura 2.7).

- *Modo indiretto*: l'indirizzo effettivo di operando o risultato è contenuto in un *registro* di processore, il cui *nome* è dato nell'istruzione.

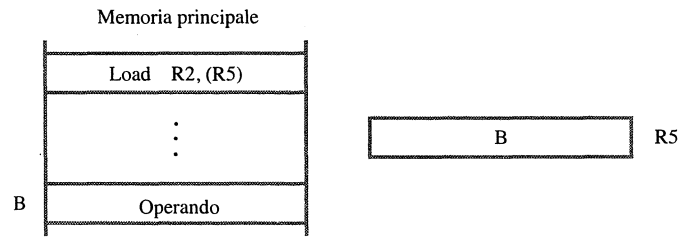


Figura 2.7 Modo indiretto di indirizzamento.

Nel formato di istruzione generico si segnala il modo indiretto racchiudendo tra parentesi tonde (•) il nome del registro (si vedano la Tabella 2.1 e la Figura 2.7).

Per eseguire l'istruzione Load della Figura 2.7, il processore usa il valore B, che è nel registro R5, come indirizzo effettivo dell'operando. Un'operazione Read è necessaria per prelevare il contenuto della locazione B in memoria. Il valore in memoria è l'operando desiderato, che il processore carica nel registro R2. L'indirizzamento indiretto attraverso una locazione di memoria è anche possibile, ma si trova solo nei processori di stile CISC.

Nella programmazione, il modo indiretto e l'uso dei puntatori sono concetti importanti e potenti. Essi permettono di usare lo stesso codice per operare su dati differenti. Per esempio, il registro R5 in Figura 2.7 serve all'istruzione Load come puntatore per caricare un operando dalla memoria nel registro R2. A un certo punto, R5 può puntare alla locazione B della memoria. In seguito, il programma può cambiare il contenuto di R5 per puntare a una locazione differente; in questo caso la stessa istruzione Load caricherà il valore da quella locazione in R2. Quindi un segmento di programma che include questa istruzione Load è convenientemente riusato solo cambiando il valore del puntatore.

Tornando al programma in Figura 2.6 che somma una lista di numeri, l'indirizzamento indiretto può essere usato per accedere sequenzialmente ai numeri della lista, conseguentemente si avrà il programma mostrato in Figura 2.8. Il registro R4 è usato come un puntatore ai numeri della lista e l'accesso agli operandi avviene indirettamente per mezzo di R4. La sezione di inizializzazione del programma carica il valore del contatore *n* dalla locazione di memoria N in R2. Quindi, l'istruzione Clear è usata per porre R3 a 0. La successiva istruzione usa il modo di indirizzamento immediato per inserire il valore dell'indirizzo NUM1, che è

	Load	R2,N	Carica la dimensione della lista
	Clear	R3	Inizializza la somma a 0
	Move	R4,#NUM1	Carica l'indirizzo del primo numero
CICLO:	Load	R5, (R4)	Preleva il prossimo numero
	Add	R3, R3, R5	Aggiungi questo numero alla somma
	Add	R4, R4, #4	Incrementa il puntatore alla lista
	Subtract	R2, R2, #1	Decrementa il contatore
	Branch_if_[R2]>0	CICLO	Salta indietro se non ancora finito
	Store	R3, SOMMA	Immagazzina la somma finale

Figura 2.8 Uso del modo indiretto di indirizzamento nel programma di Figura 2.6.

l'indirizzo del primo numero della lista, in R4. Si osservi che non è possibile usare l'istruzione Load per caricare il valore immediato, poiché l'istruzione Load può operare solo se l'operando sorgente è in memoria. In alternativa, si usa l'istruzione Move:

Move R4, #NUM1

In molti processori di tipo RISC, un registro di uso generale è dedicato a contenere il valore costante zero. Di solito, questo è il registro R0. Il suo contenuto non può essere modificato da istruzioni di programma. Si assumerà, comunque, che R0 sia usato in questo modo nella discussione sui processori in stile RISC. Quindi, l'istruzione Move suddetta può essere effettuata come:

Add R4, R0, #NUM1

Spesso Move è fornita come una *pseudoistruzione* per convenienza dei programmatori, ma è in realtà effettuata usando l'istruzione Add.

Le prime tre istruzioni del ciclo in Figura 2.8 realizzano il blocco di istruzioni non specificate che inizia con CICLO in Figura 2.6. Alla prima esecuzione del ciclo, l'istruzione:

Load R5, (R4)

preleva l'operando alla locazione NUM1 e lo carica in R5. La prima istruzione Add aggiunge questo numero alla somma nel registro R3. La seconda istruzione Add aggiunge 4 al contenuto del puntatore R4, così che conterrà il valore dell'indirizzo NUM2 quando l'istruzione Load sarà eseguita alla seconda passata del ciclo.

Come altro esempio di puntatori, consideriamo l'espressione in linguaggio C:

A = *B

dove B è una variabile puntatore e il simbolo "*" è l'operatore per gli accessi indiretti. Questa espressione fa in modo che il contenuto della locazione di memoria puntata da B sia caricato nella locazione di memoria A. L'espressione può essere compilata come segue:

Load R2, B

Load R3, (R2)

Store R3, A

L'indirizzamento indiretto attraverso i registri è ampiamente usato. Il programma in Figura 2.8 mostra la flessibilità da esso offerta.

2.4.3 Modo con indice e vettore

Il prossimo modo di indirizzamento aggiunge nuovi gradi di flessibilità al linguaggio macchina e in particolare si dimostra utile nel gestire vettori e liste. Eccone la definizione:

- *Modo con indice e spiazzamento*: l'indirizzo effettivo di operando o risultato è ottenuto addizionando un *valore costante*, chiamato *spiazzamento*, al *contenuto di un registro*, dove sia spiazzamento sia nome del registro sono dati nell'istruzione.

Il registro può essere o di uso speciale, dedicato al modo con indice e spiazzamento, o di uso generale, caso che in realtà è più frequente, ma comunque sia viene chiamato *registro indice* (*index register*). In linguaggio macchina si segnala il modo con indice e spiazzamento mediante la scrittura X(R*i*), dove X (*offset*) indica lo *spiazzamento* (costante) e R*i* è il nome del

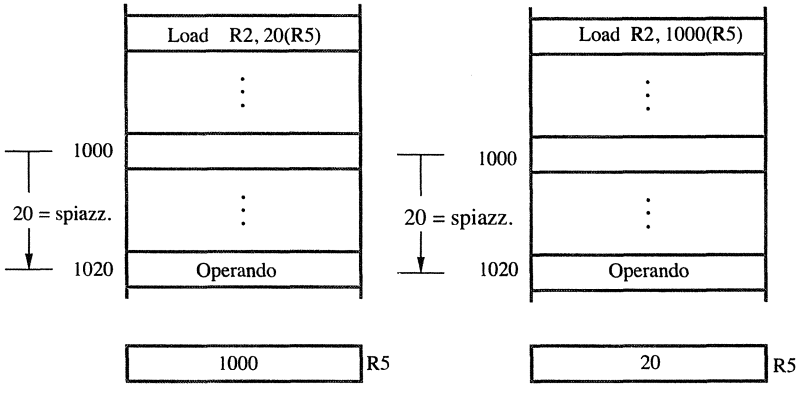
registro indice. In formalismo RTN l'indirizzo effettivo EA si calcola secondo la formula EA = X + [R*i*]. Si tenga comunque presente che il contenuto del registro indice R*i* *non viene modificato* calcolando l'indirizzo effettivo.

In linguaggio assemblativo si può specificare la costante X in forma sia numerica sia simbolica. Nel Paragrafo 2.5 si mostrerà come associare l'indirizzo simbolico al valore numerico corrispondente. Traducendo in forma numerica binaria il programma assemblativo, la costante X viene incorporata nell'istruzione perché in genere consuma meno bit di quanti non ne siano concessi dalla lunghezza di parola caratteristica del calcolatore. Siccome però la costante X è dotata di segno e di solito si rappresenta in complemento a due, prima di addizionarla al registro indice R*i* occorre estenderla in segno (si veda il Paragrafo 1.4).

In Figura 2.9 sono illustrate due concezioni diverse del modo con indice e spiazzamento. In Figura 2.9a si immagina che il registro indice R*i* contenga l'indirizzo di una parola di memoria di partenza e che la costante X lo trasli a una certa distanza su un'altra parola di memoria. Invece in Figura 2.9b è la costante X a rappresentare l'indirizzo di una parola di memoria di partenza e la traslazione è data dal contenuto del registro indice R*i*. In entrambi i casi però l'indirizzo effettivo è somma di due componenti: uno è X ed è dato esplicitamente nell'istruzione, mentre l'altro è il contenuto di R*i*.

Per comprendere l'utilità del modo con indice e spiazzamento si consideri un esempio semplice, incentrato su un elenco di allievi iscritti a un certo corso. La Figura 2.10 mostra la struttura della lista di elementi che in memoria rappresenta l'elenco. L'elemento è costituito da un blocco di quattro parole consecutive e contiene le informazioni rilevanti per ciascun allievo. Esse sono il numero di matricola dell'allievo seguito dai voti guadagnati in una serie di tre prove di verifica. Il corso conta *n* allievi iscritti e il numero *n* è tenuto nella parola di memoria all'indirizzo simbolico N che precede il primo elemento della lista. Si suppone infine che la parola sia da 32 bit e che la memoria sia indirizzabile per byte.

È utile osservare che si può immaginare la lista in Figura 2.10 come *matrice* o *vettore* 2D (2D *array*) dotato di *n* righe e quattro colonne. Ciascuna riga corrisponde a un allievo e i quattro elementi nella riga ne rappresentano estremi e valutazione.



(a) Spiazzamento dato come costante

(b) Spiazzamento dato nel registro indice

Figura 2.9 Modo di indirizzamento con indice e spiazzamento.

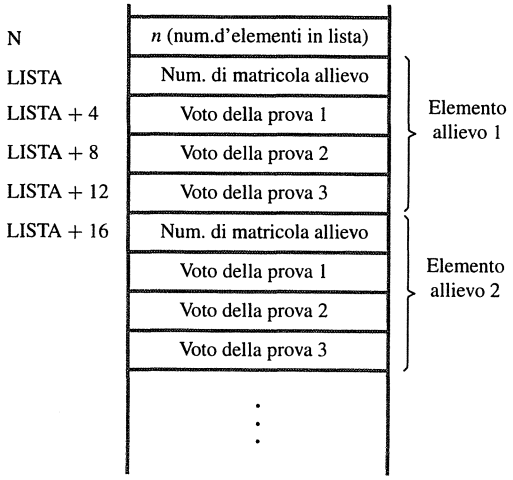


Figura 2.10 Elenco di studenti con dati relativi alla carriera didattica.

Ciò che si vuole fare è calcolare le tre somme, ciascuna estesa a tutti gli allievi, dei voti conseguiti nelle tre prove e scriverle nelle tre parole di memoria agli indirizzi SOMMA1, SOMMA2 e SOMMA3. La Figura 2.11 mostra un possibile programma che svolge tale compito. Dentro il ciclo si usa il modo con indice e spiazzamento, concepito secondo il modello illu-

```
        Move          R2, #LISTA     Carica l'indirizzo LISTA
        Clear         R3
        Clear         R4
        Clear         R5
        Load          R6, N          Carica il valore n
CICLO:  Load          R7, 4(R2)      Aggiungi il voto della prova 1 del prossimo
        Add           R3, R3, R7       studente alla somma parziale
        Load          R7, 8(R2)      Aggiungi il voto della prova 2 dello studente
        Add           R4, R4, R7       alla somma parziale
        Load          R7, 12(R2)     Aggiungi il voto della prova 3 dello studente
        Add           R5, R5, R7       alla somma parziale
        Add           R2, R2, #16    Incrementa il puntatore
        Subtract      R6, R6, #1     Decrementa il contatore
        Branch_if_[R6]>0  CICLO      Salta indietro se non ancora finito
        Store         R3, SOMMA1     Immagazzina il totale per la prova 1
        Store         R4, SOMMA2     Immagazzina il totale per la prova 2
        Store         R5, SOMMA3     Immagazzina il totale per la prova 3
```

Figura 2.11 Uso del modo con indice e spiazzamento per gestire l'elenco di studenti con dati di carriera didattica.

strato in Figura 2.9a, per indirizzare i voti delle prove di ciascun allievo. Il registro R2 ha funzione di indice e prima di avviare il ciclo viene predisposto facendolo puntare alla testa della lista; detto altrimenti, all'inizio R2 contiene l'indirizzo LISTA.

Al primo giro del ciclo i tre voti del primo allievo vengono aggiunti ai tre registri R3, R4 e R5, che contengono le tre somme parziali e che all'inizio sono stati azzerati. Le tre parole di memoria contenenti i voti sono indirizzate in modo con indice e spiazzamento mediante 4(R2), 8(R2) e 12(R2). Passando al secondo giro il registro indice R2 viene aumentato di 16 unità facendolo puntare al numero di matricola del secondo allievo in elenco. Il registro R6, che ha funzione di contatore di elementi e in cui all'inizio è stato scritto *n*, viene diminuito di un'unità a ogni giro; quando assume valore 0 tutta la lista è stata scandita e il programma si avvia a conclusione. Fino ad allora l'istruzione di salto condizionato in fondo al ciclo riporta il flusso di esecuzione in testa al ciclo per passare a elaborare il prossimo elemento in lista. Le tre istruzioni finali del programma memorizzano le somme totali in tre parole agli indirizzi SOMMA1, SOMMA2 e SOMMA3.

È bene ribadire che il contenuto di R2 non viene modificato dall'uso che se ne fa come registro indice. Nel programma il contenuto di R2 viene cambiato solo dall'ultima istruzione di addizione dentro il ciclo, aumentandolo di 16 unità per passare dall'elemento corrente a quello successivo.

In generale il modo con indice e spiazzamento torna utile per indirizzare operandi o risultati la cui posizione sia definita relativamente alla testa della struttura dati cui appartengono. Nella lista in questione la testa dell'elemento è data dal numero di matricola e i tre voti a seguire vengono indirizzati in modo con indice e spiazzamento a partire dalla posizione del numero di matricola dell'allievo cui si riferiscono.

Quella vista finora è la forma più semplice del modo con indice e spiazzamento, ma ci sono parecchie varianti più o meno raffinate molto utili per indirizzare in modo efficiente operandi e risultati in tutte le situazioni di interesse programmativo pratico. Per esempio, si potrebbe usare un secondo registro ottenendo il *modo con base e indice*, che si denota così: (R*i*, R*j*). L'indirizzo effettivo EA è la somma di R*i*, chiamato *registro base*, e R*j*, chiamato *registro indice*, come nel modo con indice e spiazzamento: EA = [R*i*] + [R*j*] in formalismo RTN. In un certo senso qui il registro R*j* ha funzione di puntatore e il registro R*i* tiene luogo di spiazzamento; dunque quest'ultimo non è più costante ma può variare. Si noti sempre l'uso della coppia di parentesi tonde (•) conforme al modo con indice e spiazzamento originale. Rispetto a questo, il modo con base e indice è più versatile perché entrambi i componenti che concorrono a formare l'indirizzo effettivo sono modificabili.

Andando ancora più in là si trova il *modo con base, indice e spiazzamento*, che come si intuisce fa uso di due registri R*i* e R*j* come base e indice, rispettivamente, e di una costante X come spiazzamento. In linguaggio macchina simbolico il nuovo modo si segnala così: X(R*i*, R*j*); in formalismo RTN l'indirizzo effettivo EA si calcola come segue: EA = X + [R*i*] + [R*j*]. Questa ulteriore flessibilità è utile nell'accesso a componenti multiple dentro ciascun elemento di una struttura, dove l'inizio di un elemento è specificato dal frammento (R*i*, R*j*) del modo di indirizzamento.

Infine, dovremmo notare che nel modo con indice e spiazzamento:

X(R*i*)

se il contenuto del registro è pari a zero, allora l'indirizzo effettivo è uguale al valore di X con estensione del segno. Questo ha lo stesso effetto del modo assoluto. Se il registro R0 contiene sempre il valore zero, allora il modo assoluto è realizzato semplicemente come:

X(R0)

Approfondimento applicativo 2.2

Modi di indirizzamento NIOS II, ColdFire, ARM, IA-32

La sintesi comparativa delle caratteristiche dei quattro processori qui considerati prosegue con il raffronto dei loro principali modi di indirizzamento, raccolti nella Tabella A2.2, in cui se ne indica anche la notazione nei linguaggi assemblativi nativi. Per completezza, la tabella mostra anche modi di indirizzamento che sono introdotti nel Paragrafo 2.10.1, ovvero i modi con autoincremento, autodecremento, il modo relativo a PC e altri modi, presenti in uno solo dei quattro processori considerati, ai quali si applicano le note esplicative che seguono. Occorre però premettere alcuni chiarimenti, rispettivamente relativi a notazione e terminologia impiegate in Tabella A2.2.

Le parentesi quadre nella notazione dei modi di indirizzamento dei processori ARM e IA-32 non riguardano la descrizione delle loro funzioni di indirizzamento in RTN, come avviene in Tabella 2.1, ma sono parte della notazione concreta di modi di indirizzamento di tali processori, come lo è pure il simbolo "#" per i processori ARM e ColdFire.

Tabella A2.2 Modi di indirizzamento e relativa notazione simbolica nelle ISA NIOS II, ColdFire, ARM e IA-32

Modo di indirizzamento	NIOS II	ColdFire	ARM	IA-32
Immediato	Valore	#Valore	#Val	Valore
Assoluto o diretto	LOC(r0)	Valore	Val	LOC
Di registro	r*i*	R*i*	R*i*	R
Indiretto di registro	(r*i*)	(A*i*)	[R*i*]	[R]
Con base e spiazzamento	X(r*i*)	W(A*i*)	[R*i*,#Val]	[R+X]
Con indice e spiazz.				[R_x*S+X]
Con base e indice			[R*i*,±R*j*,s]	[R+R_x*S]
Con base, indice e spiazz.		B(A*i*,R*j*)		[R+R_x*S+X]
Con autoincremento		(A*i*)+		
Con autodecremento		–(A*i*)		
Relativo a PC		W(PC)	L	L
Relativo a PC con indice		B(PC,R*i*)		
Indiretto da memoria				*LOC oppure [R_x*S+X]
Con pre-base e spiazz.			[R*i*,#Val]!	
Con post-base e spiazz.			[R*i*],#Val	
Con pre-base e indice			[R*i*,±R*j*,s]!	
Con post-base e indice			[R*i*],±R*j*,s	

Legenda:

Valore numero con segno (a 16 bit in NIOS II, a 8 o 32 bit in IA-32) rappresentato esplicitamente o da etichetta;
Val numero rappresentato in valore assoluto e segno a 9 bit nel modo immediato, a 13 bit nei modi assoluto e con spiazzamento;
LOC indirizzo assoluto (a 16 bit in NIOS II, a 32 bit in IA-32);
R, R_x uno degli otto registri generali IA-32, ma non si può usare il registro ESP (puntatore alla pila) come registro indice R_x; R*i*, R*j*, ISA ColdFire: registro A*i* o D*i* (rispettivamente A*j* o D*j*);
X spiazzamento: numero con segno (a 16 bit in NIOS II, a 8 o 32 bit in IA-32, ma solo a 32 bit nel modo con indice e spiazzamento);
S fattore di scala (IA-32): 1, 2, 4 o 8;
s scorrimento logico (ARM): ds #vs dove ds ∈ {LSL, LSR}: direzione dello scorrimento e vs: valore dello scorrimento (numero a 5 bit);
W Valore a 16 bit;
B Valore a 8 bit;
L Etichetta.

Il modo denominato "con indice e spiazzamento" in Tabella 2.1 e nel Paragrafo 2.4.3 è qui denominato *con base e spiazzamento*, perché nei processori ARM e IA-32 il termine *registro indice* è proprio dell'uso di un registro in un modo di indirizzamento un po' più complesso, in cui il contenuto del registro indice può essere moltiplicato per un fattore di scala (nei processori IA-32) o sottoposto a uno scorrimento logico (nei processori ARM), prima di essere addizionato a uno spiazzamento e/o al contenuto di un registro di base. Infine, la terminologia IA-32 denomina *diretto* il modo di indirizzamento assoluto.

Nell'ISA IA-32, i modi di indirizzamento relativi a PC e indiretto da memoria sono impiegati solo per indirizzi di istruzioni. Nel processore ColdFire il modo relativo a PC ha due varianti, rispettive istanze del modo con base e spiazzamento e del modo con base, indice e spiazzamento, dove il registro contatore di programma funge da registro di base. Il modo relativo a PC si effettua similmente anche nei processori ARM e IA-32, tuttavia il programmatore specifica solo l'etichetta simbolica della locazione indirizzata, poiché l'assemblatore provvede a calcolare il valore (con segno) dello spiazzamento, tenendo conto dell'indirizzo della successiva istruzione nella sequenza di memorizzazione, ovvero del contenuto del PC a tempo di esecuzione.

Nel modo indiretto da memoria si calcola un indirizzo di memoria come nel modo con indice e spiazzamento, ma detto indirizzo non è quello effettivo, bensì quello di una locazione di memoria che lo contiene.

Si osservi che l'uso, nel modo di indirizzamento indiretto da memoria, della stessa notazione impiegata per il modo con indice e spiazzamento non è fonte di ambiguità, poiché l'impiego del modo indiretto da memoria è limitato solo ad alcune istruzioni (in cui l'indirizzo effettivo è quello di un'istruzione), mentre nelle altre istruzioni tale notazione designa il modo con indice e spiazzamento. Se invece si fornisce un indirizzo assoluto (in forma simbolica o numerica) nel modo indiretto da memoria, non si può impiegare la notazione propria del modo di indirizzamento diretto, che diverrebbe perciò ambigua; si spiega così il ruolo del simbolo "*" in tale forma del modo di indirizzamento indiretto da memoria.

Gli ultimi quattro modi di indirizzamento ARM in Tabella A2.2 generalizzano i modi di indirizzamento ColdFire con autoincremento e con autodecremento (si veda il Paragrafo 2.10.1), nel seguente senso. In genere, l'uso di un registro per determinare l'indirizzo effettivo di un operando non modifica il contenuto del registro. Il modo relativo a PC costituisce un'ovvia eccezione, perché in tal caso il contenuto del PC, che funge da registro di base, cambia comunque. I modi con autoincremento e autodecremento, che si possono considerare varianti del modo di indirizzamento indiretto, costituiscono altre eccezioni, non ovvie bensì deliberate. Nel primo, l'indirizzo contenuto nel registro viene incrementato subito *dopo* la determinazione dell'indirizzo effettivo dell'operando, mentre nel secondo viene decrementato *prima* di determinare l'indirizzo effettivo. I suddetti quattro modi di indirizzamento ARM generalizzano tali possibilità come segue:

- si impiega un registro di base, la cui variazione di contenuto non è necessariamente unitaria, bensì specificata nell'istruzione, da un numero con segno o da uno scorrimento logico;
- la variazione di contenuto del registro di base segue il suo uso per la determinazione dell'indirizzo effettivo nei modi con post-base, mentre lo precede nei modi con pre-base (questa relazione temporale è pertanto indipendente dal segno della variazione di contenuto).

I modi di indirizzamento con post-base e con pre-base sono anche detti con *writeback*, perché il registro di base viene sovrascritto per effetto del suo uso per il calcolo dell'indirizzo effettivo; ciò non avviene nei modi di indirizzamento che hanno notazione simile a quella dei modi con pre-base ma priva del simbolo "!" di writeback.

Va notato che la presente terminologia per i suddetti quattro modi di indirizzamento ARM si discosta da quella nella documentazione ufficiale ARM, per una ragione di uniformità con la terminologia degli altri modi di indirizzamento in cui si impiega un registro di base.

2.5 Linguaggio assemblativo

Quando è collocata nella parola di memoria l'istruzione macchina ha una forma numerica binaria, direttamente interpretabile ed eseguibile da parte del processore, e consiste in una stringa di bit 0 e 1. Tale forma è adatta al calcolatore ma molto scomoda per il programmatore. Pertanto scrivendo il programma in linguaggio macchina è più opportuno ricorrere alla forma simbolica dell'istruzione, pensata apposta per l'uomo. Nei paragrafi precedenti si è già fatto uso esteso di un linguaggio macchina simbolico di tipo generico, dove le istruzioni hanno nomi espressivi come Load, Store, Add, Branch e altri ancora, invece di darne la rispettiva forma numerica. Ogni calcolatore ha il proprio linguaggio macchina e lo rappresenta in forma simbolica o numerica secondo un insieme di convenzioni ben definito; in linea generale tutte le forme simboliche più comuni sono abbastanza simili a quella generica usata finora e anche tra loro stesse, ma purtroppo non sono mai identiche (invece le forme numeriche possono differire profondamente tra loro. Tra le varie forme simboliche c'è sempre qualche differenza, specialmente per quanto riguarda i nomi delle istruzioni, la notazione dei numeri, come segnalare i modi di indirizzamento e altri particolari ancora. Per passare dal linguaggio macchina simbolico generico a quello specifico di un processore commerciale basta cambiare qualche nome, di solito con abbreviazioni similari corrispondenti, quali LD, ST, ADD, BR e così via. L'insieme completo di regole che definisce la sintassi e la semantica (cioè l'interpretazione) del linguaggio macchina di un processore ben individuato si chiama *linguaggio assemblativo* (*assembly language*). Le regole sono di due tipi: quelle che definiscono la forma simbolica del linguaggio, che interessano qui, e quelle che definiscono la forma numerica binaria e come si passi dalla prima forma alla seconda, che si vedranno più avanti nel Paragrafo 2.13.

L'analisi della correttezza sintattica del programma scritto in linguaggio macchina simbolico e la produzione della forma numerica binaria corrispondente vengono svolte dall'*assemblatore* (*assembler*), un programma apposito simile ai compilatori e sempre compreso nel software di sistema del calcolatore. L'assemblatore prende in ingresso la forma simbolica (la quale di per sé è un file di testo) del programma macchina scritto dall'utente o generato dal compilatore, ne analizza la sintassi, segnala eventuali errori e se tutto risulta corretto genera la forma numerica binaria equivalente creando un file apposito. Per l'uomo la forma numerica è difficile e tediosa da interpretare, giacché consiste in una successione di parole di memoria date come stringhe di bit, ma del resto non occorre mai farlo se non in casi rarissimi.

Il programma di partenza, in forma simbolica, e la forma numerica binaria equivalente costruita dall'assemblatore sono chiamati *programma sorgente* (*source program*) e *programma oggetto* (*object program*), rispettivamente. Da qui in avanti il *linguaggio del processore* è generalmente chiamato *linguaggio macchina* (in forma simbolica o numerica), mentre il termine *assemblatore* indica lo *strumento di traduzione* descritto sopra. Se nel presente paragrafo non si precisa a quale forma di linguaggio macchina si stia facendo riferimento si intende quella simbolica, e questo vale anche altrove. Il funzionamento dell'assemblatore sarà spiegato nel Paragrafo 2.5.2 e nel Capitolo 4; qui si illustrano alcuni aspetti di linguaggio macchina (simbolico).

Talvolta la sintassi del linguaggio macchina è sensibile alla differenza tra lettera maiuscola e minuscola, talvolta no. Qui, come nella maggior parte dei linguaggi assemblativi, si useranno solo lettere maiuscole (salvo nei commenti) per maggiore leggibilità.

Per esempio, un'istruzione Store è scritta come:

```
ST   R2, SOMMA
```

Il codice mnemonico ST rappresenta la sequenza binaria, o *codice operativo* (*OP code*), per l'operazione effettuata dall'istruzione. Il programma assemblatore traduce questo codice mnemonico nel codice operativo binario che il calcolatore riconosce. Il codice operativo mnemonico è seguito da almeno uno spazio bianco o da un carattere di tabulazione, quindi viene

fornita l'informazione che specifica l'operando. Nell'istruzione Store sopra, l'operando sorgente è nel registro R2. Questa informazione è seguita dalla specifica dell'operando di destinazione, separata dall'operando sorgente da una virgola. L'operando di destinazione è nella locazione di memoria che ha l'indirizzo binario rappresentato dal nome SOMMA.

Poiché ci sono diversi possibili modi di indirizzamento per indicare le locazioni degli operandi, un'istruzione assemblativa deve indicare quale modo si sta usando. Per esempio, un valore numerico o un nome, come SOMMA nell'istruzione precedente, possono essere usati per indicare il modo assoluto. Il simbolo "#" generalmente indica un operando immediato. Quindi l'istruzione:

```
ADD   R2, R3, #5
```

somma il numero 5 al contenuto del registro R3 e mette il risultato nel registro R2. Il simbolo "#" non è l'unico mezzo per indicare il modo di indirizzamento immediato. In alcuni linguaggi assemblativi, l'indirizzamento immediato è indicato nel codice operativo mnemonico. Per esempio, la precedente istruzione Add può essere scritta come:

```
ADDI   R2, R3, 5
```

Il suffisso I nel codice mnemonico ADDI indica che il secondo operando sorgente è fornito mediante l'indirizzamento immediato.

L'indirizzamento indiretto è in genere specificato inserendo delle parentesi tonde attorno al nome o al simbolo che indica il puntatore all'operando. Per esempio, se il registro R2 contiene l'indirizzo di un numero in memoria, allora questo numero può essere caricato nel registro R3 usando l'istruzione:

```
LD   R3, (R2)
```

2.5.1 Direttive di assemblatore

Oltre a specificare la sintassi delle istruzioni il linguaggio macchina contiene regole per specificare altre informazioni necessarie a tradurre il programma sorgente nel programma oggetto. Qualsiasi simbolo o nome usato nel programma sorgente deve avere un valore numerico associato. Supponiamo che il nome VENTI sia usato per rappresentare il valore 20. Questa informazione può essere trasferita al programma assemblatore usando una dichiarazione di eguaglianza *equate* come:

```
VENTI   EQU   20
```

La dichiarazione non denota un'istruzione macchina da tradurre in forma numerica binaria affinché il processore la possa eseguire. In effetti, se si esaminasse il programma oggetto ottenuto tramite l'assemblatore non se ne troverebbe traccia. Essa è invece una *direttiva* (o *comando*) *di assemblatore* (*assembler directive*) e si limita a informarlo che il simbolo VENTI dovrebbe essere sostituito dal valore 20 ovunque esso sia presente nel programma.

Per capire meglio come funzioni la direttiva di assemblatore si riprenda il programma in Figura 2.8, in linguaggio macchina generico. Per eseguire il programma su un calcolatore dato occorre prima trasporlo nel linguaggio macchina specifico del processore, comprese tutte le informazioni ausiliarie necessarie. Si supponga che tutte le istruzioni e i dati occupino esattamente una parola ciascuno. Si supponga anche che la memoria sia indirizzabile per byte e che la parola di memoria sia lunga 32 bit. La Figura 2.12 mostra come si potrebbe collocare il programma in memoria; è una scelta tra tante ma vale quanto tutte le altre. Si vedono gli indirizzi delle parole di memoria dove collocare le istruzioni del programma e quelli dove contenere i dati: nel caso specifico i dati seguono le istruzioni, ma potrebbero anche precederle

	100	Load R2, N
	104	Clear R3
	108	Move R4, #NUM1
CICLO	112	Load R5, (R4)
	116	Add R3, R3, R5
	120	Add R4, R4, #4
	124	Subtract R2, R2, #1
	128	Branch_if_[R2]>0 CICLO
	132	Store R3, SOMMA
		⋮
SOMMA	200	
N	204	150
NUM1	208	
NUM2	212	
		⋮
NUM*n*	804	

Figura 2.12 Modo in cui è collocato in memoria il programma che calcola la somma della lista di numeri.

perché in fondo le parole di memoria sono tutte identiche e pertanto interscambiabili. Affinché l'assemblatore riesca a tradurre il programma sorgente e produrre un programma oggetto completo e funzionante, deve risolvere i problemi seguenti:

- come interpretare nomi e simboli assegnando loro valori numerici;
- dove collocare in memoria le istruzioni macchina;
- dove collocare in memoria gli operandi e i risultati del programma.

Per fornire all'assemblatore le informazioni necessarie a risolverli il programma sorgente può essere scritto come mostra la Figura 2.13. Il programma inizia con la direttiva di assemblatore, ORIGIN, che dice all'assemblatore in quale parte della memoria inserire le istruzioni che seguono. Essa specifica che le istruzioni del programma oggetto dovranno essere caricate nella memoria a partire dall'indirizzo 100. Essa è seguita dalle istruzioni del programma sorgente scritte mediante appositi codici mnemonici e sintassi. Si noti l'uso dell'espressione:

BGT R2, R0, CICLO

per rappresentare un'istruzione che effettua l'operazione:

	Etichetta di indirizzo di memoria	Operazione	Indirizzi o dati
Direttiva di assemblatore		ORIGIN	100
Dichiarazioni che generano istruzioni macchina		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	CICLO:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, CICLO
		ST	R3, SOMMA
		prossima istruzione	
Direttive di assemblatore		ORIGIN	200
	SOMMA:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

Figura 2.13 Programma che calcola la somma della lista di numeri codificato in linguaggio assemblativo.

Branch_if_[R2]>0 CICLO

La seconda direttiva ORIGIN dice all'assemblatore dove inserire nella memoria il blocco di dati che segue. In questo caso, la locazione specificata ha l'indirizzo 200. Questa è la locazione in cui verrà memorizzata la somma finale. Uno spazio di 4 byte per la somma viene riservato per mezzo della direttiva di assemblatore RESERVE. La successiva parola, all'indirizzo 204, deve contenere il valore 150 che è il numero di elementi della lista. La direttiva DATAWORD è usata per informare l'assemblatore di questo requisito. La successiva direttiva RESERVE dichiara che un blocco di memoria di 600 byte deve essere riservato per i dati. La direttiva non causa il caricamento di dati in tali locazioni. I dati possono essere caricati in memoria usando una procedura di input, come spiegheremo nel Capitolo 3. L'ultima dichiarazione nel programma sorgente è la direttiva END, che dice all'assemblatore che questa è la fine del testo del programma sorgente.

Precedentemente abbiamo descritto come la direttiva EQU possa essere usata per associare un valore specifico, che può essere un indirizzo, a un particolare nome. Un modo diverso per associare indirizzi a nomi o etichette è illustrato in Figura 2.13. Ogni espressione che risulta in istruzioni o dati che sono inseriti in una locazione di memoria può essere fornita di un'etichetta di un indirizzo di memoria. L'assemblatore assegna automaticamente l'indirizzo della locazione all'etichetta. Per esempio, nel blocco di dati che segue la seconda direttiva ORIGIN, usiamo le etichette SOMMA, N e NUM1. Poiché alla prima espressione RESERVE dopo la direttiva ORIGIN è stata data l'etichetta SOMMA, al nome SOMMA è assegnato il valore 200. Quando SOMMA è incontrata nel programma, essa sarà sostituita con questo valore. Usare SOMMA come etichetta in questo modo è equivalente a usare la direttiva di assemblatore

SOMMA EQU 200

Analogamente, alle etichette N e NUM1 sono rispettivamente assegnati i valori 204 e 208, poiché rappresentano gli indirizzi delle due locazioni di parole che seguono immediatamente la locazione della parola con indirizzo 200.

Secondo la sintassi della maggior parte dei linguaggi macchina in generale la riga contenente l'istruzione deve avere la struttura seguente:

etichetta *operazione* *operandi* *commento*

I quattro campi vanno separati tramite uno o più spazi, tabulazione o altro separatore valido. Di solito l'inizio del commento è marcato dai caratteri ";", "%", "!" o simili. L'*etichetta* è facoltativa e viene associata all'indirizzo della parola di memoria che l'assemblatore assegna all'*istruzione*. Se invece si tratta di una *direttiva* che riserva una porzione di memoria per contenere dati, all'etichetta viene associato l'indirizzo del byte iniziale della parola o porzione assegnata. In Figura 2.13 ci sono quattro etichette: CICLO, SOMMA, N e NUM1.

Il campo *operazione* contiene una direttiva di assemblatore o il codice operativo mnemonico dell'istruzione desiderata. Il campo *operandi* contiene l'informazione di indirizzamento per accedere agli operandi. Il campo *commento* viene ignorato del tutto dall'assemblatore e serve solo al programmatore come documentazione.

Ecco dunque illustrate le caratteristiche essenziali del linguaggio macchina. Nei linguaggi specifici dei vari processori commerciali ci sono poi moltissimi particolari che li diversificano a seconda del calcolatore e li rendono più o meno efficienti da eseguire o difficili da apprendere.

2.5.2 Assemblaggio ed esecuzione

Il programma sorgente in linguaggio macchina va *assemblato* per trasformarlo in programma oggetto e poi eseguirlo. Il processo di assemblaggio è compito dell'*assemblatore*. Esso trasforma l'istruzione simbolica mettendola in forma numerica binaria, sostituendo ogni simbolo (etichetta, codice mnemonico, nome di registro e segnale di modo di indirizzamento) con il valore numerico associato.

Per fare tutto ciò l'assemblatore deve decidere a quali indirizzi in memoria collocare istruzioni e blocchi di dati partendo dall'indirizzo specificato tramite la direttiva ORIGIN, preparare i valori iniziali dei dati specificati mediante la direttiva DATAWORD e riservare spazio in memoria per i blocchi di byte specificati mediante la direttiva RESERVE.

Si vede dunque come uno dei compiti primari dell'assemblatore sia quello di determinare il valore numerico associato al simbolo. Talvolta la risposta è immediata, come per l'etichetta dichiarata tramite la direttiva EQU. Quando, però, si tratta di decidere quale sia il valore da dare all'etichetta di un'istruzione non è più così semplice, perché il valore è l'indirizzo dell'istruzione stessa, che non è noto immediatamente. L'assemblatore deve allora tenere traccia degli indirizzi dove collocare progressivamente le istruzioni macchina mentre assembla il programma e deve sapere quali siano gli *ingombri* delle istruzioni stesse, cioè quante parole occupi ciascuna di esse (è facile fintantoché le istruzioni occupano una parola ciascuna, ma non è sempre così). Per esempio, le etichette CICLO e SOMMA del programma in Figura 2.13 vengono in tale modo associate agli indirizzi 112 e 200, rispettivamente.

Talvolta l'assemblatore non deve sostituire direttamente l'etichetta con l'indirizzo corrispondente. Per esempio, nell'istruzione di salto non sarebbe corretto sostituire l'etichetta rappresentante la destinazione di salto con l'indirizzo numerico. Infatti nell'istruzione di salto va solitamente specificato lo spiazzamento relativamente all'indirizzo corrente, vale a dire la differenza tra i due indirizzi di destinazione e corrente. L'assemblatore deve allora calcolare tale differenza e usarla per codificare l'istruzione in forma numerica.

Nel Paragrafo 2.13 sarà mostrato come si possano realizzare le istruzioni di salto.

Approfondimento applicativo 2.3

Direttive di assemblatore e sintassi di GNU Assembler

Terminologia e sintassi delle direttive di assemblatore variano, non solo da un processore all'altro considerandone i linguaggi assemblativi nativi, ma anche fra diversi assemblatori per lo stesso processore.

Le direttive sono anche dette *pseudo-operazioni* (*pseudo-op*) nella documentazione degli assemblatori nativi ColdFire, ARM e IA-32, come pure nel *GNU Assembler* (GAS), che può tradurre programmi in linguaggio macchina per tutti i processori qui considerati e per molti altri, mentre il termine *pseudoistruzioni* nella documentazione NIOS II designa istruzioni disponibili nel linguaggio assemblativo simbolico ma che l'assemblatore non traduce in istruzioni macchina in formato binario, bensì in (sequenze di) altre istruzioni simboliche, che vengono poi tradotte in formato binario.

La sintassi delle direttive negli assemblatori nativi NIOS II e ColdFire è conforme alla sintassi GAS, ma quella dell'assemblatore *inline* del secondo, che ha un insieme ridotto di direttive, è mutuata da quella nativa dei processori Motorola M680x0, dai quali sono evoluti i processori ColdFire; si usa tale assemblatore per istruzioni assemblative incorporate in programmi C o C++..

Alcune delle direttive principali di GAS espandono e completano quelle generiche viste prima e sono riportate in Tabella A2.3 in relazione con quelle generiche, dove possibile.

Tabella A2.3 Direttive di GAS e relazione con quelle generiche

Assemblatore "generico"			GNU GAS			Significato in GAS
SIM	EQU	val		.equ	SIM, val_{32}	Dichiara il simbolo "SIM" con valore "val_{32}" a 32 bit
			SIM =		val_{32}	
ETI	ORIGIN	num	ETI	.org	num	Allinea l'indirizzo (virtuale) di impianto del programma al valore "num"
ETI	RESERVE	num	ETI	.space	num	Alloca come spazio di memoria un numero di byte pari a "num" (senza dare loro valore iniziale)
ETI	DATAWORD	val	ETI	.byte	val_8 [, val_8, ...]	Alloca come spazio di memoria un byte (o una lista di byte), dandogli (dando loro) valore iniziale "val_8"
			ETI	.word	val_{16} [, val16, ...]	Idem, ma con parole a 16 bit
			ETI	.long	val_{32} [, val_{32}, ...]	Idem, ma con parole a 32 bit
			ETI	.quad	val_{32} [, val_{32}, ...]	Idem, ma con parole a 64 bit
	non esiste		ETI	.ascii	"stringa"	Alloca come spazio di memoria un numero di byte pari alla lunghezza della stringa, e dà loro come valori iniziali i car. della stringa
	non esiste		ETI	.asciz	"stringa"	Idem, aggiunge '\0'
	non esiste		ETI	.data		Dichiara il segmento dati del programma
	non esiste		ETI	.text		Dichiara il segmento codice del programma
	non esiste		ETI	.title	"stringa"	Dichiara il titolo del programma (pura informazione)
	END		non esiste			Specifica la fine del programma

Tutte le direttive hanno nome iniziante con il carattere "." (punto) per distinguerle dai codici mnemonici delle istruzioni macchina. Nella Tabella A2.3 compaiono alcuni metasimboli per indicare i tipi di argomenti da dare alle varie direttive. Eccone la spiegazione: le parentesi quadre [e] indicano che il contenuto è facoltativo oppure si può ripetere due o più volte interponendo "," (virgola) come separatore; i metasimboli val_8, val_{16}, val_{32} e val_{64} indicano valori numerici interi qualunque, senza o con segno, rappresentabili in binario a 8, 16, 32 o 64 bit, rispettivamente, che possono essere specificati da un'espressione costituita da simboli già definiti, parentesi e operatori aritmetici e logici; il metasimbolo num indica un generico numero intero positivo o nullo, specificabile da un'espressione; i metasimboli ETI e SIM indicano etichette o simboli qualunque; infine il metasimbolo stringa indica una stringa di uno o più caratteri ASCII (è ammessa anche la stringa vuota).

La Tabella A2.4 mostra le varianti sintattiche delle direttive appena introdotte negli assemblatori nativi dei quattro processori e nell'assemblatore ColdFire *inline*; per i processori IA-32 si dà la sintassi dell'assemblatore MASM, non nativo ma in largo uso. Per ulteriori dettagli su tali assemblatori, rilevanti anche ad altri aspetti trattati negli approfondimenti successivi, si possono consultare le fonti bibliografiche [9, 2, 8, 5, 6, 11, 12]. Alcune precisazioni sulle convenzioni adottate in Tabella A2.4 servono alla sua corretta interpretazione.

Si è omessa l'indicazione di argomenti associati alle direttive (simboli, espressioni ecc.), che generalmente ha la stessa sintassi che si trova in GAS, con alcune eccezioni esplicitamente notate. Il metasimbolo "|" separa forme alternative equivalenti, o quasi (per le differenze si veda la documentazione ufficiale degli assemblatori). Infine, le direttive ColdFire *inline* sono qui distinte da quelle dell'assemblatore nativo dal fatto che queste ultime iniziano con il carattere ".", tuttavia si tratta di una distinzione di comodo, poiché tale carattere iniziale è opzionale in entrambi gli assemblatori.

Tabella A2.4 Direttive di assemblatore GAS, NIOS II, ColdFire, ARM e MASM

GAS	NIOS II	ColdFire	ARM	MASM
.org	.org	.org		ORG
.equ | =	.equ	.equ | =	EQU | =	EQU | =
.space | .skip	.skip	.space | ds.*t*	SPACE	D*t n* DUP(*v*)
.byte	.byte	.byte | dc.b	DCB	DB
.short | .hword | .word	.hword	.short | dc.w	DCW*a*	DW
.long | .int | .word	.word	.long | dc.l	DCD*a*	DD
.quad			DCQ*a*	DQ
.ascii	.ascii	.ascii	DCB	
.asciz | .string	.asciz	.asciz	DCB "*s*",0	
.data	.data	.data	AREA *s* DATA	.DATA
.text	.text	.text	AREA *s* CODE	.CODE
		entry	ENTRY	
		.textequ		TEXTEQU
.req			RN	
.title			TTL	TITLE
.end	.end		END	END *e*

Legenda:
t suffisso del codice mnemonico: tipo (dimensione) di ciascun elemento ColdFire: $t \in$ {b,w,l}; MASM: $t \in$ {B,W,D,Q};
n numero di elementi;
v valore iniziale di ogni elemento, "?" se dati non inizializzati;
a suffisso del codice mnemonico: nessun allineamento se *a* = U, altrimenti *a*, assente e allineamento a indirizzo pari se DCW, multiplo di 4 se DCD o DCQ;
s stringa di caratteri ASCII (nella direttiva AREA: nome del segmento);
e etichetta (opzionale): indirizzo di inizio dell'esecuzione del programma.

Le seguenti note spiegano altri dettagli su alcune direttive in Tabella A2.4.

L'assemblatore MASM non ha una direttiva distinta che corrisponda a .space in GAS, tuttavia si ottiene lo stesso risultato qualificando con DUP una delle successive quattro direttive; il qualificatore permette anche di specificare un valore iniziale comune a tutti i dati per i quali la direttiva riserva spazio.

Similmente, l'assemblatore ARM impiega un qualificatore, della sua direttiva AREA, per specificare se il segmento contenga dati o codice (cioè istruzioni). Lo stesso assemblatore ottiene l'effetto delle direttive GAS .ascii e .asciz con la direttiva DCB, che inizializza in tal caso una sequenza di byte con una stringa data, o con la concatenazione di stringhe separate da ",".

Gli assemblatori ARM e ColdFire *inline* hanno direttive per dichiarare un'istruzione, tramite la sua etichetta, quale quella di inizio dell'esecuzione. GAS usa l'etichetta _start a tal fine, dunque non necessita di una tale direttiva. In MASM, l'etichetta di inizio è un argomento opzionale della direttiva END.

Infine, due direttive in Tabella A2.4 che non corrispondono a direttive GAS in Tabella A2.3 differiscono dalle altre viste finora perché i loro effetti modificano l'interpretazione del successivo codice sorgente.

La direttiva ARM RN (.req nella sintassi GAS, ma solo per architettura ARM), definisce un *sinonimo* (*alias*) per un nome di registro; il sinonimo può essere usato nelle successive istruzioni, dove l'assemblatore ne sostituirà ogni occorrenza con il nome del registro *prima* di generare l'istruzione macchina. Questa direttiva può essere utile a migliorare la leggibilità dei programmi, permettendo, per esempio, di usare nomi di variabili invece che di registri.

La direttiva ColdFire .textequ (TEXTEQU in MASM) generalizza l'idea dei sinonimi a stringhe di testo arbitrarie. Per esempio, con direttive quali "dc.b .textequ ".byte"", "dc.w .textequ ".short"" ecc. si può incorporare un codice assemblativo preesistente o *inline* in codice ColdFire nativo, conforme alla sintassi GAS. L'idea di agire sul *testo* del programma simbolico prima della traduzione ha altre generalizzazioni nelle direttive per la definizione di *macroistruzioni*, trattate nell'Approfondimento applicativo 2.4.

Si danno ora i tratti essenziali della sintassi concreta dell'assemblatore GAS, che riconosce, analizza e traduce i linguaggi macchina di molti dei processori esistenti. Basta invocare GAS specificando a quale processore il programma da tradurre si riferisca. Tutti i linguaggi riconosciuti da GAS condividono la medesima sintassi generale di numeri, simboli, direttive e commenti, e si differenziano solo per i codici mnemonici (che ovviamente sono specifici per ciascun processore) e talvolta per qualche dettaglio su come specificare gli operandi dell'istruzione e i modi di indirizzamento (per esempio, spesso processori diversi usano nomi di registro specifici). Peraltro GAS è ampiamente tollerante e ammette anche qualche commistione di sintassi purché non dia luogo ad ambiguità e inoltre è capace di correggere automaticamente gli errori più banali, come avere dimenticato un carattere separatore e simili.

In GAS il file contenente il programma in linguaggio macchina simbolico è suddiviso in linee con la forma seguente:

etichetta: codice mnemonico lista operandi commento

sostanzialmente identica a quella generica data nel Paragrafo 2.5.1. L'etichetta deve iniziare in prima colonna ed è seguita dal carattere ":", che si può omettere se non c'è ambiguità; nel seguito è generalmente omesso. È vietato etichettare linee diverse con la medesima etichetta.

Etichette e simboli sono sequenze di caratteri alfabetici (comunemente maiuscoli, ma non è d'obbligo, tuttavia l'assemblatore distingue fra maiuscoli e minuscoli) e numerici, e devono iniziare con una lettera o con uno dei caratteri "._". I valori numerici interi possono

essere rappresentati in decimale, binario (naturale o complemento a due) ed esadecimale (è ammessa anche la rappresentazione ottale). Il numero decimale non ha prefisso, quello binario ha il prefisso "0b" e quello esadecimale "0x". Il singolo carattere ASCII si denota come 'c, dove "c" è un qualunque elemento della tabella ASCII (si veda la Tabella 1.1) o la sua *sequenza di fuga* (*escape sequence*), che ne è una rappresentazione mediante caratteri stampabili; per esempio il carattere di *nuova linea* (*new line*) si denota come '\n, quello di tabulazione orizzontale (*horizontal tab*) come '\t e così via. GAS riconosce anche i numeri frazionari e li codifica in formato IEEE 754 (si veda il Capitolo 9), ma qui tale funzione non servirà.

In GAS i codici mnemonici, ovvero i nomi delle istruzioni macchina, sono in genere brevi stringhe di tipo alfabetico. Ciascun processore ha il proprio insieme di codici mnemonici.

La sintassi dei modi di indirizzamento usabili nelle istruzioni macchina è più o meno la stessa per tutti i linguaggi macchina riconosciuti da GAS, con ben poche differenze. GAS riconosce la sintassi universale AT&T, un formalismo concepito in modo da valere per qualunque processore: per i modi di indirizzamento essa è sostanzialmente identica a quella mostrata in Tabella 2.1, estesa ai modi di indirizzamento in uso nei processori CISC (si veda il Paragrafo 2.10.1), e ha ordinamento degli operandi delle istruzioni macchina costantemente di tipo sorgente-destinazione, al contrario di quello prevalente nel linguaggio assemblativo generico qui usato.

Per ogni altro dettaglio relativo a GAS si rimanda alla documentazione relativa, reperibile sul sito ufficiale GNU (www.gnu.org), dove è inclusa nella documentazione della collezione di strumenti *binutils*, della quale l'assemblatore, denominato *as*, fa parte (www.sourceware.org/binutils).

L'assemblatore crea un file su disco contenente il programma oggetto in forma numerica binaria. Il programma oggetto viene poi caricato in memoria centrale quando va posto in esecuzione e l'operazione è svolta dal componente di software di sistema chiamato *caricatore* (*loader*), che deve già trovarsi in memoria centrale. Durante il caricamento il programma oggetto viene sottoposto ad alcune trasformazioni conclusive che danno alle istruzioni la forma numerica definitiva con cui vengono collocate nelle parole di memoria destinate a contenerle. Per esempio il caricatore deve sapere a quale indirizzo iniziale posizionare il programma in memoria centrale. L'informazione è data dalla direttiva ORIGIN e l'assemblatore la mette in testa al file contenente il programma oggetto, dove il caricatore si aspetta di trovarla. Dopo avere scritto in memoria il programma oggetto il caricatore ne lancia l'esecuzione passando all'istruzione iniziale del programma stesso, il cui indirizzo è specificato tramite la direttiva START e che è anch'esso inserito in testa al file. In effetti il file oggetto ha in testa una struttura dati apposita (e piuttosto complessa) che non fa propriamente parte del programma, bensì contiene un po' di informazioni ausiliarie necessarie per effettuare il processo di caricamento, e che è seguita dalla lista delle istruzioni effettive del programma. Il caricatore estrae dalla struttura dati quanto gli serve per definire tutti gli aspetti della procedura di caricamento in memoria e messa in esecuzione del programma, ma non la scrive in memoria centrale, in cui colloca solo le istruzioni e i dati aventi valore iniziale predichiarato.

Una volta avviata, l'esecuzione del programma procede fino al termine naturale a meno che non si verifichino errori di esecuzione. L'assemblatore segnala solo gli errori di sintassi e di assemblaggio. Per trovare gli errori di esecuzione si ricorre al programma di *esecuzione controllata* (*debugger*), un altro componente di software di sistema tramite cui l'utente può arrestare a piacimento l'esecuzione in un punto intermedio ed esaminare il contenuto della memoria e dei registri di processore. L'argomento verrà ripreso con maggiore dettaglio nel Capitolo 4.

2.5.3 Notazione per i numeri

Quando lavora con i numeri, l'uomo trova naturale servirsi della familiare notazione decimale, che fa capo alla rappresentazione in base 10. Nella memoria del calcolatore però si trovano solo numeri binari, di solito in rappresentazione naturale o in complemento a due. Scrivendo il programma in linguaggio macchina torna talvolta utile servirsi di numeri dati direttamente come sequenze di bit. La maggior parte degli assemblatori permette di denotare numeri in decimale o altra forma, secondo convenzioni sintattiche ad hoc. Per esempio, 93 è valore decimale e come numero binario naturale a 8 bit si scrive 01011101. Se lo si vuole usare come operando in modo immediato lo si può denotare in decimale come segue:

ADDI R2, R3, 93

oppure come un numero binario identificato da un simbolo per il prefisso che è specifico per l'assemblatore, quale, per esempio, un segno di percentuale, come in:

ADDI R2, R3, %01011101

Si può abbreviare ricorrendo alla *notazione esadecimale* (*hexadecimal* o *hex*), ovvero in base 16: partendo dal bit meno significativo si sostituisce ogni quaterna di bit consecutivi con una sola cifra hex: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E e F, per rappresentare le quaterne 0000, 0001, 0010, ..., 1111, rispettivamente. Per esempio, il numero decimale 93 si scrive 5D in notazione hex, perché 93 = 01011101 = 01011101 = 5D. La notazione hex è un'estensione naturale della notazione BCD, coincidendo con questa nelle prime dieci cifre. Spesso in linguaggio macchina per rappresentare il numero hex si usa il prefisso "0x" (identico alla notazione del linguaggio C) o da un prefisso (che è un segno di dollaro). Ecco come va riscritta in notazione hex l'istruzione di addizione vista sopra:

ADDI R2, R3, 0x5D

2.6 Gestione di pila

La disposizione dei dati in memoria centrale è molto varia, come lo sono le strutture dati che si possono dichiarare in linguaggio di alto livello. Qui interessa una struttura dati fondamentale, la *pila*: si tratta di una lista di elementi, alla quale però si applicano alcune restrizioni circa il modo di inserirli ed estrarli. Si può lavorare con la pila solo agendo a uno degli estremi, la *cima* (*top*) della pila; l'altro estremo è chiamato *base* o *fondo* (*bottom*) della pila. Si giustificano i nomi immaginando la pila come una serie di elementi poggianti in verticale l'uno sopra l'altro, come appunto una pila di libri. Sulla pila sono consentite solo le due operazioni seguenti: porre un nuovo elemento sulla cima, diventando quindi la nuova cima, oppure togliere l'elemento correntemente sulla cima, e l'elemento sottostante diventa la nuova cima, a meno che l'elemento tolto non fosse l'unico nella pila, nel qual caso questa resta *vuota*, condizione in cui è consentita solo la prima operazione. Ne consegue che gli elementi inseriti per primi in pila sono gli ultimi estraibili; ciò giustifica il nome LIFO (*last-in-first-out*) che si dà a tale modo di operare. I nomi inglesi tradizionali delle due operazioni sono *push*, per porre in cima l'elemento, e *pop*, per togliere dalla cima l'elemento. In questo volume i due nomi sono resi con i termini espressivi *impila* per *push* e *spila* per *pop*.

Si possono organizzare i dati conservandoli nella pila: i dati aggiunti progressivamente occupano parole di memoria consecutive nel verso di crescita della pila. Qui si adotta la convenzione di fare crescere la pila (cioè aggiungere elementi in cima) nel verso degli indirizzi decrescenti, che del resto è la più comune anche per i processori; alcuni però adottano la con-

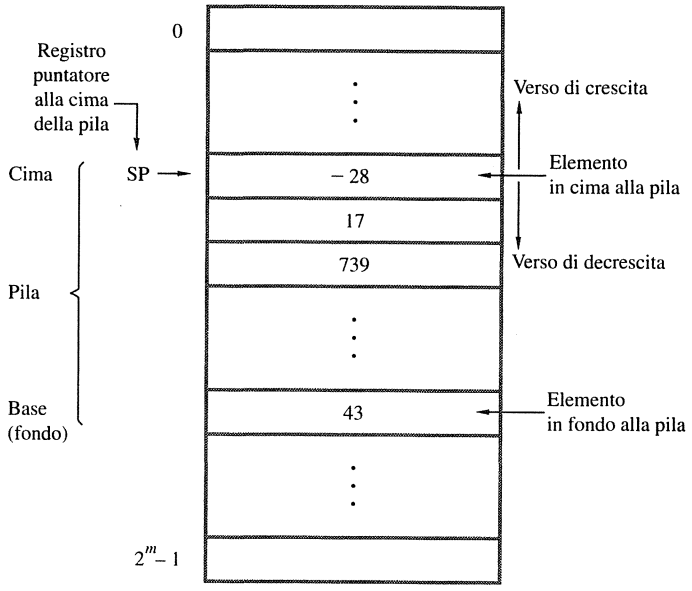


Figura 2.14 Struttura della pila di memoria.

venzione opposta (che qui non si usa mai). Siccome di solito nei libri la memoria è disegnata incrementando gli indirizzi scendendo verso il fondo della pagina, secondo la convenzione sopradetta la pila cresce andando verso l'alto nella pagina, cosicché il senso dei termini *cima*, *base*, *crescere* o *alzarsi* e *decrescere* o *abbassarsi* risulta visivamente ovvio; così è anche in questo libro.

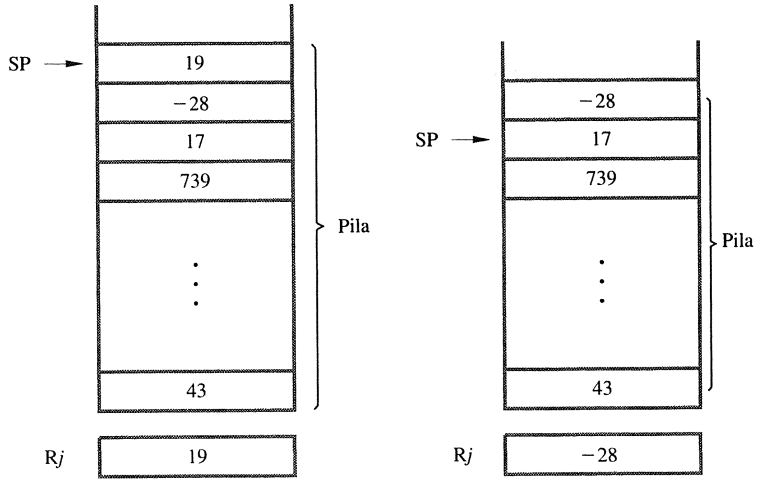
La Figura 2.14 mostra la struttura della pila (con le convenzioni fatte sopra), supponendo che l'elemento abbia dimensione di parola. Se si preferisce si può immaginare che l'elemento sia un numero intero rappresentato in complemento a due su un numero di bit pari a quello della parola di memoria. Cima e base sono gli elementi −28 e 43, rispettivamente. Nel processore si trova un registro apposito chiamato *puntatore alla pila* o SP (*Stack Pointer*), che funziona appunto come puntatore all'elemento correntemente in cima alla pila. Si noti che il registro SP punta esattamente all'elemento in cima alla pila, non a quello subito sopra che ha indirizzo diminuito in modo conforme alla dimensione del dato, e non va considerato appartenente alla pila.

Se si suppone che la memoria sia indirizzabile per byte e abbia parola da 32 bit, si ottiene l'operazione *impila* o *push* mediante la coppia di istruzioni seguente, in linguaggio macchina generico:

Subtract SP, SP, #4

Store R*j*, (SP)

La prima istruzione diminuisce di 4 unità il contenuto del registro SP. Assumendo che il nuovo elemento da inserire in cima alla pila sia nel registro R*j* del processore, l'istruzione Store inserirà tale valore sulla pila. Queste due istruzioni copiano la parola da R*j* sulla cima della



(a) Operazione PUSH o IMPILA (b) Operazione POP o SPILA

Figura 2.15 Effetti delle operazioni di pila.

pila, decrementando di 4 il puntatore alla pila prima dell'operazione di memorizzazione. Si ottiene l'operazione *spila* o *pop* mediante

Load R*j*, (SP)

Add SP, SP, #4

Queste due istruzioni estraggono il valore dalla cima della pila e lo inseriscono nel registro R*j*, quindi incrementano di 4 il puntatore alla pila così che esso punti al nuovo elemento in cima. La Figura 2.15 mostra l'effetto di entrambe le operazioni *impila* e *spila*.

2.7 Gestione di sottoprogramma

Spesso il programma in esecuzione deve ripetere per più volte un compito identico, ma ogni volta lavorando su dati differenti. Ricorrere al meccanismo di *sottoprogramma* o *routine* è un modo di programmare utile appunto per gestire efficientemente attività ripetitive. Per esempio, si potrebbe ricorrere a una routine per calcolare ripetutamente la funzione seno circolare (sin) o iperbolico (sinh), oppure per ordinare una lista di elementi.

A livello di linguaggio macchina, se un dato compito è richiesto in più punti del programma basterebbe inserire in ogni punto una copia dalla lista di istruzioni che lo esegue. Chiaramente però si sprecherebbe spazio di memoria; in effetti si usa di rado tale soluzione e solo per compiti brevi ripetuti poche volte. Invece è molto desiderabile includere nel programma una sola copia della lista di istruzioni costituenti la routine che svolge il compito. Ogniqualvolta il programma deve eseguire il compito, esso sospende la normale sequenza di esecuzione ed effettua un salto all'unica copia presente della routine corrispondente e quando essa è terminata riprende l'esecuzione dal punto dove l'aveva sospesa.

Quasi tutti i linguaggi di alto livello comprendono il meccanismo di sottoprogramma e, come si vedrà qui, anche il linguaggio macchina lo prevede. Il programma che effettua il salto è detto *chiamante* (*calling program* o *caller*), mentre il sottoprogramma a cui si passa è detto *chiamato* (*called program* o *callee*). L'istruzione macchina di salto tramite cui si passa a sottoprogramma è speciale ed è detta *istruzione di chiamata a sottoprogramma* (*call instruction*) o *a routine*.

Quando la routine termina si deve fare in modo che il programma chiamante riprenda la propria esecuzione tornando all'istruzione consecutiva a quella di chiamata alla routine. Si dice che la routine *rientra* al programma chiamante e l'istruzione macchina che effettua questo passaggio cruciale è detta *istruzione di rientro* (*return instruction*). Siccome la routine può benissimo essere chiamata in più punti del programma chiamante, ma in genere non termina con altrettante istruzioni di rientro differenti (spesso anzi l'istruzione di rientro che la conclude è una sola), ci deve essere un meccanismo capace di garantire che il rientro si diriga verso la chiamata giusta. L'indirizzo del punto di rientro è facile da individuare: è il valore contenuto nel registro contatore di programma PC del processore quando si esegue l'istruzione di chiamata a routine, naturalmente già aggiornato in modo da puntare all'istruzione consecutiva a quella di chiamata. Pertanto il meccanismo di rientro si basa in modo essenziale sull'idea di conservare in qualche modo tale indirizzo per farne uso al momento opportuno.

La tecnica tramite cui si riescono a gestire chiamata e rientro è detta *meccanismo di collegamento a routine* (*subroutine linkage method*). Ce ne sono versioni differenti: quella più semplice prevede di scrivere e conservare l'indirizzo di rientro, ottenuto come visto sopra, in un elemento di memoria apposito collocato all'interno del processore. Per solito si tratta di un registro di uso speciale dedicato esclusivamente a tale funzione, chiamato appunto *registro di collegamento* (*link register*): quando la routine termina eseguendo l'istruzione di rientro quest'ultima riporta correttamente il flusso di esecuzione al programma chiamante mediante salto, servendosi proprio del registro di collegamento come puntatore alla destinazione.

Come si è detto in precedenza, si effettua la chiamata a sottoprogramma tramite un'istruzione di salto speciale. Eccone le operazioni:

- copia il contenuto del registro PC nel registro di collegamento (talvolta si dice anche che PC viene *salvato* nel registro di collegamento);
- salta all'indirizzo di destinazione dato nell'istruzione stessa, secondo uno dei modi di indirizzamento disponibili.

Anche l'istruzione di rientro a programma chiamante è una variante di istruzione di salto. Essa salta all'indirizzo di rientro contenuto nel registro di collegamento e precedentemente scrittovi dall'istruzione di chiamata. La Figura 2.16 mostra entrambi i meccanismi, di chiamata e rientro.

2.7.1 Annidamento e pila

In linguaggio macchina (come anche in quello di alto livello) è lecito che una routine ne chiami un'altra o se stessa ricorsivamente. Tale modo di programmare si chiama *annidamento di sottoprogramma* (*subroutine nesting*) ed è pratica comune. Pertanto prima di chiamare una seconda routine è necessario provvedere a salvare il contenuto del registro di collegamento in una qualche struttura dati apposita, per *recuperarlo* o *ripristinarlo* in seguito; se non lo si facesse, infatti, al momento della seconda chiamata di routine l'indirizzo di rientro salvato nel registro al momento della prima chiamata andrebbe perso.

La successione di chiamate annidate può andare avanti a profondità arbitraria. Prima o poi però l'ultima routine chiamata, la più recente, termina e rientra alla rispettiva routine chiamante. Dunque l'indirizzo relativo al primo rientro eseguito è l'ultimo generato. La situazione si ripete per tutti i passi di rientro che concludono progressivamente la successione di routine.

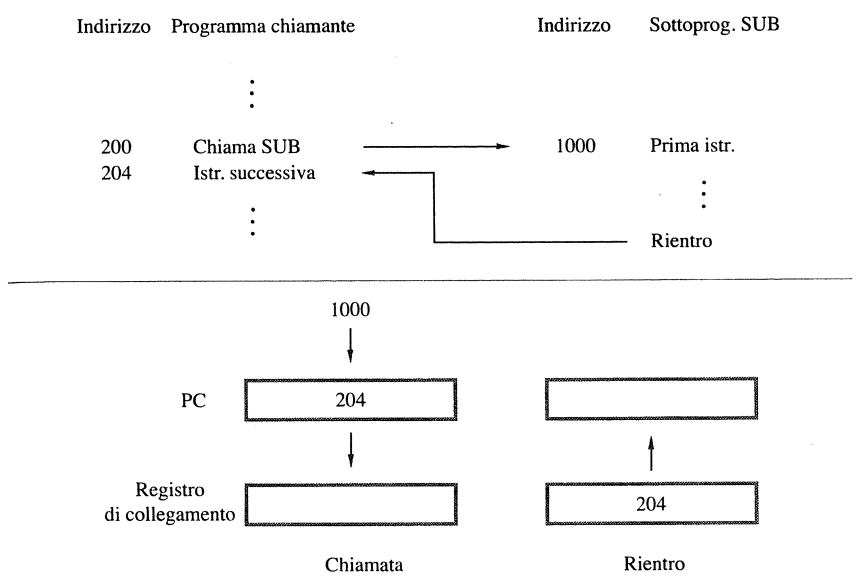


Figura 2.16 Meccanismo di chiamata e rientro basato su registro di collegamento.

Si vede subito come i rientri siano disposti in ordine inverso rispetto alle chiamate, cioè l'ordine sia di tipo LIFO (*last-in-first-out*), ovvero l'ultimo elemento che entra sarà il primo a uscire. Tutto ciò suggerisce che la struttura dati adatta a conservare gli indirizzi di rientro sia proprio la pila di memoria.

Numerosi processori impilano l'indirizzo di rientro direttamente come parte dell'esecuzione dell'istruzione stessa di chiamata a sottoprogramma, senza prima scriverlo nel registro di collegamento (che a questo punto non serve neppure più). Il registro SP, puntatore alla cima della pila, viene usato per gestire la pila, dunque l'istruzione di chiamata ne modifica il contenuto: essa impila il contenuto del registro PC del processore, che rappresenta l'indirizzo di rientro, e naturalmente aggiorna il registro SP di conseguenza. L'istruzione di rientro da sottoprogramma spila l'indirizzo in cima alla pila e lo riscrive nel registro PC del processore; anch'essa aggiorna il registro SP di conseguenza. Se le istruzioni di chiamata e rientro lavorano in pila si dice che il meccanismo di chiamata a sottoprogramma si serve della pila in *modo implicito*, cioè automatico.

2.7.2 Passaggio di parametri

Quando si chiama una routine occorre spesso passarle parametri, vale a dire operandi o indirizzi, e se la routine è una funzione quando termina occorre anche recuperarne il risultato. Il processo di scambio di informazione tra programma chiamante e sottoprogramma chiamato è denominato *passaggio di parametri* (*parameter passing*), il quale si può ottenere in diversi modi. Due sono quelli principali: passare i parametri tramite i registri del processore (quelli di uso generale) o tramite parole di memoria a indirizzi prefissati, dove la routine li trova quando è in esecuzione, oppure passarli tramite la pila, che si usa comunque per conservare l'indirizzo di rientro quando ciò occorra.

Programma chiamante

	Load	R2, N	Il parametro 1 è la dimensione della lista
	Move	R4, #NUM1	Il parametro 2 è la locazione della lista
	Call	TOTALE	Chiama il sottoprogramma
	Store	R3, SOMMA	Immagazzina il risultato

Sottoprogramma

TOTALE:	Subtract	SP, SP, #4	Salva in pila R5
	Store	R5, (SP)	
	Clear	R3	Inizializza la somma a 0
CICLO:	Load	R5, (R4)	Preleva il prossimo numero
	Add	R3, R3, R5	Aggiungi questo numero alla somma
	Add	R4, R4, #4	Incrementa di 4 il puntatore
	Subtract	R2, R2, #1	Decrementa il contatore
	Branch_if_[R2]>0	CICLO	
	Load	R5, (SP)	Ripristina il contenuto di R5
	Add	SP, SP, #4	
	Return		Rientra al programma chiamante

Figura 2.17 Routine che calcola la somma della lista di numeri e meccanismo di passaggio di parametri tramite registri.

Usare i registri del processore è una tecnica diretta ed efficiente. La Figura 2.17 mostra come trasformare in routine il programma che calcola la somma della lista di numeri, passando i parametri nei registri. La lunghezza della lista, contenuta nella parola di memoria di indirizzo N, e l'indirizzo NUM1 della testa della lista vengono entrambi passati alla routine scrivendoli nei registri R2 e R4, mentre la routine restituisce il risultato lasciandolo nel registro R3. In Figura 2.17 le istruzioni importanti del programma chiamante sono quattro, mostrate esplicitamente. Le prime due caricano la lunghezza e NUM1 in R2 e R4; l'istruzione Call di chiamata a sottoprogramma passa a eseguire la routine all'indirizzo TOTALE impilando l'indirizzo di rientro; poi la routine calcola il risultato lasciandolo in R3 e infine la quarta istruzione lo memorizza da R3 nella parola di indirizzo SOMMA.

In aggiunta ai registri R2, R3 e R4, che sono usati per il passaggio dei parametri, il sottoprogramma usa anche R5. Poiché R5 può essere usato nel programma chiamante, il suo contenuto viene conservato impilandolo al momento dell'ingresso nel sottoprogramma e ripristinato prima del rientro al programma chiamante.

Se i parametri sono numerosi o di dimensione grande lo spazio nei registri di uso generale è troppo poco per passarli tutti quanti alla routine. Invece la pila è illimitata (almeno in linea di principio) e si presta bene a risolvere il problema. Ecco un esempio: la Figura 2.18 mostra il programma di Figura 2.8 riscritto come sottoprogramma, di nome TOTALE, che calcola la somma della lista di numeri, e un frammento di programma chiamante dove si effettua la chiamata con passaggio di parametri nella pila. La routine ha due parametri: l'indirizzo NUM1 di testa della lista (primo parametro a essere posto sulla pila dal chiamante) e la lunghezza n della lista (secondo parametro); essa calcola la somma di tutti gli elementi della lista e la restituisce come valore in uscita.

Per semplicità, si sfrutta il fatto che in questo esempio non occorre conservare nella pila l'indirizzo di rientro, cioè si trascura l'annidamento di sottoprogrammi; di simile semplificazione si farà a meno nel Paragrafo 2.7.3. Con tale premessa, si supponga che nel contesto del programma chiamante la cima della pila si trovi nella posizione indicata come livello 1 in Figura 2.19. Prima di chiamare la routine il programma chiamante impila i due parametri NUM1 e n, nell'ordine indicato; poi esso chiama la routine TOTALE, pertanto al momento dell'avvio della routine la cima della pila si trova nella posizione indicata come livello 2 in Figura 2.19.

Per i propri calcoli la routine si serve di quattro registri di uso generale. Siccome può darsi che contengano dati di pertinenza del programma chiamante, che esso si aspetta di ri-

Assumere che la cima della pila sia a livello 1 in Figura 2.19

	Move	R2, #NUM1	Impila i parametri
	Subtract	SP, SP, #4	
	Store	R2, (SP)	
	Load	R2, N	
	Subtract	SP, SP, #4	
	Store	R2, (SP)	
	Call	TOTALE	Chiama il sottoprogramma (cima della pila a livello 2)
	Load	R2, 4(SP)	Spila il risultato e conservalo in SOMMA
	Store	R2, SOMMA	
	Add	SP, SP, #8	Ripristina la cima della pila (cima della pila a livello 1)
	⋮		
TOTALE :	Subtract	SP, SP, #16	Salva in pila i registri
	Store	R2, 12(SP)	
	Store	R3, 8(SP)	
	Store	R4, 4(SP)	
	Store	R5, (SP)	(cima della pila a livello 3)
	Load	R2, 16(SP)	Inizializza il contatore a n
	Load	R4, 20(SP)	Inizializza il puntatore alla lista
	Clear	R3	Inizializza la somma a 0
CICLO:	Load	R5, (R4)	Preleva il prossimo numero
	Add	R3, R3, R5	Aggiungi questo numero alla somma
	Add	R4, R4, #4	Incrementa di 4 il puntatore
	Subtract	R2, R2, #1	Decrementa il contatore
	Branch_if_[R2]>0	CICLO	
	Store	R3, 20(SP)	Impila il risultato
	Load	R5, (SP)	Ripristina i registri
	Load	R4, 4(SP)	
	Load	R3, 8(SP)	
	Load	R2, 12(SP)	
	Add	SP, SP, #16	(cima della pila a livello 2)
	Return		Rientra al programma chiamante

Figura 2.18 Routine che calcola la somma della lista di numeri e meccanismo di passaggio di parametri tramite pila.

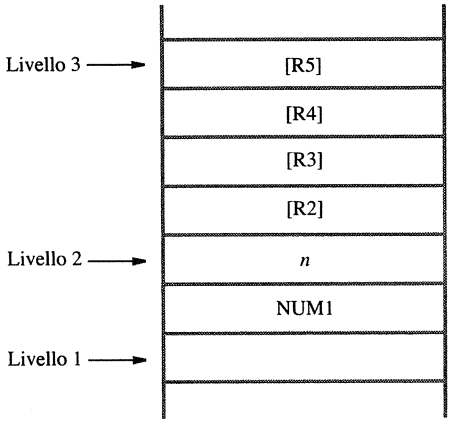


Figura 2.19 Contenuto e cima della pila in vari momenti di esecuzione.

trovare intatti dopo la terminazione della routine, i rispettivi contenuti vanno impilati e conservati lì fino al momento di recuperarli. In tale modo la cima della pila si porta nella posizione indicata come livello 3 in Figura 2.19.

La routine continua lavorando sui parametri *n* e NUM1 contenuti nella pila, tramite il modo di indirizzamento con indice. Si noti come la cima della pila non venga modificata; la routine lavora *all'interno della pila*, come se fosse memoria normale. Basta sapere di quante posizioni si deve guardare dentro la pila per trovare gli elementi che interessano. I parametri *n* e NUM1 vengono caricati in R2 e R4, e servono come contatore di lunghezza della lista e come puntatore scorrevole per scandire la lista, rispettivamente. La routine calcola la somma parziale e, al termine del ciclo di addizione, in R3 si trova la somma finale. Essa viene poi memorizzata dentro la pila, sovrascrivendo il parametro NUM1 che a questo punto non serve più.

Essendo terminato il calcolo la routine recupera i valori originali dei quattro registri, spilandoli e ricaricandoli nel processore (si dice anche che i registri *salvati* in pila vengono *ripristinati* nel processore). Inoltre, il puntatore alla pila è incrementato per puntare alla cima della pila che era presente quando il sottoprogramma era stato chiamato, ovvero alla locazione del parametro *n* a livello 2. Dopo il rientro dal sottoprogramma, il programma chiamante memorizza il risultato nella locazione SOMMA e abbassa la cima della pila al suo livello originale incrementando SP di 8.

Si osservi che per il sottoprogramma TOTALE in Figura 2.18, non è stata usata una coppia di istruzioni:

Subtract SP, SP, #4

Store R*j*, (SP)

per salvare in pila ciascun registro. Poiché dobbiamo salvare quattro registri, questo richiederebbe otto istruzioni. Tuttavia, sono state necessarie solo cinque istruzioni, poiché SP è stato modificato una sola volta per puntare alla cima della pila, che sarà raggiunta quando tutti e quattro i registri saranno salvati in pila. Quindi è stato usato il modo con indice e spiazzamento per memorizzare i contenuti dei registri. È stata usata la stessa ottimizzazione per ripristinare i registri prima di effettuare il rientro dal sottoprogramma.

Si dovrebbe inoltre notare che alcuni calcolatori hanno istruzioni speciali per caricare e memorizzare più registri. Per esempio, i quattro registri in Figura 2.18 possono essere salvati in pila usando l'istruzione:

StoreMultiple R2-R5, −(SP)

I registri sorgente sono specificati dall'intervallo R2-R5. La notazione −(SP) specifica che il puntatore alla pila deve essere aggiustato concordemente. Il segno meno prefisso indica che SP deve essere decrementato (di 4) prima che il contenuto di ciascun registro sia impilato.

In modo simile, l'istruzione:

LoadMultiple R2-R5, (SP)+

caricherà i registri R2, R3, R4 e R5, in ordine inverso, con i valori che erano stati conservati sulla pila. La notazione (SP)+ indica che il puntatore alla pila deve essere incrementato (di 4) dopo che ogni valore sia stato caricato nel corrispondente registro. Si discuteranno i modi di indirizzamento indicati da −(SP) e (SP)+ con maggiori dettagli nel Paragrafo 2.9.1.

Passaggio di parametri per valore o riferimento

È istruttivo fermarsi un istante per riflettere sulla natura dei parametri passati alla routine vista sopra, come mostrano le Figure 2.17 e 2.18. La funzione calcolata dalla routine è la somma della lista di numeri e ha due parametri: lista e lunghezza. Il primo parametro, NUM1, non è il contenuto dell'intera lista, bensì un *riferimento* alla lista, cioè l'indirizzo del primo elemento o, se si vuole, il puntatore al primo elemento. Invece il secondo parametro, *n*, è effettivamente il valore della lunghezza della lista. Pertanto si dice che il primo parametro, che in senso logico dovrebbe essere l'intera lista, è passato *per riferimento*, mentre il secondo è passato *per valore*.

2.7.3 Area di attivazione

Descrivendo le attività della routine mostrata nella Figura 2.18, che calcola la somma degli elementi della lista, si è visto come essa lavori usando le parole di memoria appartenenti al blocco sommitale della pila, e non solo con l'unica parola esattamente in cima. Tale blocco costituisce per la routine una sorta di spazio di lavoro privato, creato all'attivazione della routine ed eliminato quando essa termina. Il blocco di parole in questione costituisce l'*area di attivazione in pila* (*stack frame* o anche *activation record*) della routine. Contiene i parametri della routine e, se le occorrono, anche variabili locali, nonché spazio per conservare registri e altri elementi di informazione da mettere temporaneamente da parte.

Conviene tenere le variabili locali nell'area di attivazione perché sono usate solo dal sottoprogramma, pertanto possono stare in una zona di memoria che esiste solo per la durata della vita del sottoprogramma stesso. L'area di attivazione ha una struttura ben definita, fatta a strati posti l'uno sopra l'altro, un po' come un *sandwich*; alcuni strati possono talvolta mancare, ma altri sono sempre presenti. La cima dell'area coincide con la cima della pila ed è sempre indicata dal registro puntatore SP. La Figura 2.20 mostra un esempio di schema usato comunemente per la disposizione delle informazioni in una pila. In aggiunta al puntatore alla pila SP, è utile avere un altro registro puntatore, chiamato *puntatore all'area di attivazione* o FP (*Frame Pointer*), per accedere facilmente ai parametri passati al sottoprogramma e alle variabili della memoria locale usate dal sottoprogramma. Nella figura, si assume che quattro parametri siano passati al sottoprogramma, tre variabili locali siano usate dentro il sottoprogramma e che i registri R2, R3 e R4 debbano essere salvati in pila, poiché essi saranno usati dal sottoprogramma. Quando il meccanismo di chiamata a sottoprogramma si serve della pila in modo implicito, o altrimenti quando si usano sottoprogrammi annidati, l'area di attivazione del sottoprogramma chiamante conterrà l'indirizzo di rientro, come si può vedere nell'esempio seguente.

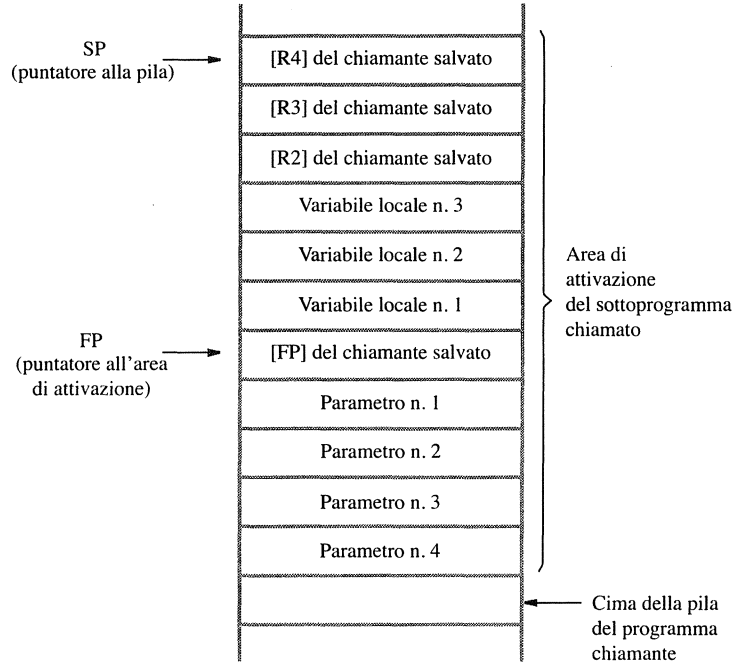


Figura 2.20 Esempio di area di attivazione di sottoprogramma.

Quando la routine è in esecuzione, il registro puntatore FP del processore fa riferimento alla posizione interna, nell'area di attivazione della routine, in cui si trova il *valore del registro* FP *del chiamante*, proprio come mostra la Figura 2.20. Pertanto si possono facilmente indirizzare i parametri mediante il modo con indice tramite spiazzamento positivo, multiplo della dimensione in byte del parametro: 4(FP), 8(FP), ..., per i parametri numero 1, 2, ..., rispettivamente (parametri con dimensione da 4 byte ciascuno). In modo analogo si possono indirizzare le variabili locali, ma con spiazzamento negativo: −4(FP), −8(FP), ..., per le variabili locali numero 1, 2, ..., rispettivamente (variabili con dimensione da 4 byte ciascuna). Il registro FP ha contenuto costante per tutta la durata dell'esecuzione della routine, mentre il contenuto del registro SP può variare, giacché punta sempre alla cima della pila e la routine potrebbe impilare o spilare elementi, in particolare se deve calcolare espressioni.

Ora bisogna vedere come si manipolano i registri SP e FP durante le varie fasi in cui l'area di attivazione viene prima costruita, poi è in uso e viene infine smantellata; si segue l'esempio di Figura 2.20. All'inizio la routine non è stata ancora chiamata e SP punta alla cima della pila del programma chiamante. Subito prima di chiamare la routine il programma chiamante impila il blocco di 4 parametri, nell'ordine voluto. Adesso, SP punta all'ultimo parametro che è stato impilato. È il momento di aggiornare il registro FP, dandogli il valore giusto per fare riferimento all'area di attivazione della routine. Tuttavia, FP è stato finora in uso da parte del programma chiamante (che deve avere anch'esso una propria area di attivazione da qualche parte; si vedrà più avanti dove), pertanto il contenuto corrente di FP non deve andare perso, dunque viene salvato impilandolo e di conseguenza anche SP viene aggiornato. Subito dopo il contenuto di SP viene copiato in FP, che da ora in avanti punta dentro l'area di attivazione della routine proprio alla posizione dove si conserva il valore di FP valido per il programma chiamante. Si può dire che proprio in questo istante l'area di attivazione della routine entri effettivamente in uso (benché la costruzione non sia ancora completa), cioè quando il registro FP, su cui si incardina l'accesso ai contenuti dell'area, viene aggiornato in modo da riferirsi all'area. Pertanto le tre istruzioni iniziali della routine sono le seguenti:


```
Subtract   SP, SP, #4
Store      FP, (SP)
Move       FP, SP
```

L'istruzione Move copia il contenuto di SP in FP. Dopo che queste istruzioni saranno state eseguite, SP e FP punteranno entrambi al contenuto di FP salvato in pila. Lo spazio per le tre variabili locali viene adesso allocato sulla pila eseguendo l'istruzione:

```
Subtract   SP, SP, #12
```

Infine, si salvano in pila i registri R2, R3 e R4. A questo punto, l'area di attivazione è stata inizializzata come mostrato in Figura 2.20.

Il sottoprogramma adesso esegue il suo compito. Quando tale compito viene completato, il sottoprogramma ripristina i registri R4, R3 e R2, dealloca le variabili locali dall'area di attivazione eseguendo l'istruzione:

```
Add   SP, SP, #12
```

e ripristina il registro FP. Adesso SP punta all'ultimo parametro che era stato impilato. Si esegue quindi l'istruzione Return, trasferendo così il controllo al programma chiamante.

Il programma chiamante ha la responsabilità di deallocare i parametri dall'area di attivazione, alcuni dei quali possono fornire risultati provenienti dal sottoprogramma. Dopo la deallocazione dei parametri, il puntatore alla pila punta alla vecchia cima della pila; si è così ritornati al punto di partenza.

Area di attivazione con annidamento di sottoprogramma

Si è già visto che la pila funziona egregiamente come struttura di conservazione degli indirizzi di rientro anche quando i sottoprogrammi sono annidati. Dovrebbe ormai essere chiaro come le aree di attivazione di sottoprogrammi annidati siano tutte contenute nella pila e installate una sopra l'altra. In particolare i vari valori di FP conservati nelle aree di attivazione impilate sono *concatenati* in serie (in modo molto simile a una lista concatenata), cioè quello sopra punta a quello sotto e così via a scendere.

La Figura 2.21 mostra una situazione di annidamento: il programma principale, PRO, chiama la routine SUB1, che a sua volta chiama la routine SUB2. La Figura 2.22 mostra la parte sommitale della pila, dove si trovano impilate le aree di attivazione di SUB1 (sotto) e SUB2 (sopra); l'area di attivazione di PRO sta sotto quella di SUB1, ma non è mostrata. I parametri sono passati in pila, che è mostrata nel momento di espansione massima. Si constati come gli elementi che conservano i valori di FP del chiamante siano concatenati in modo simile agli elementi di una lista con puntatori.

Il flusso di esecuzione procede nel modo seguente. Il programma principale PRO impila i due parametri numero 1 e 2, e chiama la routine SUB1. Essa effettua un certo calcolo (qui non precisato) e produce un risultato che restituisce a PRO tramite la pila. Nel corso della propria attività SUB1 chiama un'altra routine, SUB2, per affidarle un qualche compito ausiliario. La routine SUB1 passa in pila a SUB2 il parametro numero 3 e le viene restituito

Locazione di memoria		Istruzioni		Commenti
Programma principale				
		⋮		
2000	PRO:	Load	R2, PARAM2	Impila i parametri
2004		Subtract	SP, SP, #4	
2008		Store	R2, (SP)	
2012		Load	R2, PARAM1	
2016		Subtract	SP, SP, #4	
2020		Store	R2, (SP)	
2024		Call	SUB1	Chiama il sottoprogramma
2028		Load	R2, (SP)	Immagazzina il risultato
2032		Store	R2, RIS	
2036		Add	SP, SP, #8	Ripristina il livello della pila
2040		prossima istruzione		
		⋮		
Primo sottoprogramma				
2100	SUB1:	Subtract	SP, SP, #24	Salva in pila i registri
2104		Store	LINK_reg,20(SP)	
2108		Store	FP, 16(SP)	
2112		Store	R2, 12(SP)	
2116		Store	R3, 8(SP)	
2120		Store	R4, 4(SP)	
2124		Store	R5, (SP)	
2128		Add	FP, SP, #16	Inizializza il puntatore all'area di attivazione
2132		Load	R2, 8(FP)	Preleva il primo parametro
2136		Load	R3, 12(FP)	Preleva il secondo parametro
		⋮		
		Load	R4, PARAM3	Impila un parametro da passare a SUB2
		Subtract	SP, SP, #4	
		Store	R4, (SP)	
		Call	SUB2	
		Load	R4, (SP)	Spila il risultato ottenuto da SUB2
		Add	SP, SP, #4	
		⋮		
		Store	R5, 8(FP)	Impila la risposta
		Load	R5, (SP)	Ripristina i registri
		Load	R4, 4(SP)	
		Load	R3, 8(SP)	
		Load	R2, 12(SP)	
		Load	FP, 16(SP)	
		Load	LINK_reg,20(SP)	
		Add	SP, SP, #24	
		Return		Rientro al programma principale

Continua nella parte b

Figura 2.21a Programma principale PRO con due sottoprogrammi SUB1 e SUB2 di primo e secondo livello.

Locazione di memoria		Istruzioni		Commenti
Secondo sottoprogramma				
3000	SUB2:	Subtract	SP, SP, #12	Salva in pila i registri
3004		Store	FP, 8(SP)	
		Store	R2, 4(SP)	
		Store	R3, (SP)	
		Add	FP, SP, #8	Inizializza il puntatore all'area di attivazione
		Load	R2, 4(FP)	Preleva il parametro
		⋮		
		Store	R3, 4(FP)	Impila il risultato di SUB2
		Load	R3, (SP)	Ripristina i registri
		Load	R2, 4(SP)	
		Load	FP, 8(SP)	
		Add	SP, SP, #12	
		Return		Rientro al sottoprogramma 1

Figura 2.21b Sottoprogramma SUB2.

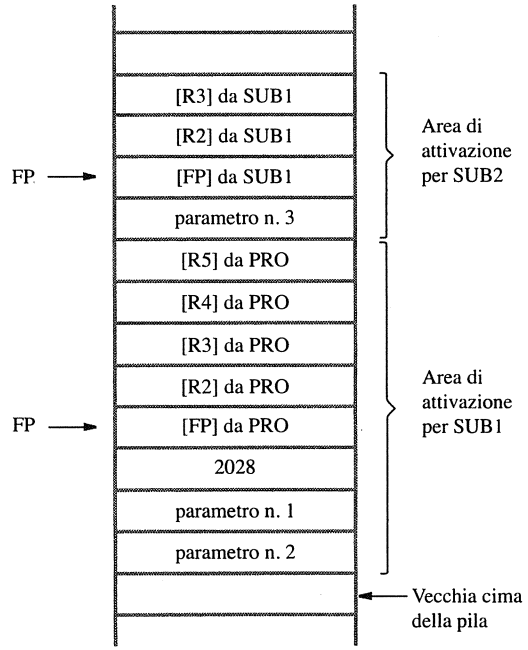


Figura 2.22 Area di attivazione di sottoprogramma.

da parte di SUB2 un risultato, sempre tramite la pila. Terminata SUB2, SUB1 ne carica il risultato nel registro R4 e continua il proprio lavoro producendo un risultato che restituisce a PRO, di nuovo tramite la pila. Terminata SUB1, PRO ne copia il risultato nella parola di memoria di indirizzo RIS e ripristina il livello della pila, quindi continua con le sue elaborazioni alla prossima istruzione di indirizzo 2040. Si noti come l'indirizzo di rientro al programma chiamante, 2028, sia memorizzato dentro l'area di attivazione per SUB1 in Figura 2.22.

La Figura 2.22 mostra il contenuto della pila, come si viene determinando con l'avanzare dell'esecuzione fino al momento di espansione massima. Come primo compito, se il meccanismo di chiamata a sottoprogramma usa il registro di collegamento invece che la pila in modo implicito, ciascuna routine che sia chiamante di qualche altra routine annidata salva in pila il registro di collegamento; nell'esempio ciò avviene nell'area di attivazione di SUB1, essendo questa chiamante di SUB2. Ciascuna routine, quindi, impila il puntatore FP e lo aggiorna; subito dopo impila i contenuti dei registri da liberare. La routine SUB1 ne usa quattro: R2, R3, R4 e R5; SUB2 ne usa solo due: R2 e R3. Registri e puntatore FP vengono ripristinati subito prima del termine della routine e del rientro al chiamante.

Per lavorare sul contenuto dell'area di attivazione ciascuna routine si serve del modo di indirizzamento con indice, sfruttando il puntatore FP come registro indice, appunto: caricare i parametri nel processore e memorizzare il risultato in pila (né SUB1 né SUB2 hanno variabili locali, dunque non c'è spazio libero in pila per loro). Per il parametro lo spiazzamento è positivo: 4, 8, ... (come si è già visto prima esaminando la Figura 2.20), se il registro di collegamento non è salvato in pila né la chiamata di sottoprogramma usa la pila in modo implicito, altrimenti 8, 12 ecc. Infine, si noti come il compito di eliminare il blocco di parametri impilati tocchi al chiamante. Ciò viene fatto dall'istruzione di addizione, sia nel programma principale sia nel sottoprogramma SUB1 (chiamante di SUB2), che abbassa la cima della pila.

2.8 Altre istruzioni macchina

Finora sono state introdotte le istruzioni macchina generiche seguenti: Load, Store, Move, Clear, Add, Subtract, Branch, Call e Return.

Queste nove istruzioni e i modi di indirizzamento elencati in Tabella 2.1 bastano per scrivere programmi capaci di svolgere una notevole varietà di compiti, puramente sequenziali, contenenti cicli o anche sottoprogrammi. Nel presente paragrafo si introdurranno alcune nuove istruzioni indispensabili o importanti, che si trovano nella maggior parte degli insiemi di istruzioni di interesse pratico commerciale.

2.8.1 Istruzioni logiche

È ben noto che le operazioni logiche elementari, che lavorano su bit come il prodotto logico o AND, la somma logica o OR e la negazione o NOT, costituiscono gli elementi base per la costruzione di circuiti digitali, come si spiega in Appendice A. D'altro canto si sa anche che le stesse operazioni hanno ruolo fondamentale nei linguaggi di programmazione, dove servono per definire condizioni (di ciclo, ecc). In linguaggio macchina, le operazioni logiche si applicano a parole lavorando in parallelo su tutti i bit della parola, o come si dice agendo in modo *bit a bit* (*bitwise*). Per esempio, l'istruzione macchina seguente:

And R4, R2, R3

Approfondimento applicativo 2.4

Collegamento di sottoprogrammi, macroistruzioni

Un sommario delle principali caratteristiche del collegamento di sottoprogrammi nei quattro processori considerati è fornito in Tabella A2.5. Si può notare la preferenza per l'uso del registro di collegamento in quelli di stile RISC, al contrario dei processori CISC, che usano la pila in modo implicito per l'indirizzo di rientro. I registri di interesse al meccanismo di collegamento hanno nomi diversi nei quattro processori; si noti che nel processore ColdFire, a differenza degli altri, si può usare qualsiasi registro (di indirizzo) per il puntatore all'area di attivazione, tranne A7, riservato al puntatore alla cima della pila.

I processori NIOS II e ColdFire hanno due diverse istruzioni per la chiamata di sottoprogramma, che in sostanza differiscono, in ciascuno dei due, per il modo di indirizzamento nel salto a quest'ultimo. Nel primo modo, il programmatore fornisce l'etichetta dell'istruzione di inizio dell'esecuzione del sottoprogramma, e il modo effettivamente impiegato è quello delle istruzioni di salto. Questo è anche il (solo) modo impiegato nelle istruzioni di chiamata ARM e IA-32. L'altra istruzione di chiamata NIOS II prevede l'indirizzo di salto in un registro, mentre nell'altra ColdFire si possono usare vari modi di indirizzamento, indicati in tabella.

A differenza degli altri tre, l'ISA ARM non ha una specifica istruzione per il rientro, ma usa un'istruzione di caricamento multiplo di registri, che generalizza quella dell'istruzione generica LoadMultiple (si veda il Paragrafo 2.7.2) e che ha anche altre forme e usi. In questo caso si ha l'indirizzo sorgente nel registro SP e una *lista* di registri di destinazione. Il registro SP è post-incrementato di 4 a ogni caricamento di registro, poiché il suffisso FD (*full descending*) del codice mnemonico indica che la cima della pila è la sua ultima locazione *occupata*

Tabella A2.5 Collegamento di sottoprogrammi NIOS II, ColdFire, ARM, IA-32

Caratteristica	NIOS II	ColdFire	ARM	IA-32
Meccanismo di collegamento	Registro di collegamento	In pila, implicito	Registro di collegamento	In pila, implicito
Registri:				
PC		PC	R15|PC	EIP
SP	r27|sp	A7	R13|SP	ESP
FP	r28|fp	A*j*	R12|FP	EBP
LR	r31|ra		R14|LR	
Istruzioni di chiamata e di rientro	call *l*	BSR *l*	BL *l*	CALL *l*
	callr *ri*	JSR *a*		
	ret	RTS	LDMFD SP!, {*r*,PC}	RET
Altre istruzioni rilevanti		MOVEM.L *r*, (A7)	STMFD SP!, {*r*,LR}	PUSH *s*
				POP *d*
		LINK A*j*, #*p*		PUSHAD
		UNLK A*j*		POPAD

Legenda:
A*j* registro di indirizzo usato come puntatore all'area di attivazione ($0 \leq j \leq 6$);
l etichetta;
a indirizzo del sottoprogramma, modi: assoluto, indiretto da registro, con base, indice e spiazzamento, relativo a PC (con indice);
r lista di registri;
p spiazzamento;
s operando sorgente da impilare;
d destinazione di operando da spilare.

e che la pila *decresce* verso indirizzi crescenti. Si effettua così il ripristino dei registri salvati in pila alla chiamata, l'ultimo dei quali è il PC, il che fa trasferire il controllo all'istruzione successiva a quella di chiamata. Ciò è correttamente realizzato dal fatto che l'esecuzione dell'istruzione di chiamata BL (*Branch and Link*) carica il registro di collegamento con il contenuto del PC (ovvero l'indirizzo dell'istruzione successiva a quella di chiamata), e che la prima istruzione del sottoprogramma è quella di salvataggio in pila di un insieme di registri indicata in tabella, con successivi pre-decrementi di 4 del registro SP e con il registro di collegamento per primo a essere salvato. Si noti che l'ordine di salvataggio e ripristino dei registri è determinato dal fatto che a registri di nomi crescenti corrispondono indirizzi di memoria crescenti.

Anche il processore ColdFire ha un'istruzione di salvataggio multiplo di registri simile a quella ARM, ma non ha un registro di collegamento (poiché usa la pila in modo implicito), mentre ha altre istruzioni specifiche, LINK e UNLK, rispettivamente per l'allocazione e la deallocazione della memoria in pila che ospita l'area di attivazione del sottoprogramma. Lo spiazzamento *p* è un numero negativo il cui valore assoluto indica la dimensione (in byte) della parte dell'area di attivazione riservata alle variabili locali e al salvataggio di altri registri oltre al registro FP, salvato nella locazione in cima ai parametri passati dal chiamante e che subito dopo il salvataggio viene aggiornato con l'indirizzo di tale locazione (si veda il Paragrafo 2.7.3). Il salvataggio del registro FP, il suo aggiornamento e quello del registro SP (incluso il decremento specificato dallo spiazzamento *p*) sono effettuati dalla singola istruzione LINK, mentre l'istruzione UNLK effettua le operazioni inverse.

Le istruzioni IA-32 di salvataggio in pila e ripristino multiplo di registri, rispettivamente PUSHAD e POPAD, sono più limitate di quelle disponibili nelle ISA ColdFire e ARM, poiché l'insieme dei registri è prefissato: tutti gli otto registri di uso generale. Quando tanto risulta troppo si ricorre alle istruzioni PUSH e POP, per impilare e spilare un singolo operando.

Come accennato alla fine dell'Approfondimento applicativo 2.3, direttive per definizioni di macroistruzioni, o *macro* per brevità, sono disponibili nei *macroassemblatori*. Tutti gli assemblatori qui considerati le offrono, sebbene con differenze di forma e funzionalità. Le definizioni di macro e le loro invocazioni hanno forme sintattiche simili a quelle delle definizioni e invocazioni di sottoprogrammi, procedure e funzioni nei linguaggi di alto livello, ma una semantica affatto diversa. La differenza di base sta nel fatto che l'invocazione di una macro non si traduce nel salto a un sottoprogramma, ma è eseguita dall'assemblatore come *espansione della macro* stessa, ovvero sostituzione testuale, nel codice simbolico, dell'invocazione con il corpo della definizione. Se una definizione di macro ha parametri, la sua invocazione può specificarne la sostituzione con corrispondenti stringhe di testo nell'espansione del corpo.

La sintassi delle principali direttive per definizioni di macro negli assemblatori GAS, ColdFire, ARM e MASM è schematizzata in Tabella A2.6. Il nome della macro è un argomento

Tabella A2.6 Forma delle direttive per definizioni di macro negli assemblatori GAS, ColdFire, ARM e MASM

GAS	ColdFire	ARM	MASM
.macro *m p*	*m* .macro *p*	MACRO *l m p*	*m* MACRO
.exitm	.mexit	MEXIT	EXITM *v*
.endm	.endm	MEND	ENDM

Legenda:
m nome della macro;
p lista (opzionale) di nomi di parametri, separati da "," con opzionali dichiarazioni (valori di default ecc.) in GAS, ARM e MASM;
l parametro etichetta (opzionale, etichetta interna);
v valore di ritorno (opzionale).

della direttiva di definizione negli assemblatori GAS e ARM, mentre ne è l'etichetta negli altri due. Le direttive in ultima riga marcano la fine del corpo della definizione, quelle in penultima arrestano l'espansione. Le macro MASM possono restituire un valore di ritorno, nel qual caso possono essere invocate come funzioni in espressioni e l'espansione si conclude rimpiazzandone l'invocazione con il valore di ritorno.

calcola l'AND bit a bit degli operandi nei registri R2 e R3, e lascia il risultato in R4. Una forma diretta di questa istruzione può essere:

And R4, R2, #Valore

dove Valore è un valore logico a 16 bit che è esteso a 32 bit mettendo a zero i 16 bit delle posizioni più significative.

Eccone un esempio di uso: si supponga che il registro R2 (da 32 bit) contenga quattro caratteri ASCII (da 8 bit ciascuno) e che si debba stabilire se il carattere più a destra dei quattro sia la lettera 'Z' (maiuscola). Dalla Tabella 1.1 si vede che il codice ASCII a 8 bit[3] di 'Z' è 01011010 (in binario), che in esadecimale si rappresenta con 5A (si veda il Paragrafo 2.5.3). La sequenza di tre istruzioni mostrata di seguito svolge il compito richiesto:

```
And                 R2, R2, #0xFF
Move                R3, #0x5A
Branch_if_[R2]=[R3] TROVATOZ
```

L'istruzione di prodotto logico And azzera i tre caratteri a sinistra nel registro R2 lasciando inalterato quello a destra. Il motivo è che il primo argomento dell'istruzione, in modo immediato, è una costante i cui 24 bit più a sinistra valgono 0 e gli 8 bit più a destra valgono 1. L'istruzione Move carica il valore esadecimale 5A dentro R3. Poiché entrambi R2 e R3 hanno a zero i 24 bit più a sinistra, l'istruzione Branch confronta il carattere rimanente nella parte destra di R2 con la rappresentazione binaria del carattere 'Z' e provoca il salto verso TROVATOZ se c'è corrispondenza.

2.8.2 Scorrimento e rotazione

Spesso occorre fare scorrere verso destra o sinistra i bit dell'operando, per un numero specificato di posizioni. Come effettuare l'operazione di *scorrimento* (*shift*) dipende però dalla natura dell'operando, vale a dire se lo si voglia considerare come un numero con segno o qualche informazione più generale codificata in binario. Nel primo caso si conserva il segno dell'operando da fare scorrere.

Scorrimento logico

Vi sono due istruzioni di scorrimento logico differenti: verso sinistra LShiftL (*logic shift left*) e verso destra LShiftR (*logic shift right*). Queste istruzioni effettuano lo scorrimento (*shift*) di un operando per un numero di posizioni di bit specificato in un operando contatore contenuto nell'istruzione. La forma generale di un'istruzione di scorrimento logico verso sinistra è:

LShiftL R*i*, R*j*, contatore

[3] Il codice ASCII a 8 bit estende quello a 7 bit di Tabella 1.1 con un bit 0 a sinistra.

che fa scorrere il contenuto del registro R*j* a sinistra di un numero di posizioni di bit dato dall'operando contatore e pone il risultato nel registro R*i*, senza cambiare il contenuto di R*j*. L'operando contatore può essere fornito in modo immediato, o può essere contenuto in un registro del processore. Bisogna però precisare anche come riempire con nuovi bit l'operando dal lato destro e dove vadano i bit in uscita dal lato sinistro. A destra le posizioni liberate si riempiono con nuovi bit di valore zero, a sinistra i bit in uscita passano prima attraverso il bit di riporto (uno dei bit di esito, si veda il Paragrafo 2.10.2), poi vengono definitivamente scartati, tranne l'ultimo a uscire che resta disponibile nel bit di riporto (salvo perdersi anch'esso al prossimo scorrimento). Lasciare l'ultimo bit nel riporto è di aiuto per i calcoli aritmetici in precisione multipla, quando cioè il numero intero è rappresentato su due o più parole. La Figura 2.23a mostra come funziona l'istruzione di scorrimento logico verso sinistra: il

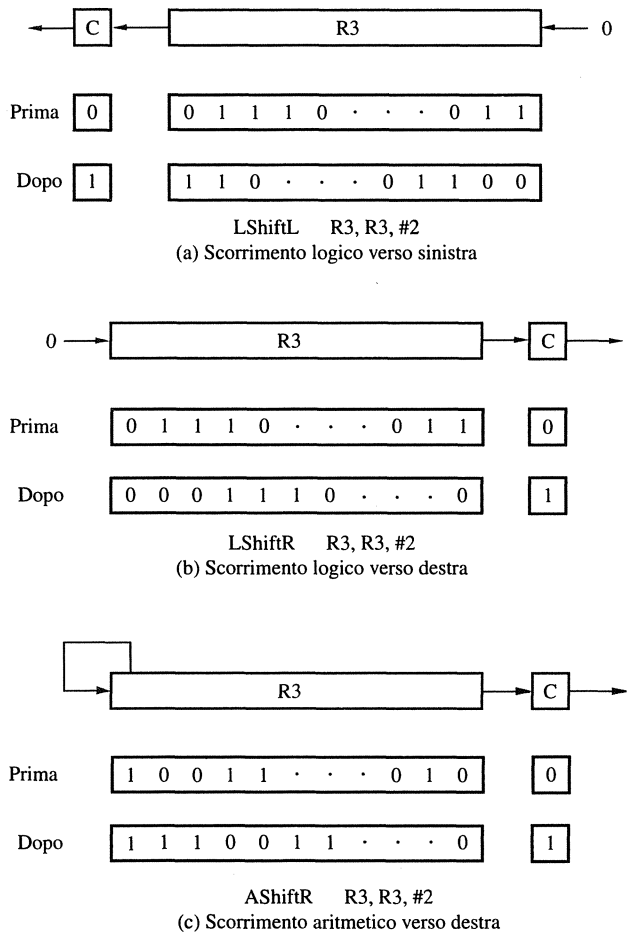


Figura 2.23 Istruzioni di scorrimento logico e aritmetico.

contenuto del registro R3 viene fatto scorrere di due bit coinvolgendo anche il riporto. La Figura 2.23b mostra lo scorrimento verso destra, che funziona in modo del tutto analogo, salvo il verso di scorrimento.

Esempio di impaccamento di cifre decimali

Si consideri il compito seguente, molto semplice, che illustra bene l'uso sia dello scorrimento sia delle operazioni logiche. Si supponga di avere due cifre decimali (dunque di valore compreso nell'intervallo [0, 9], contenute nei due byte di memoria di indirizzo DATO e DATO + 1, rispettivamente. La cifra decimale occupa solo i 4 bit meno significativi del byte in cui è contenuta; tale formato di rappresentazione si chiama *codice BCD* (*Binary Coded Decimal*, cifra decimale codificata in modo binario). Si vogliono prendere le due cifre e impaccarle concatenandole in un solo byte all'indirizzo di memoria COPPIA: il risultato è un modo efficiente per condensare molta informazione in pochi bit e si chiama *formato BCD impaccato* (*packed BCD format*). Dalla Tabella 1.1 si può constatare come i 4 bit più a destra del codice ASCII della cifra decimale corrispondano in effetti al codice BCD associato alla cifra. Dunque tutto si riduce a prendere i 4 bit meno significativi dei due byte di indirizzo DATO e DATO + 1 e concatenarli in un solo byte di indirizzo COPPIA.

Il programma mostrato in Figura 2.24 svolge il compito richiesto usando il registro R2 come puntatore ai due byte in memoria contenenti le due cifre decimali, mentre R3 e R4 come registri di lavoro dove elaborarle. Il programma utilizza l'istruzione LoadByte, che carica un byte dalla memoria e lo inserisce negli otto bit più a destra di un registro a 32 bit del processore e mette a zero i rimanenti bit di ordine più alto. L'istruzione StoreByte scrive il byte più a destra del registro sorgente nella locazione di destinazione specificata, ma non cambia i byte di altre locazioni. Il valore 0xF nell'istruzione And è usato per mettere a zero tutti i bit di R4 tranne i quattro bit più a destra. Si noti che l'operando sorgente immediato è scritto come 0xF, che, interpretato come una sequenza di 32 bit, ha 28 zeri nelle posizioni più significative.

Scorrimento aritmetico

Ripensando a quanto detto in precedenza, si capisce subito che facendo scorrere in modo logico di un bit verso sinistra un numero naturale lo si raddoppia, verso destra lo si dimezza; il tutto si estende in modo ovvio in caso di scorrimento logico di più bit. Dunque in pratica lo scorrimento logico equivale a moltiplicare o dividere il numero naturale per una potenza di due. Beninteso, scorrendo verso sinistra si potrebbe verificare trabocco, mentre verso destra i bit del resto della divisione vanno persi; ciò che resta è il quoziente.

Move	R2, #LOC	R2 punta alla parola contenente i due byte
LoadByte	R3, (R2)	Carica in R3 il primo byte
LShiftL	R3, R3, #4	Scorrimento logico a sinistra di 4 bit
Add	R2, R2, #1	Incrementa il puntatore
LoadByte	R4, (R2)	Carica in R4 il secondo byte
And	R4, R4, #0xF	Azzera i bit di ordine più alto
Or	R3, R3, R4	Concatena in R3 le due cifre BCD
StoreByte	R3, COPPIA	Immagazzina il risultato

Figura 2.24 Routine per compattare in un solo byte due cifre decimali a 4 bit.

Tutto questo si estende senza difficoltà anche al numero relativo rappresentato in complemento a due, tranne che scorrendo verso destra bisogna replicare il bit più significativo (e non introdurre sempre nuovi bit di valore 0). Lo scorrimento aritmetico verso destra si distingue da quello logico nello stesso verso proprio per tale modo di introdurre nuovi bit da sinistra; in tutto il resto le due operazioni si somigliano. La Figura 2.23c mostra lo scorrimento aritmetico verso destra (con riporto). Quanto allo scorrimento aritmetico verso sinistra, è identico a quello logico nello stesso verso e non c'è bisogno di istruzione distinta. Va da sé che verso sinistra non si può scorrere indefinitamente, pena fare traboccare il numero.

Rotazione

Nello scorrimento i bit in uscita vanno persi, tranne l'ultimo che resta disponibile nel bit di riporto (ma si perderà anch'esso al prossimo scorrimento). L'istruzione di rotazione somiglia allo scorrimento, ma preserva tutti i bit. Essa lavora nel modo ovvio: i bit in uscita da un estremo rientrano all'estremo opposto, cosicché non si perde niente. Come lo scorrimento, la rotazione può girare verso sinistra o destra e di solito sono disponibili entrambe le versioni. L'ultimo bit emesso resta comunque disponibile nel bit di riporto. Ma la rotazione può anche coinvolgere direttamente il bit di riporto, che si aggiunge allora come posizione ulteriore di rotazione. La Figura 2.25, divisa nei quattro casi (a), (b), (c) e (d), mostra l'istruzione di rotazione di due bit verso sinistra senza e con aggiunta di riporto, e di due bit verso destra senza e con aggiunta di riporto, rispettivamente. I codici mnemonici delle quattro forme dell'istruzione di rotazione sono del tutto ovvi, come anche il formato.

2.8.3 Moltiplicazione e divisione

Si possono moltiplicare due fattori dotati di segno, vale a dire rappresentati in complemento a due, mediante un'istruzione analoga a quella di addizione e con lo stesso formato.

L'istruzione:

Multiply R*k*, R*i*, R*j*

effettua l'operazione:

$$R_k \leftarrow [R_i] \times [R_j]$$

In generale occorrono $2n$ bit per rappresentare correttamente in complemento a due il prodotto di due numeri relativi a n bit, dunque può succedere che il prodotto non ci stia per intero nel registro R*k*. Nella maggior parte dei linguaggi macchina l'istruzione di moltiplicazione scrive nella destinazione solo gli n bit meno significativi del prodotto, mentre gli n bit più significativi vanno persi o, più esattamente, non vengono neppure calcolati. Nonostante ciò, tutto va bene se si è certi a priori che il prodotto sia abbastanza piccolo da non estendersi sui bit più significativi. Siccome non sempre è così sono previste varianti dell'istruzione che scrivono il prodotto in una coppia di registri di destinazione solitamente contigui, come a dire in R*k* e R_{k+1}, dove in R_{k+1} va la metà più significativa.

Benché sia meno frequente, in alcuni linguaggi macchina si trova anche l'istruzione di divisione intera (di numeri relativi in complemento a due).

Divide R*k*, R*i*, R*j*

che effettua l'operazione:

$$R_k \leftarrow [R_j]/[R_i]$$

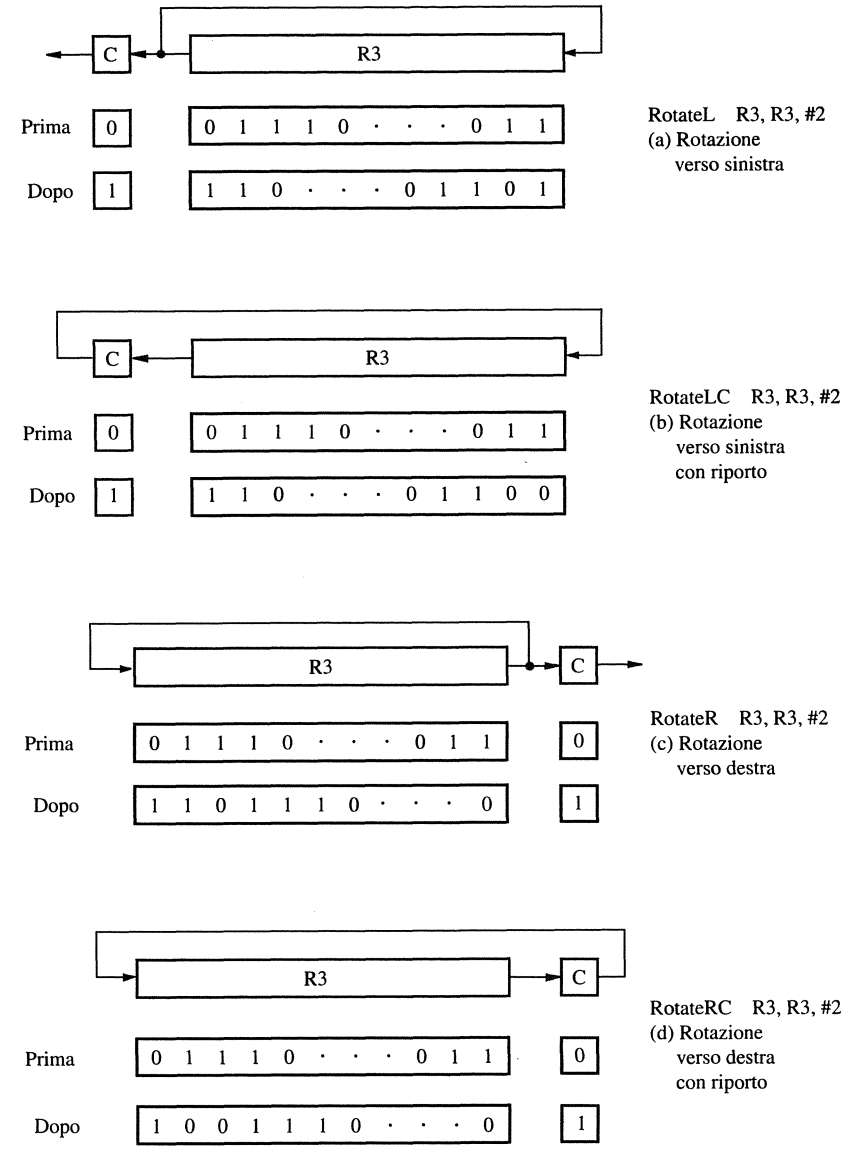


Figura 2.25 Istruzioni di rotazione.

Essa calcola il quoziente prendendo R*j* come dividendo e R*i* come divisore, scrivendolo in R*k*. Il resto della divisione o viene scritto in R_{k+1} o non viene neppure calcolato.

Non tutti i calcolatori dispongono di istruzioni di moltiplicazione e divisione, ma possono comunque calcolare queste e altre operazioni aritmetiche e algebriche, più complesse, ridu-

cendole a sequenze di istruzioni semplici come addizione, sottrazione, scorrimento e rotazione. Questo diventerà più evidente quando si descriverà la realizzazione di operazioni aritmetiche nel Capitolo 9.

2.9 Trattamento di valori immediati a 32 bit

Nella discussione dei modi di indirizzamento, nel Paragrafo 2.4.1, è stata sollevata la questione di come un valore a 32 bit che rappresenta una costante o un indirizzo di memoria possa essere caricato in un registro del processore. I modi immediato e assoluto di un processore in stile RISC limitano a 16 bit lo spazio disponibile, nel formato binario dell'istruzione, per l'operando o per il suo indirizzo, rispettivamente. Dunque un valore a 32 bit non può essere fornito esplicitamente in una singola istruzione che deve occupare una parola da 32 bit.

Una possibile soluzione è usare due istruzioni per questo scopo. Un approccio utilizzato in processori RISC si serve di istruzioni che effettuano due operazioni di OR logico differenti. L'istruzione:

Or Rdst, Rsrc, #Valore

estende l'operando immediato a 16 bit mettendo a zero i bit di ordine più alto per formare un valore a 32 bit, su cui è poi effettuato l'OR con il contenuto del registro Rsrc. Se Rsrc contiene zero allora Rdst sarà caricato con il valore esteso a 32 bit. Un'altra istruzione:

OrHigh Rdst, Rsrc, #Valore

forma un valore a 32 bit prendendo i 16 bit dell'operando immediato per costituire i bit di ordine più alto e mettendo a zero i bit di ordine più basso. Su questo valore è quindi effettuato l'OR con il contenuto di Rsrc. Usando queste istruzioni e assumendo che R0 contenga il valore 0, possiamo caricare il valore a 32 bit 0x20004FF0 nel registro R2 come segue:

```
OrHigh   R2, R0, #0x2000
Or       R2, R2, #0x4FF0
```

Per facilitare la stesura dei programmi, un insieme di istruzioni in stile RISC può includere pseudoistruzioni che indicano un'azione che richiede più di un'istruzione macchina. Tali pseudoistruzioni sono sostituite con le sequenze di istruzioni macchina corrispondenti dal programma assemblatore. Per esempio, la pseudoistruzione:

MoveImmediateAddress R2, LOC

potrebbe essere usata per caricare un indirizzo a 32 bit rappresentato dal simbolo LOC nel registro R2. Nel programma assemblato, essa sarebbe sostituita da due istruzioni che usano valori a 16 bit come mostrato sopra.

Un'alternativa all'uso di due istruzioni per caricare un indirizzo di 32 bit in un registro è usare più di una parola per istruzione. In questo caso, un'istruzione a due parole potrebbe specificare il codice operativo e il registro nella prima parola, mentre includere un valore a 32 bit nella seconda parola. Questo è l'approccio seguito nei processori in stile CISC.

Infine, si noti che nei precedenti paragrafi si è sempre assunto che singole istruzioni Load e Store possano essere usate per accedere a locazioni di memoria rappresentate da nomi simbolici. Questo rende più semplici e facili da leggere i programmi di esempio. I programmi eseguiranno correttamente se gli indirizzi di memoria richiesti potranno essere specificati in

16 bit. Se sono necessari indirizzi più lunghi, allora dovrà essere usato il suddetto approccio di costruzione di indirizzi a 32 bit.

2.10 Insiemi di istruzioni CISC

Nei paragrafi precedenti abbiamo introdotto lo stile RISC per gli insiemi di istruzioni. Adesso esamineremo alcune caratteristiche importanti dei *Calcolatori con Insieme di Istruzioni Complesso* o CISC (*complex instruction set computer*).

Una differenza essenziale è che gli insiemi di istruzioni CISC non sono vincolati all'*architettura carica/immagazzina*, in cui operazioni aritmetiche e logiche possono essere effettuate solo su operandi che si trovano nei registri dei processori. Un'altra differenza essenziale è che le istruzioni non necessariamente devono occupare una singola parola. Alcune istruzioni possono occupare una singola parola, ma altre possono estendersi su più parole.

Tipicamente le istruzioni nei processori CISC moderni non usano un formato a tre indirizzi. La maggior parte delle istruzioni aritmetiche e logiche usano il formato a due indirizzi:

Operazione destinazione, sorgente

Un'istruzione Add di questo tipo è:

Add B, A

che effettua l'operazione B ← [A] + [B] su operandi in memoria. Quando viene calcolata la somma, il risultato è inviato in memoria e immagazzinato nella locazione B, sostituendo il contenuto originario di questa locazione. Questo significa che la locazione di memoria B è sia una sorgente sia una destinazione.

Si consideri di nuovo il compito di sommare due numeri:

$$C = A + B$$

dove tutti i tre operandi possono risiedere in locazioni di memoria. Ovviamente, questo non può essere fatto con una sola operazione a due indirizzi. Il compito può essere effettuato usando un'ulteriore istruzione a due indirizzi che copia il contenuto di una locazione di memoria in un'altra. Questa istruzione è:

Move C, B

che effettua l'operazione C ← [B], lasciando il contenuto della locazione B immutato. L'operazione C ← [A] + [B] può essere adesso effettuata dalla sequenza di due istruzioni:

```
Move   C, B
Add    C, A
```

Si osservi che usando questa sequenza di istruzioni nessuno dei contenuti delle locazioni A o B sarà sovrascritto.

In alcuni processori CISC un operando può essere in memoria ma l'altro deve essere in un registro. In questo caso, la sequenza di istruzioni per il compito richiesto sarebbe:

```
Move   Ri, A
Add    Ri, B
Move   C, Ri
```

Il formato generale dell'istruzione Move è:

Move destinazione, sorgente

dove sia sorgente sia destinazione possono essere una locazione di memoria o un registro del processore. L'istruzione Move generalizza le funzionalità delle istruzioni Load e Store utilizzate precedentemente nella discussione dei processori in stile RISC. Nell'istruzione Load, la sorgente è una locazione di memoria e la destinazione è un registro del processore. Nell'istruzione Store, la sorgente è un registro e la destinazione è una locazione di memoria. Mentre le istruzioni Load e Store sono vincolate allo spostamento di operandi tra memoria e registri del processore, l'istruzione Move ha una portata più ampia. Move può essere usata per spostare operandi immediati e per trasferire operandi tra due locazioni di memoria o tra due registri.

2.10.1 Altri indirizzamenti

La maggior parte dei processori CISC ha tutti i cinque modi di indirizzamento: immediato, di registro, assoluto, indiretto, con indice e spiazzamento. Tre modi di indirizzamento aggiuntivi sono spesso usati nei processori CISC.

Modi con autoincremento e autodecremento

Ci sono due modi di indirizzamento che sono particolarmente convenienti per l'accesso a elementi di dati in locazioni consecutive in memoria e per la realizzazione di pile.

- *Modo con autoincremento*: l'indirizzo effettivo di operando o risultato è contenuto in un *registro* di processore, il cui *nome* è specificato nell'istruzione; dopo avere calcolato l'indirizzo effettivo il contenuto del registro viene modificato incrementandolo.

Solitamente, siccome si usa come indirizzo il contenuto di un registro, il suo nome va racchiuso tra parentesi tonde (•) (similmente al modo indiretto da registro), ma si fa seguire il tutto dal simbolo "+" per indicare che dopo l'uso il registro viene modificato incrementandone il contenuto. Pertanto in linguaggio macchina simbolico si segnala il modo con autoincremento mediante la scrittura "(R*i*)+".

Resta da stabilire quante unità valga l'incremento. Lo scopo è quello di puntare all'elemento consecutivo in memoria, pertanto l'incremento dipende dalla dimensione dell'operando o risultato il cui indirizzo effettivo si sta calcolando. In generale il processore che dispone di modo con autoincremento è sensibile alla dimensione dell'operando o risultato cui l'istruzione si riferisce. In base alla dimensione il processore determina in modo automatico la quantità da addizionare quando incrementa: 1 per dimensione da un byte, 2 per dimensione da 16 bit, 4 per dimensione da 32 bit e così via. Quanto a sapere quale sia la dimensione del dato, di solito questa informazione è già incorporata nell'istruzione, nel nome o tramite altro simbolo, pertanto non occorre alcun nuovo simbolo per segnalarla, ma basta scrivere "(R*i*)+".

Come si intuisce facilmente il modo con autoincremento ammette una forma duale, chiamata *modo con autodecremento*, anch'essa molto utile per scandire la memoria ma nel verso opposto. Eccone la definizione:

- *Modo con autodecremento*: funziona in modo simile al *modo con autoincremento*, ma il contenuto del registro viene *decrementato prima* di usarne il contenuto come indirizzo effettivo.

In linguaggio macchina simbolico si segnala il modo con autodecremento mediante la scrittura "−(R*i*)", e non c'è bisogno di alcun commento, salvo dire che il segno "−" precede la solita coppia di parentesi tonde (•) per rammentare che il decremento del registro ne precede l'uso.

Nel modo con autodecremento la memoria viene dunque scandita in ordine inverso. Ci si potrebbe chiedere per quale motivo il decremento del registro sia anteposto all'uso invece di posporlo come nel modo con autoincremento. Usando i due modi insieme si riesce a gestire efficacemente la pila. Si può quasi dire che i due modi importino più per come si combinano che per come si usano indipendentemente.

Invece di aver bisogno di due istruzioni per impilare un nuovo elemento:

Subtract SP, #4

Move (SP), NUOVOELEMENTO

si può usare solo un'istruzione:

Move −(SP), NUOVOELEMENTO

Allo stesso modo, invece di aver bisogno di due istruzioni per spilare un elemento:

Move ELEMENTO, (SP)

Add SP, #4

si utilizza soltanto:

Move ELEMENTO, (SP)+

Modo relativo (a PC)

Finora il modo con indice e spiazzamento è stato applicato solo a registri di uso generale, ma se invece lo si applica al registro contatore di programma PC si ottiene un nuovo modo di indirizzamento con caratteristiche interessanti. In poche parole X(PC) specifica l'indirizzo effettivo EA = X + [PC], vale a dire la parola di memoria traslata di X byte rispetto a quella correntemente puntata da PC. Dunque l'indirizzo effettivo punta a una parola la cui posizione è specificata in modo relativo a PC. Siccome quest'ultimo identifica sempre il punto di avanzamento corrente del flusso di esecuzione, si capisce per quale motivo il nuovo modo di indirizzamento venga detto *modo relativo a contatore di programma*. La coppia di parentesi tonde (•) dà il solito segnale, ovvero che si tratta di un registro usato come puntatore. Eccone la definizione precisa:

- *Modo relativo a contatore di programma*: l'indirizzo effettivo di operando, risultato o destinazione di salto è calcolato *come nel modo con indice e spiazzamento*, prendendo però il *registro contatore di programma* PC come registro indice.

2.10.2 Bit di esito o condizione

Il processore tiene traccia di alcune informazioni ausiliarie relative all'esito di svariate operazioni, che servono come condizione per l'istruzione di salto. Tali informazioni sono espresse sotto forma di insieme di bit, chiamati *codici* o *bit di condizione* (*condition code flag*), o in breve *bit di esito*. I bit di esito sono raggruppati in un registro apposito interno al processore, chiamato *registro di stato* (*status register*). Ogni esito è binario, assumendo

valore 1 o 0 secondo l'andamento dell'operazione cui si riferisce. Ecco i quattro bit di esito più comuni:

N (negativo)	vale 1 se risultato negativo, 0 se positivo o nullo
Z (zero)	vale 1 se risultato nullo, 0 altrimenti
V (trabocco)	vale 1 se trabocco in comp. a due, 0 altrimenti (da *oVerflow*)
C (riporto)	vale 1 se trabocco in binario naturale, 0 altrimenti (da *Carry*)

Tutti i bit di esito vengono aggiornati quando si esegue un'istruzione logica o aritmetica, ma per i bit N e Z anche quando si esegue un'istruzione di trasferimento di dato come caricamento, memorizzazione e copia. In quest'ultimo caso, tramite una successiva istruzione di salto condizionato si riesce a variare l'esecuzione del programma secondo segno e valore del dato trasferito. Qualche processore dispone anche di un'istruzione speciale Test, che esamina il numero contenuto nel registro o nella memoria dato come operando e aggiorna i bit N e Z in conseguenza; qui i bit di esito stessi figurano come risultato implicito dell'istruzione.

Quando l'operazione lavora su numeri interi relativi, cioè in complemento a due, il bit V indica se si sia verificato trabocco. Come spiegato nel Paragrafo 1.4, il trabocco si verifica quando il risultato di un'operazione aritmetica è fuori dall'intervallo di valori che possono essere rappresentati dal numero di bit disponibili per il risultato stesso. In tale modo si riesce a controllare se l'operazione sia andata bene o in caso contrario a saltare a sottoprogramma per correggere l'errore. Un'istruzione quale Branch_if_overflow serve appunto a tale fine.

Quando l'operazione tratta numeri interi positivi o nulli, cioè in binario naturale, è il bit C (e non V) a indicare se si sia verificato trabocco. Si usa il bit C per lavorare con numeri positivi di lunghezza in bit superiore a quella della parola, ovvero in aritmetica a precisione multipla; l'argomento è discusso nel Capitolo 9.

Si consideri l'istruzione Branch nella Figura 2.6. Se si usano i codici di condizione allora l'istruzione Subtract metterà entrambi i bit N e Z a 0 se il contenuto del registro R2 è ancora maggiore di 0. Il salto desiderato potrebbe essere semplicemente specificato mediante:

Branch>0 CICLO

senza indicare il registro coinvolto nella verifica della condizione. Questa istruzione provoca un salto se né N né Z sono 1, ovvero se il risultato prodotto dall'istruzione Subtract non è negativo né uguale a zero. L'insieme di istruzioni di un calcolatore fornisce molte istruzioni di salto condizionato che permettono di controllare una varietà di condizioni. Le condizioni sono definite da espressioni logiche che coinvolgono i codici di condizione.

Per illustrare l'uso dei codici di condizione, si consideri nuovamente il programma in Figura 2.8, che somma una lista di numeri impiegando istruzioni in stile RISC. Usando un insieme di istruzioni in stile CISC, questo compito può essere effettuato da un numero minore di istruzioni, come mostrato in Figura 2.26. L'istruzione Add usa il registro puntatore

	Move	R2, N	Carica la dimensione della lista
	Clear	R3	Inizializza la somma a 0
	Move	R4, #NUM1	Carica l'indirizzo del primo numero
CICLO:	Add	R3, (R4)+	Aggiungi il prossimo numero alla somma
	Subtract	R2, #1	Decrementa il contatore
	Branch>0	CICLO	Salta indietro se non ancora finito

Figura 2.26 Una versione CISC del programma di Figura 2.8.

(R4) per accedere ai numeri consecutivi nella lista e li aggiunge alla somma nel registro R3. Dopo l'accesso all'operando sorgente, il processore incrementa automaticamente il puntatore, poiché è stato usato il modo di indirizzamento con autoincremento per l'accesso detto. L'istruzione Subtract assegna i codici di condizione che sono quindi usati dall'istruzione Branch.

2.11 Stili RISC e CISC

RISC e CISC sono due stili diversi di insiemi di istruzioni. Si è introdotto prima RISC poiché è più semplice e facile da capire. Dopo aver visto alcuni aspetti di base di entrambi gli stili, si riassumono qui sotto le loro principali caratteristiche.

Lo stile RISC è caratterizzato da:

- modi di indirizzamento semplici;
- tutte le istruzioni occupanti una singola parola;
- meno istruzioni nell'insieme di istruzioni, in conseguenza della semplicità dei modi di indirizzamento;
- operazioni aritmetiche e logiche che possono essere effettuate solo su operandi nei registri dei processori;
- architettura carica/immagazzina che non permette trasferimenti diretti da una locazione di memoria a un'altra; tali trasferimenti devono essere effettuati attraverso un registro del processore;
- istruzioni semplici che favoriscono l'esecuzione veloce da parte dell'unità di elaborazione usando tecniche come l'elaborazione a stadi, presentata nel Capitolo 6;
- programmi tendenzialmente di dimensioni più grandi, poiché un maggior numero di istruzioni più semplici sono necessarie per svolgere compiti complessi.

Lo stile CISC è caratterizzato da:

- modi di indirizzamento più complessi;
- istruzioni più complesse, in cui un'istruzione può impegnare varie parole di memoria;
- tante istruzioni che effettuano compiti complessi;
- operazioni aritmetiche e logiche che possono usare operandi in memoria come pure operandi nei registri del processore;
- trasferimenti da una locazione di memoria a un'altra per mezzo di una singola istruzione Move;
- programmi tendenzialmente di dimensioni più piccole, poiché un minor numero di istruzioni più complesse sono necessarie per svolgere compiti complessi.

Prima degli anni '70, tutti i calcolatori erano di tipo CISC. Un obiettivo importante fu semplificare lo sviluppo del software rendendo l'hardware capace di effettuare compiti abbastanza complessi, ovvero spostare la complessità dal livello del software al livello dell'hardware. Questo facilita la produzione di programmi più semplici e corti, il che fu importante quando la memoria dei calcolatori era più piccola e costosa. Oggi, la memoria è poco costosa e molti calcolatori ne hanno in grandi quantità.

Progetti per lo stile RISC furono il risultato di un tentativo per ottenere prestazioni molto elevate rendendo l'hardware molto semplice, così che le istruzioni potessero essere eseguite molto rapidamente in modalità di elaborazione a stadi, come sarà discusso nel Capitolo 6. Questo significa spostare la complessità dal livello dell'hardware al livello del software. Compilatori

sofisticati furono sviluppati per ottimizzare il codice costituito da istruzioni semplici. La dimensione del codice diventò meno importante con il crescere delle capacità della memoria.

Mentre gli stili RISC e CISC sembrano definire due approcci significativamente diversi, i processori di oggi mostrano quel che potrebbe sembrare un compromesso tra questi approcci. Per esempio, è invitante aggiungere alcune istruzioni non RISC a un processore RISC al fine di ridurre il numero di istruzioni eseguite, a patto che l'esecuzione di queste nuove istruzioni sia veloce. Affronteremo le prestazioni in dettaglio nel Capitolo 6, in cui discuteremo il concetto di elaborazione a stadi.

2.12 Esempi di programma

Qui verranno discussi due esempi significativi di programmazione in linguaggio macchina che mettono bene in evidenza alcuni usi caratteristici delle istruzioni. Il primo è di tipo numerico e in particolare riguarda Algebra Lineare (teoria di vettori e matrici), Geometria e Meccanica; l'altro è di tipo non-numerico.

2.12.1 Prodotto scalare di vettori

Il primo esempio è il calcolo del *prodotto scalare* (o *prodotto interno*) di due vettori di numeri. Siano A e B due vettori di numeri interi relativi rappresentati in complemento a due, ciascuno di dimensione $n \geq 1$; gli elementi si denotano A_i, B_i $(0 \leq i \leq n - 1)$. Il prodotto scalare $A \times B$ è il numero intero relativo dato dalla formula seguente:

$$A \times B = A_0 \times B_0 + A_1 \times B_1 + \ldots + A_{n-1} \times B_{n-1} = \sum_{i=0}^{n-1} A_i \times B_i$$

Per calcolare $A \times B$ basta estendere il ciclo per il calcolo della somma degli elementi della lista, già mostrato nelle Figure 2.8 e 2.26. Le Figure 2.27 e 2.28 mostrano programmi negli

	Move	R2, #AVETT	R2 punta al vettore A
	Move	R3, #BVETT	R3 punta al vettore B
	Load	R4, N	R4 è usato come contatore
	Clear	R5	In R5 si accumula il prodotto scalare
CICLO:	Load	R6, (R2)	Preleva il prossimo elemento del vettore A
	Load	R7, (R3)	Preleva il prossimo elemento del vettore B
	Multiply	R8, R6, R7	Calcola il prodotto della coppia successiva
	Add	R5, R5, R8	Aggiungilo alla somma precedente
	Add	R2, R2, #4	Incrementa il puntatore al vettore A
	Add	R3, R3, #4	Incrementa il puntatore al vettore B
	Subtract	R4, R4, #1	Decrementa il contatore
	Branch_if_[R4]>0	CICLO	Reitera se non ancora finito
	Store	R5, PROD	Immagazzina il prodotto scalare in memoria

Figura 2.27 Un programma in stile RISC per calcolare il prodotto scalare di due vettori.

	Move	R2, #AVETT	R2 punta al vettore A
	Move	R3, #BVETT	R3 punta al vettore B
	Move	R4, N	R4 è usato come contatore.
	Clear	R5	In R5 si accumula il prodotto scalare
CICLO:	Move	R6, (R2)+	Calcola il prodotto dei componenti successivi
	Multiply	R6, (R3)+	
	Add	R5, R6	Aggiungilo alla somma precedente
	Subtract	R4, #1	Decrementa il contatore
	Branch>0	CICLO	Reitera se non ancora finito
	Move	PROD, R5	Immagazzina il prodotto scalare in memoria

Figura 2.28 Un programma in stile CISC per calcolare il prodotto scalare di due vettori.

stili RISC e CISC per calcolare il prodotto scalare (*dot product*) e memorizzarlo nella locazione di memoria PROD. I primi elementi di ciascun vettore, A_0 e B_0, sono immagazzinati nelle locazioni di memoria AVETT e BVETT, mentre i rimanenti elementi nelle parole delle locazioni successive.

Il compito di accumulare una somma di prodotti viene svolto in molte applicazioni per elaborazione di segnali. In questo caso, uno dei vettori consiste dei più recenti n campioni di un segnale in una sequenza continua nel tempo forniti in input a un'unità di elaborazione di segnali. L'altro vettore è un insieme di n pesi. Gli n campioni del segnale sono moltiplicati per i pesi e la somma di questi prodotti costituisce un campione del segnale in uscita.

Alcuni insiemi di istruzioni del calcolatore abbinano le operazioni delle istruzioni Multiply e Add usate nei programmi delle Figure 2.27 e 2.28 in una singola istruzione MultiplyAccumulate. Per esempio, questo è effettuato nel processore ARM (si veda l'Approfondimento applicativo 2.5).

2.12.2 Programma di ricerca di una stringa

Come esempio di applicazione non numerica, si consideri il problema della ricerca di una stringa. Date due stringhe di caratteri ASCII, una stringa T lunga e una stringa P corta, vogliamo determinare se la sequenza P sia una sottosequenza contigua della stringa T. Poichè P potrebbe essere trovata all'interno di T in più posti, semplificheremo il compito focalizzando l'interesse solo alla prima occorrenza di P in T, quando si ricerca su T scorrendola da sinistra a destra. Supponiamo che T e P consistano di n e m caratteri, rispettivamente, con $n > m$. I caratteri sono immagazzinati in memoria in byte di locazioni consecutive. Assumiamo che i dati richiesti siano posizionati come segue:

- T è l'indirizzo di $T(0)$, che è il primo carattere della stringa T;
- N è l'indirizzo della parola di 32 bit che contiene il valore n;
- P è l'indirizzo di $P(0)$, che è il primo carattere della stringa P;
- M è l'indirizzo della parola di 32 bit che contiene il valore m;
- RISULTATO è l'indirizzo della parola in cui sarà memorizzato il risultato della ricerca. Se la sottostringa P viene trovata in T, allora l'indirizzo della locazione corrispondente in T sarà memorizzato in RISULTATO, altrimenti vi sarà memorizzato il valore -1.

Approfondimento applicativo 2.5

Codici operativi e sintassi delle istruzioni

Proponiamo qui una sintesi della sintassi delle istruzioni viste finora nei quattro processori in considerazione. Le istruzioni sono raccolte in sei categorie principali e i loro codici mnemonici sono mostrati in Tabella A2.7, che in qualche caso indica anche i modi di indirizzamento degli operandi. Altre indicazioni su questo e altri aspetti della sintassi delle istruzioni sono fornite nelle successive note esplicative. Ulteriori dettagli sulle istruzioni IA-32 sono reperibili nel Caso di studio C2. La sintesi qui proposta non è esaustiva; si veda la bibliografia [1, 7, 3, 4, 10] per le ISA complete di tutti i dettagli.

Il numero degli operandi e i modi di indirizzamento ammessi dipendono dal tipo di istruzione. Per esempio, le istruzioni RISC Load e Store hanno due operandi, con il modo di registro per uno di essi, mentre l'altro operando è in memoria. Come in questo caso indica la tabella, con la notazione adottata in Tabella A2.2, l'ISA NIOS II prescrive l'indirizzamento con base e spiazzamento dell'operando in memoria, mentre l'ISA ARM ammette tutti i modi di indirizzamento della memoria. Inoltre, entrambi i processori hanno registri a 32 bit, ma il dato da trasferire può avere dimensione più piccola, indicata dal suffisso del codice mnemonico che designa *byte, halfword, word*. Il suffisso è obbligatorio nel formato NIOS II, mentre è opzionale nel formato ARM, dove il default è *word*. Nelle istruzioni Load, se il dato ha dimensione più piccola l'ulteriore suffisso "u" (*unsigned*) nel formato NIOS II ne indica l'estensione con zeri, mentre il default è l'estensione del segno.

Tabella A2.7a Istruzioni NIOS II, ColdFire, ARM e IA-32

Tipo di istruzioni	NIOS II	ColdFire	ARM	IA-32
(a) Istruzioni di trasferimento, di controllo				
Trasferimento				
Load	ld*b* *ri*, X(*rj*)		LDR*b*	
Store	st*b* *ri*, X(*rj*)		STR*b*	
Move	mov*a*	MOVE*a.b*	MOV, MVN	MOV, LEA, PUSH, POP
Multiplo		MOVEM.*b*	LDM*w*, STM*w*	POPAD, PUSHAD
Controllo				
Salto incondiz.	br *l*, jmp *ri*	JMP	B *l*	JMP
Salto condiz.	bc *ri*, *rj*, *l*	Bc *l*	Bc *l*	Jc *l*
				LOOP *l*
Chiamata e rientro: si veda Tabella A2.5				

Legenda:
b suffisso del codice operativo, dimensione del dato in memoria, estensione a 32 bit: NIOS II: $b \in$ {w,b,h,bu,hu}, ColdFire: $b \in$ {B,W,L}, ARM: $b \in$ {B,H,SB,SH} opzionale;
a suffisso opzionale del codice operativo, modo di indirizzamento: NIOS II: $a \in$ {i,ui,ia}, indirizzamento immediato, ColdFire: $a \in$ {A,Q}, se la dest. è un registro indirizzo o dati, rispettivamente;
w suffisso del codice operativo, progressione del registro di base, $w \in$ {IA,DA,IB,DB}; *l*, etichetta; *c*, suffisso del codice operativo, condizione aritmetico-logica di salto: NIOS II: $c \in$ {eq,ne,ge,geu,gt,gtu,le,leu,lt,ltu}, ColdFire, ARM, IA-32: si veda Tabella A2.8

Tabella A2.7b Istruzioni NIOS II, ColdFire, ARM e IA-32

Tipo di istruzioni	NIOS II	ColdFire	ARM	IA-32
(b) Istruzioni aritmetiche, di confronto, logiche, di scorrimento				
Aritmetiche				
Addizione	add*m*	ADD*m*.L	ADD*f*, ADC	ADD, ADC
Sottrazione	sub*m*	SUB*m*.L	SUB*f*, SBC	SUB, SBB
		NEG*m*.L		NEG
Moltiplicazione	mul*m*	MUL*s.b*	MUL, MLA	IMUL
Divisione	div, divu	DIV*s.b*		IDIV
Resto		REM*s*.L		
Altre		EXT.*b*		INC, DEC
		CLR.*b*		
Confronto	cmp*cm*	CMP*m*.L	CMP, CMN	CMP
Logiche				
Congiunzione	and*m*, andhi	AND*m*.L	AND, TST	AND
Disgiunzione	or*m*, orhi	OR*m*.L	ORR	OR
Disg. esclusiva	xor*m*, xorhi	EOR*m*.L	EOR, TEQ	XOR
Negazione		NOT*m*.L		NOT
Altre	nor		BIC	
Scorrimento				
Logico	srl*m*	LSR.L	*t*, LSR	SHR
	sll*m*	LSL.L	*t*, LSL	SHL
Aritmetico	sra*m*	ASR.L	*t*, ASR	SAR
		ASL.L		SAL
Rotazione	ror		*t*, ROR	ROR, RCR
	rol*m*			ROL, RCL

Legenda:
m suffisso opzionale del codice operativo, NIOS II: $m \in$ {i}, secondo operando sorgente immediato, ColdFire: $m \in$ {I,A,X}, sorgente immediato (sola opzione nelle istruzioni logiche), o dest. registro indirizzo, o riporto in ingresso da bit di esito X (sola opzione in NEG, esclusa in CMP);
f suffisso opzionale del codice operativo, imposta i bit di esito C, V se f = S; *s*, suffisso del codice operativo, operandi con o senza segno, $s \in$ {S,U};
b suffisso del codice operativo, dimensione dell'operando sorgente o di destinazione; ColdFire: $b \in$ {W,L} in MUL, DIV; $b \in$ {B,W,L} in EXT, CLR;
c suffisso del codice operativo, condizione di confronto (come nelle istruzioni di salto); *t*, istruzione MOV o ADD, lo scorrimento si applica al secondo operando sorgente.

Viceversa, nel formato ARM il suffisso S indica estensione del segno, quella con zeri è il default.

L'ISA ARM dispone anche di istruzioni per il trasferimento multiplo di un insieme di registri da/a un blocco di locazioni contigue in memoria, come si è già visto nell'Approfondimento applicativo 2.4 per le istruzioni di salvataggio in pila e ripristino dei registri nel collegamento di sottoprogrammi. I codici mnemonici lì introdotti, convenienti nella gestione di pila, corrispondono a quelli in Tabella A2.7 come segue: STMFD = STMDB, LDMFD = LDMIA. I suffissi qui sono le iniziali di *increment, decrement, before, after*. Si indirizzano le locazioni contigue di memoria tramite un registro di base, con uno spiazzamento

fisso di 4 (che non occorre indicare), e tutti i modi di indirizzamento con registro di base sono ammessi, con e senza *writeback*.

Le istruzioni Move NIOS II non corrispondono a codici operativi di istruzioni macchina, bensì sono delle *pseudoistruzioni*, realizzate tramite altre istruzioni (si veda il Paragrafo 2.4.2). Quelle ARM sono istruzioni macchina, dove l'istruzione MVN carica il complemento binario dell'operando sorgente. Un'istruzione MOV con operando sorgente immediato negativo è però una pseudoistruzione, poiché è realizzata da una MVN con operando sorgente immediato positivo. Un'altra utile pseudoistruzione ARM è quella di caricamento di registro con operando sorgente *pseudoimmediato*, di forma:

LDR R*d*, =Val

trattata come segue dall'assemblatore: se 9 bit bastano a rappresentare Val (in valore assoluto e segno), allora la pseudoistruzione è realizzata dall'istruzione MOV R*d*, #Val, cioè con indirizzamento immediato, altrimenti l'assemblatore alloca una parola di memoria nell'area dati, la inizializza con Val, genera per essa una nuova etichetta, si supponga LV, e realizza la pseudoistruzione mediante l'istruzione LDR R*d*, LV, che usa il modo relativo a PC.

In entrambe le ISA, le istruzioni Move servono solo a caricare un registro con il contenuto di un altro o con un valore immediato. I suffissi NIOS II "u" e "a" stanno per *unsigned* e *address*; quest'ultimo indica il caricamento dell'indirizzo associato a un'etichetta, fornita come operando sorgente, dove è l'indirizzo a essere trattato come valore a 32 bit. Di natura ben diversa sono le istruzioni Move nei due processori CISC qui considerati, che ampliano le possibilità di trasferimento viste sinora per i due processori RISC. Tutti i modi di indirizzamento sono ammessi per entrambi gli operandi nel formato ColdFire, tranne immediato e relativo a PC per la destinazione. Non tutte le loro combinazioni sono però ammesse, per garantire che la lunghezza dell'istruzione macchina non superi le tre parole (da 16 bit); per esempio, non si può usare l'indirizzamento assoluto per entrambi gli operandi. La dimensione del dato è indicata da un suffisso dove *word* corrisponde a *halfword* delle ISA precedenti, mentre *longword* è un dato a 32 bit. I suffissi opzionali "A" e "Q" (*address*, *quick*) indicano rispettivamente che la destinazione è un registro indirizzo (solo con suffissi W o L), e che l'operando sorgente è un dato da 8 bit mentre la destinazione è un registro dati. Dell'istruzione di trasferimento multiplo di registri si è già detto nell'Approfondimento applicativo 2.4.

Come già detto, nell'ISA IA-32 non più di un operando può essere in memoria. L'istruzione LEA (*Load Effective Address*), analogamente alla movia NIOS II, tratta l'indirizzo sorgente come valore, da caricare nel registro di destinazione. In questo caso però l'indirizzo può essere specificato in qualsiasi modo di indirizzamento, pertanto può variare dinamicamente.

L'ISA NIOS II dispone di due istruzioni di salto incondizionato, essenzialmente perché la forma con destinazione specificata da etichetta è impiegabile solo se la distanza dell'indirizzo di quest'ultima da quello dell'istruzione successiva a quella di salto è rappresentabile con segno in 16 bit. Se ciò non è possibile, si ricorre all'istruzione jmp, dopo avere caricato il registro che ne ospita l'operando con l'indirizzo di destinazione del salto. Le altre ISA hanno una sola istruzione del tipo in questione. La destinazione ARM è specificata da un'etichetta, dunque l'indirizzo è relativo a PC ma con 26 bit disponibili per lo spiazzamento (con segno). Nelle due ISA CISC sono ammessi molti modi di indirizzamento. Nell'ISA ColdFire, questi coincidono con quelli indicati per la destinazione *a* di un'istruzione JSR in Tabella A2.5. L'ISA IA-32 offre per questa istruzione anche il modo indiretto da memoria, riservato a indirizzi di istruzioni.

L'ISA NIOS II non ha codici di condizione, il che spiega il formato a tre indirizzi, tipico dei processori RISC, anche per le istruzioni di salto condizionato, mostrato in Tabella A2.7. La condizione di salto indicata dal suffisso (di facile interpretazione tenendo conto che "u" indica che gli operandi sono considerati interi senza segno) si applica agli operandi nei registri, mentre il terzo operando è l'etichetta della destinazione del salto, sottoposta alla stessa restrizione che nell'istruzione br.

Nelle altre ISA in considerazione, dove l'etichetta della destinazione del salto è l'unico operando dell'istruzione, la condizione denotata dal suffisso del codice operativo è un'espressione logica in termini di bit di esito. Nell'ISA ARM, oltre alle 14 condizioni di forma identica a quelle ColdFire, è definita anche la condizione AL (*always*), sempre soddisfatta, che può essere omessa (costituisce il default); l'istruzione di salto incondizionato ha pertanto la stessa forma di quelle di salto condizionato, con tale condizione implicita. Un aspetto peculiare dell'uso dei bit di esito nell'ISA ARM è che *tutte* le istruzioni sono condizionate: i 4 bit più significativi nel formato binario delle istruzioni macchina sono riservati alla condizione, permettendo così la codifica di 16 diverse condizioni (una delle 16 configurazioni binarie è riservata a istruzioni la cui esecuzione deve essere sempre incondizionata).

La Tabella A2.8 mostra la corrispondenza dei suffissi di condizione (con il metasimbolo "|" quale separatore di sinonimi) nelle ISA ColdFire, ARM e IA-32, nonché le espressioni logiche che definiscono le condizioni in termini di bit di esito. Questi sono denominati come nel Paragrafo 2.10.2 nelle ISA ColdFire e ARM, mentre i loro nomi IA-32 vi corrispondono come segue: N → SF, Z → ZF, V → OF, C → CF. Per semplicità, qui si usano in tutti i casi i nomi N, Z, V, C. Le espressioni logiche in Tabella A2.8 sono formulate nella sintassi della logica proposizionale, con i quattro suddetti nomi quali variabili, la notazione $\overline{\varphi}$ per la negazione di φ e l'operatore $\oplus$ di disgiunzione esclusiva.

Alle condizioni di confronto aritmetico fra interi con segno, indicate dai suffissi nelle ultime quattro righe in tabella, corrispondono simili condizioni per interi senza segno, indicate

Tabella A2.8 Condizioni sui bit di esito nelle ISA ColdFire, ARM e IA-32

Suffisso del codice mnemonico			
ColdFire	**ARM**	**IA-32**	**Condizione logica**
EQ	EQ	E|Z	Z
NE	NE	NE|NZ	$\overline{Z}$
CS|LO	CS|HS	C|B	C
CC|HS	CC|LO	NC|AE	$\overline{C}$
MI	MI	S	N
PL	PL	NS	$\overline{N}$
VS	VS	O	V
VC	VC	NO	$\overline{V}$
HI		A	$\overline{C \vee Z}$
	HI		$\overline{\overline{C} \vee Z}$
LS		BE	$C \vee Z$
	LS		$\overline{C} \vee Z$
GE	GE	GE	$\overline{N \oplus V}$
LT	LT	L	$N \oplus V$
GT	GT	G	$\overline{Z \vee (N \oplus V)}$
LE	LE	LE	$Z \vee (N \oplus V)$

nello stesso ordine da HS, LO, HI, HS nelle ISA ColdFire e ARM, e da AE, B, A, BE nell'ISA IA-32. Può stupire che a tali suffissi corrispondano condizioni logiche *diverse* nell'ISA ARM rispetto alle altre due (dove coincidono). La ragione sta in convenzioni diverse che inducono l'uso opposto del bit di riporto C nell'ISA ARM rispetto alle altre.

L'istruzione IA-32 LOOP impiega il registro ECX come contatore delle iterazioni di un ciclo che inizi a un'istruzione di etichetta l, l'operando dell'istruzione. Il registro va inizializzato con il numero di iterazioni. L'esecuzione dell'istruzione lo decrementa ed esegue un salto condizionato JG l.

Le istruzioni RISC in Tabella A2.7b hanno quasi tutte il formato a tre indirizzi $OP\ d, s_1, s_2$, con operandi in registri, ma il secondo operando sorgente s_2 può spesso essere immediato.

Considerando le istruzioni RISC aritmetiche, quelle di sottrazione eseguono $d \leftarrow s_1 - s_2$, quelle di divisione NIOS II eseguono $d \leftarrow s_1/s_2$ con risultato la parte intera del quoziente. Tutte le operazioni sono effettuate su interi con segno in complemento a due, tranne divu che tratta gli operandi e il risultato quali interi senza segno. Le forme ADDS e ADC dell'addizione ARM ne permettono l'applicazione a operandi a 64 bit, dove per esempio ciascuno degli addendi occupa una coppia di registri contigui: l'esecuzione di ADDS sulle metà meno significative degli operandi imposta il bit di esito C in base al risultato (parziale), e la successiva esecuzione di ADC sulle metà più significative usa il bit C come riporto in ingresso. Le forme SUBS e SBC della sottrazione ARM hanno un uso analogo. Si può prescrivere uno scorrimento o rotazione del secondo operando sorgente dell'addizione o sottrazione ARM ponendola nella forma $OP\ d, s_1, s_2, \sigma s_3$, con $\sigma \in \{\text{LSL,LSR,ASR,ROR}\}$ e s_3 il numero di posizioni dello scorrimento o rotazione: un valore immediato o un registro che lo contiene. La moltiplicazione NIOS II carica la destinazione con i 32 bit meno significativi del risultato. Gli operandi sono considerati senza segno, ma il risultato è corretto anche per interi con segno, se rappresentabile in 32 bit. La moltiplicazione ARM ha due forme di base, di cui una analoga a quella NIOS II, l'altra a quattro operandi, MLA, che combina moltiplicazione e addizione. Il risultato è la somma del terzo operando sorgente e del prodotto degli altri due. Si hanno varianti delle forme di base che danno un risultato a 64 bit, in versioni diverse per operandi con segno e senza segno.

Le istruzioni CISC in Tabella A2.7(b) hanno quasi tutte il formato a due indirizzi, in quelle ColdFire nell'ordine $OP\ s, d$, mentre in quelle IA-32 hanno l'ordine opposto: $OP\ s, d$. Gli operandi possono essere immediati o in registri o in memoria, ma con specifiche limitazioni al riguardo.

Sia nelle istruzioni ColdFire sia in quelle IA-32 la sottrazione effettua $d \leftarrow d - s$ e la divisione usa l'operando sorgente come divisore, ma si rammenti il diverso ordine di sorgente e destinazione nei due formati. L'addizione e sottrazione ColdFire a 64 bit usano il bit di esito X, una sorta di riporto extra, quale riporto in ingresso per compiere l'operazione sulle metà più significative degli operandi. Il bit X non richiede una forma speciale dell'istruzione per essere impostato, dunque si usano ADD e ADDX così come si utilizzano rispettivamente ADDS e ADC nell'addizione ARM a 64 bit, similmente nella sottrazione. L'istruzione a un operando NEG, in tutte le sue forme, lo sostituisce con il suo complemento a due effettuandone la sottrazione da 0. Le corrispondenti istruzioni IA-32 sono simili, eccetto che si usa il bit di esito C quale riporto in ingresso nell'istruzione ADC. Le istruzioni IA-32 INC e DEC sono ovvie specializzazioni di addizione e sottrazione.

La destinazione di una moltiplicazione o divisione ColdFire dev'essere in un registro dati e il suffisso b ne specifica la dimensione quale secondo operando sorgente. Per esempio, MULS.W #3, Di triplica il numero intero con segno contenuto nella metà meno significativa di Di, per poi caricare questo registro con il risultato a 32 bit. La moltiplicazione di operandi entrambi a 32 bit tronca il risultato ai 32 bit meno significativi. La divisione ColdFire con suffisso W opera su un dividendo a 32 bit (nel registro dati) con un divisore a 16 bit (operando sorgente) e colloca il quoziente troncato a 16 bit nella metà meno significativa del registro di destinazione, mentre il resto della divisione va nell'altra metà. Nella divisione con suffisso L il resto si perde. Per ovviare a questo è prevista l'istruzione a tre indirizzi REMS.L s, Di:Dj, che pone in Dj il resto della divisione senza modificare Di; il quoziente può quindi essere prodotto da una successiva istruzione DIVS.L s, Di. Così pure per le versioni senza segno di tali istruzioni. Moltiplicazione e divisione IA-32 hanno una maggiore varietà di opzioni, trattate nel Caso di studio C2.

Delle istruzioni aritmetiche ColdFire, solo la CLR può non avere alcun operando in registro: azzera la destinazione, per la dimensione specificata. L'altra istruzione ColdFire a un operando, EXT, prescrive che esso sia in un registro dati e ne effettua l'estensione del segno in base alla dimensione b, con il seguente significato: b = L: 16 $\rightarrow$ 32 bit; b = W: 8 $\rightarrow$ 16 bit; b = B: 8 $\rightarrow$ 32 bit.

Anche nelle istruzioni di confronto si nota la differenza tra quelle NIOS II e le altre dovuta all'assenza di codici di condizione nell'ISA NIOS II. In questo caso, infatti, l'istruzione è a tre operandi, tutti in registri eccetto che il secondo operando sorgente può essere immediato, e il registro di destinazione riceve il risultato del confronto: 1 o 0 in base alla valutazione, vera o falsa, della condizione sulla coppia ordinata di operandi sorgenti. La condizione è specificata da un suffisso del codice operativo come nelle istruzioni di salto condizionato. Nelle altre ISA in considerazione le istruzioni di confronto sono a due operandi, poiché il risultato del confronto si manifesta nei codici di condizione, impostati in base all'esito della sua esecuzione. Questa consiste nell'esecuzione della sottrazione $d - s$, dove d è l'operando di destinazione e s quello sorgente, scartandone però il risultato. Nel caso dell'istruzione ARM CMN si esegue l'addizione $d + s$ invece della sottrazione. Le operazioni in questione sono eseguite a 32 bit in complemento a due.

Le istruzioni logiche, nei formati già visti per quelle aritmetiche, sono perlopiù riconducibili a quelle generiche descritte in formato RISC nel Paragrafo 2.8.1, da cui è facile dedurne le analoghe proprietà in formato CISC. Sono da aggiungere solo poche note. Le istruzioni logiche NIOS II con il suffisso hi usano un operando immediato a 16 bit quale metà più significativa di un argomento a 32 bit dell'operazione logica, dove la metà meno significativa è azzerata. Le istruzioni ARM TST e TEQ eseguono operazioni logiche, rispettivamente di congiunzione e disgiunzione esclusiva delle coppie di bit corrispondenti degli operandi, ma scartano il risultato, poiché l'unico effetto che se ne ottiene è l'impostazione dei bit di esito. Sono pertanto istruzioni a due operandi, dove quello sorgente può essere immediato. L'istruzione ARM BIC (*Bit Clear*) è equivalente a un'istruzione AND in cui ciascun bit del secondo operando sorgente venga complementato prima di eseguire la congiunzione.

Le istruzioni di scorrimento logico, aritmetico e rotazione eseguono le operazioni descritte in forma generica nel Paragrafo 2.8.2. Quelle NIOS II hanno la dimensione dello scorrimento quale secondo operando sorgente a 5 bit, i meno significativi in un registro oppure in modo immediato. La dimensione è l'operando sorgente in quelle ColdFire, dove la destinazione dev'essere un registro dati e l'operando sorgente immediato se il suo valore non supera 8, altrimenti in un registro dati nei 6 bit meno significativi. I bit di esito sono impostati in base al risultato dello scorrimento, ma nei bit di esito C e X si copia il valore dell'ultimo bit espulso dallo scorrimento. L'ISA ARM non ha istruzioni di scorrimento, ma queste si realizzano facilmente tramite altre istruzioni, quali ADD o MOVE, specificando lo scorrimento del secondo operando sorgente. Nell'ISA IA-32 la dimensione dello scorrimento è l'operando sorgente a 8 bit, immediato o nel registro CL; la destinazione è indirizzabile in qualunque modo. Le istruzioni di rotazione hanno anche le forme estese al bit di esito C.

```
for  i ← 0 to n − m do
     j ← 0
     while j < m and P[j] = T[i + j] do
        j ← j + 1
     if j = m return i
return −1
```

Figura 2.29 Un algoritmo di forza bruta per la ricerca di stringhe.

La ricerca di stringhe è un problema importante e ampiamente investigato. Molti algoritmi sono stati sviluppati. Poiché lo scopo attuale è principalmente quello di illustrare l'impiego di istruzioni assemblative, si userà l'algoritmo più semplice, noto come algoritmo di *forza bruta* (*brute-force*), che è illustrato in Figura 2.29.

In un calcolatore RISC, l'algoritmo può essere realizzato come mostrato in Figura 2.30. I commenti spiegano l'uso dei vari registri del processore. Si noti che, nel caso di fallimento di una ricerca, il valore immediato -1 porterà il contenuto di R8 a diventare uguale a 0xFFFFFFFF, che rappresenta -1 in complemento a due.

	Move	R2, #T	R2 punta alla stringa *T*
	Move	R3, #P	R3 punta alla stringa *P*
	Load	R4, N	Preleva il valore *n*
	Load	R5, M	Preleva il valore *m*
	Subtract	R4, R4, R5	Calcola $n - m$
	Add	R4, R2, R4	Indirizzo di $T(n - m)$
	Add	R5, R3, R5	Indirizzo di $P(m)$
CICLO1:	Move	R6, R2	Usa R6 per scandire la stringa *T*
	Move	R7, R3	Usa R7 per scandire la stringa *P*
CICLO2:	LoadByte	R8, (R6)	Confronta una coppia di caratteri delle stringhe *T* e *P*
	LoadByte	R9, (R7)	
	Branch_if_[R8]≠[R9]	NONUGUALI	
	Add	R6, R6, #1	Punta al successivo carattere di *T*
	Add	R7, R7, #1	Punta al successivo carattere di *P*
	Branch_if_[R5]>[R7]	CICLO2	Reitera se non ancora finito
	Store	R2, RISULTATO	Immagazzina l'indirizzo di $T(i)$
	Branch	FATTO	
NONUGUALI:	Add	R2, R2, #1	Punta al successivo carattere di *T*
	Branch_if_[R4]≥[R2]	CICLO1	Reitera se non ancora finito
	Move	R8, # −1	Scrivi −1 per indicare che non è stata trovata alcuna corrispondenza
	Store	R8, RISULTATO	
FATTO:	prossima istruzione		

Figura 2.30 Un programma in stile RISC per la ricerca di stringhe.

	Move	R2, #T	R2 punta alla stringa *T*
	Move	R3, #P	R3 punta alla stringa *P*
	Move	R4, N	Preleva il valore *n*
	Move	R5, M	Preleva il valore *m*
	Subtract	R4, R5	Calcola $n - m$
	Add	R4, R2	Indirizzo di $T(n - m)$
	Add	R5, R3	Indirizzo di $P(m)$
CICLO1:	Move	R6, R2	Usa R6 per scandire la stringa *T*
	Move	R7, R3	Usa R7 per scandire la stringa *P*
CICLO2:	MoveByte	R8, (R6)+	Confronta una coppia di caratteri delle stringhe *T* e *P*
	CompareByte	R8, (R7)+	
	Branch≠0	NONUGUALI	
	Compare	R5, R7	Confronta con l'indirizzo di $P(m)$
	Branch>0	CICLO2	Reitera se non ancora finito
	Move	RISULTATO, R2	Immagazzina l'indirizzo di $T(i)$
	Branch	FATTO	
NONUGUALI:	Add	R2, #1	Punta al successivo carattere di *T*
	Compare	R4, R2	Confronta con l'indirizzo di $T(n - m)$
	Branch≥0	CICLO1	Reitera se non ancora finito
	Move	RISULTATO, # −1	Non è stata trovata alcuna corrispondenza
FATTO:	prossima istruzione		

Figura 2.31 Un programma in stile CISC per la ricerca di stringhe.

La Figura 2.31 mostra come si possa realizzare l'algoritmo in un calcolatore CISC. Si osservi che la prima istruzione di CICLO2 carica un carattere dalla stringa *T* nel registro R8; questa istruzione è seguita da un'istruzione che confronta questo carattere con un altro nella stringa *P*. Il lettore potrebbe chiedersi perché non sia possibile usare una singola istruzione:

CompareByte (R6)+, (R7)+

per ottenere lo stesso risultato. Mentre gli insiemi di istruzioni CISC permettono operazioni che coinvolgono operandi in memoria, tipicamente esse richiedono che se un operando è in memoria, l'altro sia in un registro del processore. Un'eccezione comune è l'istruzione Move, che coinvolge due operandi in memoria. Questa fornisce un modo semplice per spostare dati tra due locazioni di memoria diverse.

2.13 Codifica numerica di istruzione

L'istruzione macchina è il comando elementare impartibile al processore. Nei paragrafi precedenti è stato presentato in modo graduale un insieme piuttosto ampio e diversificato di istruzioni macchina e modi di indirizzamento. Il linguaggio macchina scelto come esempio è generico, cioè non esiste un processore reale in grado di eseguirlo; esso è però rappresentativo perché contiene tutte le istruzioni più comuni nei vari linguaggi macchina specifici dei processori commerciali.

Il linguaggio macchina generico visto finora è di tipo simbolico, ma per essere interpretabile ed eseguibile da parte del processore esso va rappresentato o *codificato* (*encoded*) sotto

Approfondimento applicativo 2.6

Esempi e problemi di programmazione assemblativa

Nel seguito è illustrato un semplice esempio di programmazione assemblativa nei linguaggi nativi dei quattro processori qui considerati. Per il calcolo del prodotto scalare di due vettori, si sono proposti nel Paragrafo 2.12.1 due programmi assemblativi generici, rispettivamente in stile RISC e CISC, mostrati nelle Figure 2.27 e 2.28. Per facilitare il confronto in tale applicazione pratica degli stili di programmazione assemblativa nei processori qui considerati, si mostrano in Figura A2.1 le riformulazioni del programma generico RISC nei linguaggi assemblativi NIOS II e ARM, e in Figura A2.2 le analoghe riformulazioni del programma generico CISC nei linguaggi assemblativi ColdFire e IA-32.

In ciascuno dei due stili, i commenti ai due programmi in quello stile sono perlopiù applicabili a istruzioni corrispondenti, ma talvolta solo a uno di essi (nel qual caso si lascia vuota la riga nell'altro programma), il che può accadere per diverse ragioni: corrispondenza di un'istruzione in uno dei due programmi a una sequenza di più istruzioni nell'altro, risparmio di istruzioni grazie all'uso di modi di indirizzamento con modifica di registro indice ecc.

Il confronto dei programmi RISC NIOS II e ARM, in Figura A2.1, mostra chiaramente il risparmio di istruzioni permesso da caratteristiche CISC dei modi di indirizzamento dell'architettura ARM. In due casi, inizializzazione del contatore e immagazzinamento del prodotto, si risparmia un'istruzione per l'accesso a una locazione di memoria grazie al

	NIOS II		ARM	Commenti
	movia r2, AVETT		LDR R1, =AVETT	Puntatore al vettore A
	movia r3, BVETT		LDR R2, =BVETT	Puntatore al vettore B
	movia r4, N			Carica indirizzo di N
	ldw r4, (r4)		LDR R3, N	Inizializza contatore
	mov r5, r0		MOV R0, #0	Inizializza accumulatore
CICLO:	ldw r6, (r2)	CICLO	LDR R4, [R1], #4	Carica prossimo dato da A
	ldw r7, (r3)		LDR R5, [R2], #4	Carica prossimo dato da B
	mul r8, r6, r7		MLA R0, R4, R5, R0	Moltiplica i due dati e accumula la somma
	add r5, r5, r8			
	addi r2, r2, 4			Incrementa puntatore ad A
	addi r3, r3, 4			Incrementa puntatore a B
	subi r4, r4, 1		SUBS R3, R3, #1	Decrementa contatore
	bgt r4, r0, CICLO		BGT CICLO	Reitera se non ancora finito
	movia r2, PROD			Carica indirizzo del prodotto
	stw r5, (r2)		STR R0, PROD	Immagazzina il prodotto

Figura A2.1 Prodotto scalare di due vettori nei processori NIOS II e ARM.

	ColdFire		IA-32	Commenti
	MOVEA.L #AVETT, A1		LEA EBP, AVETT	Puntatore al vettore A
	MOVEA.L #BVETT, A2		LEA EBX, BVETT	Puntatore al vettore B
	MOVE.L N, D0		MOV ECX, N	Inizializza contatore
	CLR.L D1		MOV EAX, 0	Inizializza accumulatore
			MOV EDI, 0	Inizializza registro indice
CICLO:	MOVE.W (A1)+, D2	CICLO:	MOV EDX, [EBP + EDI * 4]	Carica dato da A
	MULS.W (A2)+, D2		IMUL EDX, [EBX + EDI * 4]	Moltiplica per dato da B
	ADD.L D2, D1		ADD EAX, EDX	Accumula la somma
			INC EDI	Incrementa registro indice
	SUBQ.L #1, D0			Decrementa contatore
	BGT CICLO		LOOP CICLO	Reitera se non ancora finito
	MOVE.L D1, PROD		MOV PROD, EAX	Immagazzina il prodotto

Figura A2.2 Prodotto scalare di due vettori nei processori ColdFire e IA-32.

modo di indirizzamento assoluto, che nell'architettura ARM è realizzato dall'assemblatore come indirizzamento relativo al PC. Il risparmio di altre due istruzioni, di incremento di registri indice, si deve all'uso del modo di indirizzamento con post-base e spiazzamento. Infine, alle due istruzioni NIOS II per il calcolo del prodotto di due elementi e l'aggiornamento cumulativo della somma corrisponde una sola istruzione ARM, che ne compone gli effetti.

Per le osservazioni precedenti non sorprende che gli analoghi programmi per i processori CISC abbiano lo stesso numero (o quasi) di istruzioni del programma ARM. Il risparmio reso dal *writeback* nel programma ARM si ottiene con il modo di indirizzamento con autoincremento nel programma ColdFire, dove l'entità dell'incremento è determinata dal tipo di operandi: nel programma si assume che siano a 16 bit (dunque l'autoincremento vale 2), considerati interi in complemento a due nella moltiplicazione con segno, il cui risultato è un intero a 32 bit in complemento a due. Inizializzazione e incremento dell'indice costano invece due istruzioni nel programma IA-32, dove però si guadagna il decremento automatico del contatore grazie all'uso dell'istruzione LOOP e al prescritto impiego del registro ECX quale contatore del ciclo. Infine, in entrambi i programmi CISC si risparmia un'istruzione per la possibilità di avere un operando della moltiplicazione in memoria.

A proposito di aritmetica, va notato che i programmi proposti non sono del tutto equivalenti: la moltiplicazione NIOS II è di interi senza segno a 32 bit, con risultato modulo 2^{32}, dunque dello stesso tipo; la moltiplicazione nel programma ARM ha caratteristiche analoghe, ma operandi e risultato possono essere interi in complemento a due e l'ISA ARM offre anche varianti delle istruzioni di moltiplicazione con risultato a 64 bit; la variante di moltiplicazione usata nel programma IA-32 ha le stesse proprietà e anche in questo caso sono disponibili altre varianti con risultato a 64 bit.

forma numerica binaria, ovvero come stringa di bit. Il processo di conversione da forma simbolica a numerica è effettuato dall'*assemblatore* (*assembler*), un componente del software di sistema, come già spiegato nel Paragrafo 2.5.

Esaminiamo adesso come si possano formare le istruzioni macchina. L'istruzione Add:

Add Rdst, Rsrc1, Rsrc2

è rappresentativa di una categoria di istruzioni a tre indirizzi che usano operandi immagazzinati nei registri del processore. I registri Rdst, Rsrc1 e Rsrc2 contengono l'operando di destinazione e due operandi sorgenti. Se un processore ha 32 registri, allora è necessario usare 5 bit per specificare ognuno dei tre registri in tali istruzioni. Se ogni istruzione è codificata in una parola a 32 bit, i rimanenti 17 bit possono essere usati per specificare il codice operativo che indica l'operazione da effettuare. Un possibile formato è mostrato in Figura 2.32a.

Consideriamo adesso istruzioni in cui un operando è fornito usando il modo di indirizzamento immediato, come

Add Rdst, Rsrc, #Valore

Dei 32 bit disponibili, 10 bit sono necessari per specificare i due registri. I rimanenti 22 bit devono fornire il codice operativo e il valore dell'operando immediato. Le dimensioni più utili per operandi immediati sono 32, 16 e 8 bit. Poiché 32 bit non sono disponibili, una buona scelta è allocare 16 bit per l'operando immediato. Questo lascia 6 bit per specificare il codice operativo. Un possibile formato è presentato in Figura 2.32b. Questo formato può essere usato anche per le istruzioni Load e Store, dove il modo di indirizzamento con indice e spiazzamento usa il campo da 16 bit per specificare lo spiazzamento che è aggiunto al contenuto del registro indice.

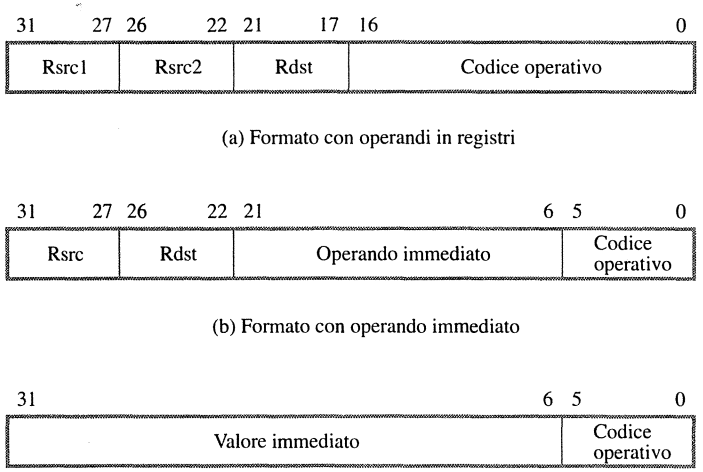


Figura 2.32 Possibili formati binari di istruzioni.

Il formato in Figura 2.32b può essere anche usato per codificare istruzioni di salto. Consideriamo il programma in Figura 2.12. L'istruzione di salto condizionato all'indirizzo di memoria 128 potrebbe essere scritta in un linguaggio assemblativo specifico mediante:

BGT R2, R0, CICLO

se il contenuto del registro R0 è zero, dove il codice mnemonico è l'acronimo di *Branch_if Greater Than*. I registri R2 e R0 possono essere specificati nei due campi registro in Figura 2.32b. Il codice operativo di 6 bit deve identificare l'operazione BGT. Il campo immediato a 16 bit può essere usato per fornire l'informazione necessaria a determinare l'indirizzo di destinazione del salto, che è la locazione dell'istruzione con l'etichetta CICLO. Generalmente l'indirizzo di destinazione consiste di 32 bit. Poiché non c'è spazio per 32 bit, l'istruzione BGT usa il campo immediato per fornire uno spiazzamento dalla locazione di questa istruzione nel programma alla destinazione del salto richiesto. Nel momento in cui l'istruzione BGT è eseguita, il contatore del programma, PC, è stato incrementato per puntare alla successiva istruzione, che è l'istruzione Store all'indirizzo 132. Quindi lo spiazzamento per il salto è $132 - 112 = 20$. Poichè il processore calcola l'indirizzo di destinazione sommando il contenuto corrente del PC e lo spiazzamento per il salto, lo spiazzamento richiesto in questo esempio è negativo, ovvero -20.

Infine, dovremmo considerare l'istruzione Call, la quale è usata per chiamare un sottoprogramma. È necessario soltanto specificare il codice operativo e un valore immediato che è usato per determinare l'indirizzo della prima istruzione del sottoprogramma. Se vengono usati 6 bit per il codice operativo, allora i rimanenti 26 bit possono essere usati per indicare il valore immediato. Questo fornisce il formato mostrato in Figura 2.32c.

In questo paragrafo è stato introdotto il concetto di base della codifica di istruzioni macchina. Processori commerciali diversi hanno insiemi di istruzioni i cui dettagli realizzativi variano. Si rinvia alla documentazione ufficiale dei processori trattati negli approfondimenti applicativi di questo capitolo, per ciò che concerne la codifica binaria delle loro istruzioni.

Sommario

Questo capitolo ha introdotto la rappresentazione e l'esecuzione di istruzioni e programmi ai livelli assemblativo e macchina dal punto di vista del programmatore. La discussione ha evidenziato i principi di base delle tecniche di indirizzamento e l'ordinamento delle istruzioni. Gli esempi di programmazione hanno illustrato i tipi di base delle operazioni fornite dall'insieme di istruzioni di qualsiasi calcolatore moderno. Sono stati introdotti modi di indirizzamento di uso comune. Sono stati discussi il concetto di sottoprogramma e le istruzioni necessarie a realizzarlo. Nel corso del capitolo è stato sviluppato il confronto tra due differenti approcci alla progettazione di insiemi di istruzioni macchina, ovvero gli approcci RISC e CISC.

Note bibliografiche

1. Altera Corp., *Nios II Processor Reference Handbook*, version 11, Maggio 2011. http://www.altera.com/.
2. Altera Corp., *Introduction to the Nios II Soft Processor*, Altera University program, Maggio 2011. http://www.altera.com/education/univ/unv-index.html.
3. Advanced RISC Machines Ltd., *ARM Architecture Reference Manual*, ARMv6, documento DDI 0100I, Luglio 2005. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0100i/.
4. Advanced RISC Machines Ltd., *ARM and Thumb-2 Instruction Set Quick Reference Card*, documento ARM QRC 0001M, Settembre 2008. http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/.

5. Advanced RISC Machines Ltd., *RealView Compilation Tools, Assembler Guide*, documento DUI 0204J, Dicembre 2010. http://infocenter.arm.com/help/topic/com.arm.doc.dui0204j/.
6. Hohl J., *ARM Assembly Language: Fundamentals and Techniques*, CRC Press, Marzo 2009. http://www.crcpress.com/product/isbn/9781439806104.
7. Freescale Semiconductor Inc., *ColdFire Family Programmers - Reference Manual*, documento CFPRM Rev. 3, Marzo 2005. http://www.freescale.com/files/dsp/doc/CFPRM.pdf.
8. Freescale Semiconductor Inc., *Code Warrior Development Studio, Assembler Reference for ColdFire Processors*, documento CFASSEM_RM Revised, Ottobre 2006. http://www.freescale.com/files/soft_dev_tools/doc/ref_manual/CFASSEM_RM.pdf.
9. Free Software Foundation, GNU Project, *GNU Binutils – Documentation for binutils 2.22*, Novembre 2011. http://www.gnu.org/software/binutils/manual/.
10. Intel Corp., *Intel 64 and IA-32 Architectures Software Developer's Manuals, Volume 1: Basic Architecture*, documento n. 253665-041US, Dicembre 2011. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.
11. Microsoft Corp., *Microsoft Macro Assembler Reference*, Visual Studio, Visual C++ Reference, 2005 (2008, 2010). http://msdn.microsoft.com/en-us/library/.
12. Microsoft Corp., *MASM 6.1 Reference Guide*, documento n. DB35749-1292, Redmond 1992 (ristampa, Ottobre 2000). http://webster.cs.ucr.edu/Page_TechDocs/.

3 Operazioni di ingresso e uscita

In questo capitolo

- Trasferimento di dati tra processore e dispositivi di ingresso/uscita o I/O (*input/output*)
- Trasferimenti di I/O dal punto di vista del programmatore
- Come viene effettuato l'I/O controllato da programma usando la scansione (*polling*)
- Come sono usate le interruzioni nei trasferimenti di I/O

Tra le funzioni fondamentali del calcolatore vi è lo scambio di dati con una varietà di dispositivi di *ingresso e uscita* o di I/O (*input/output*), o altrimenti detto di *dispositivi di periferia*, ovvero *periferiche*. Per esempio tale funzione permette all'utente di preparare testi, documenti e disegni tramite tastiera e video, e in generale di interagire con il calcolatore. Gli esseri umani fanno ampio uso del calcolatore per comunicare con altri sistemi di calcolo tramite la rete Internet e per procurarsi informazioni da ogni parte del mondo, ma esistono anche molte altre applicazioni dove il calcolatore è dominante seppure meno visibile. Il calcolatore entra come componente essenziale nelle applicazioni domestiche, negli impianti industriali, nei sistemi di trasporto, nei sistemi informativi bancari e commerciali e in tanti altri ancora. In ognuna di tali applicazioni il calcolatore è collegato e acquisisce informazioni da dispositivi sensori come telecamere, microfoni, dispositivi di allarme e così via. Oltre ad acquisire informazioni può mandare segnali e comandi a dispositivi attuatori come altoparlanti, motori, valvole, robot (solo per citarne alcuni). Insomma il calcolatore deve essere in grado di scambiare informazioni con uno spettro amplissimo di dispositivi di ingresso e uscita, collocati in ambienti molto differenti. Tali attività vanno sotto il nome di operazioni di ingresso e uscita o di I/O.

Nel capitolo si prenderanno in esame le varie metodologie secondo cui il calcolatore può svolgere le operazioni di ingresso e uscita, dal punto di vista del programmatore. Saranno presentate solo le operazioni di I/O di base, che sono disponibili in tutti i calcolatori. Questa conoscenza consentirà al lettore di svolgere esercizi interessanti e utili sui dispositivi presenti in un tipico ambiente di laboratorio didattico. Schemi di I/O più complessi, come anche l'hardware necessario per realizzare le operazioni di I/O, saranno discussi nel Capitolo 7.

3.1 Accesso a dispositivo di I/O

I componenti di un calcolatore comunicano tra loro mediante una rete di interconnessione, come mostrato in Figura 3.1. La rete di interconnessione[1] consiste dei circuiti necessari per trasferire informazioni tra processore, unità di memoria e un certo numero di dispositivi di I/O.

[1] Comunemente un bus unico, chiamato *bus di sistema* (*system bus*).

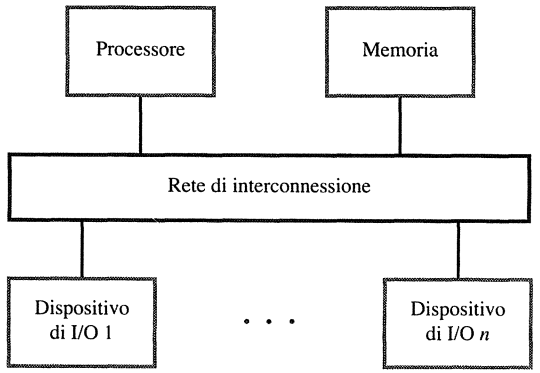


Figura 3.1 Componenti di un calcolatore.

Nel Capitolo 2 è stato descritto il concetto di spazio degli indirizzi e come il processore possa accedere a delle locazioni di memoria individuali all'interno di tale spazio degli indirizzi. Le istruzioni Load e Store usano i modi di indirizzamento per generare gli indirizzi effettivi che identificano le locazioni desiderate. Questa idea di usare gli indirizzi per accedere a varie locazioni in memoria può essere estesa per gestire anche i dispositivi di I/O. A questo scopo, ogni dispositivo di I/O deve apparire al processore come un insieme di locazioni indirizzabili, esattamente come la memoria. Alcuni indirizzi dello spazio degli indirizzi del processore sono assegnati a queste locazioni di I/O, anziché alla memoria principale. Queste locazioni sono di solito realizzate come circuiti per immagazzinare bit (*flip-flop*) organizzati in forma di registri. Di solito ci si riferisce a essi come *registri di I/O*. Poiché i dispositivi di I/O e la memoria condividono lo stesso spazio degli indirizzi, questa disposizione è chiamata *unificazione degli spazi di indirizzamento di memoria e I/O* (*memory mapped I/O*). Questa è usata nella maggior parte dei calcolatori.

Se si adotta tale tecnica qualsiasi istruzione macchina capace di lavorare con la memoria è automaticamente in grado di trasferire dati anche con le unità di I/O. Per esempio, se l'indirizzo simbolico DATO_ING corrisponde al registro buffer di ingresso della tastiera, l'istruzione seguente:

Load R2, DATO_ING

legge il carattere (byte) battuto sulla tastiera e lo scrive nel registro R2 del processore, vale a dire effettua l'introduzione del carattere da tastiera o, se si preferisce, la *lettura* della tastiera. Al contrario l'istruzione macchina seguente:

Store R2, DATO_USC

scrive il contenuto (byte) del registro R2 del processore nella parola collocata all'indirizzo simbolico DATO_USC, che in effetti è il registro buffer di uscita di una periferica quale il video o la stampante; questa è l'operazione di stampa o *scrittura* su video o stampante.

3.1.1 Interfaccia dei dispositivi di I/O

Un dispositivo di I/O è collegato alla rete di interconnessione mediante un circuito, chiamato *interfaccia del dispositivo*, che fornisce i mezzi per effettuare trasferimenti di dati e per scambiare informazioni di stato e di controllo necessarie a facilitare i trasferimenti di dati e a controllare il funzionamento del dispositivo. L'interfaccia include alcuni registri accessibili dal

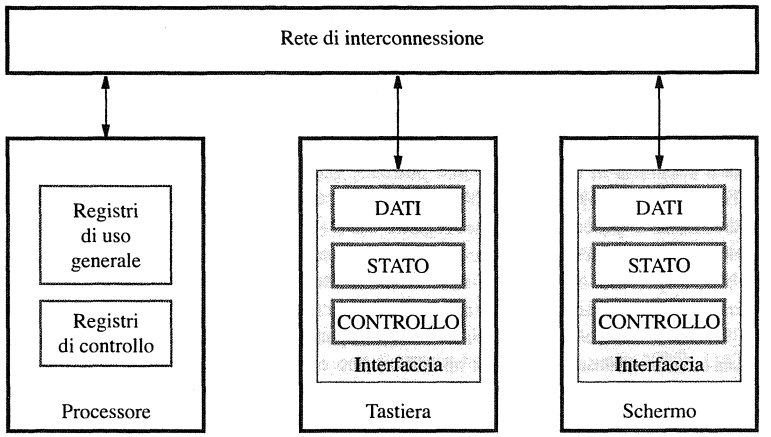


Figura 3.2 Interconnessione di processore, tastiera e schermo.

processore. Un registro può servire come buffer per i trasferimenti di dati, un altro può tenere informazioni sullo stato corrente del dispositivo, un altro ancora può memorizzare le informazioni che controllano il modo di operare del dispositivo. A questi registri di *dati*, *stato* e *controllo* si accede dalle istruzioni del programma come se fossero locazioni di memoria. Tipicamente i trasferimenti di informazioni avvengono tra i registri di I/O e i registri del processore. La Figura 3.2 illustra come tastiera e schermo siano connessi al processore dal punto di vista del software.

3.1.2 I/O controllati da programma

Iniziamo la discussione delle questioni riguardanti l'I/O osservando due dispositivi essenziali per l'interazione uomo-macchina: tastiera e schermo. Consideriamo il compito di acquisire caratteri digitati su una tastiera, immagazzinare tali dati in memoria e visualizzare gli stessi caratteri su uno schermo.

Il modo più semplice di organizzare tale programma consiste nel ricorrere alla tecnica di I/O nota come *controllo di programma* (*program controlled I/O*). Essa è adatta principalmente a unità di I/O lente come la tastiera, che non può fornire più di qualche carattere al secondo. Il video sarebbe anche in grado di visualizzare molti più caratteri per unità di tempo, tipicamente fino a qualche migliaio al secondo, ma entrambe le unità di I/O sono, comunque, enormemente più lente del processore, che può eseguire milioni o anche miliardi di istruzioni macchina al secondo. Un divario tanto ampio tra velocità di processore e unità di I/O obbliga a stabilire un meccanismo di sincronizzazione tra questo e quelle. Ecco una soluzione praticabile:

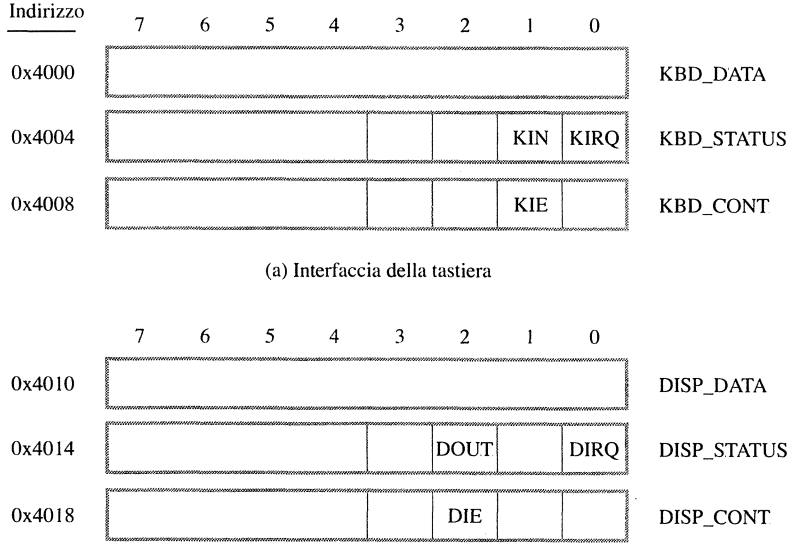
- il processore attende dalla tastiera la notifica dell'avvenuta pressione del tasto, il cui codice viene automaticamente introdotto in un registro buffer apposito facente parte della tastiera stessa; quando ciò è avvenuto il processore prende il carattere. La notifica avviene attivando un segnale apposito; la procedura si ripete identica per ogni carattere da acquisire;
- il processore manda il carattere al video e attende che questo gli notifichi di averlo ricevuto, attivando un segnale apposito; la procedura si ripete identica per ogni carattere da visualizzare.

Tastiera e video sono due unità di I/O separate. La pressione del tasto da sola non comporta affatto la visualizzazione del carattere. È il processore che trasferisce il carattere tra le due unità: il programma contiene un blocco di istruzioni per leggere caratteri da tastiera e un altro blocco di istruzioni per scriverli a video.

Si consideri ora il primo passo, cioè come trasferire il carattere da tastiera a processore. La pressione del tasto forza l'introduzione del codice di carattere a 8 bit nel registro buffer, pure a 8 bit, della tastiera; l'operazione è svolta direttamente dalla tastiera, è automatica e avviene perché la tastiera è costituita in tale modo e non potrebbe accadere diversamente. Si supponga che il registro buffer abbia nome KBD_DATA, per dato dalla tastiera (*keyboard*), come mostra la Figura 3.3. Per notificare al processore che nel registro KBD_DATA c'è un carattere disponibile, la tastiera porta a 1 il bit di stato KIN, che fa parte di un registro di stato a otto bit, KBD_STATUS; come prima, l'operazione avviene in modo automatico. Il programma *esamina ciclicamente* o *monitora* il bit KIN e quando lo trova a 1 trasferisce in memoria o in un suo registro il contenuto di KBD_DATA. Di solito la tastiera è sensibile all'operazione di lettura e quando si accorge che è avvenuta azzera automaticamente il bit KIN, sempre in modo automatico. Non appena viene premuto un altro tasto (o ripremuto lo stesso) la serie di eventi così descritta si ripete.

Il secondo passo, cioè come trasferire il carattere da memoria a video, avviene in modo simile. Il video contiene il registro buffer chiamato DISP_DATA, per dato al video (*display*), a 8 bit, e il bit di stato D_OUT, che è un bit in un registro di stato, DISP_STATUS.

La Figura 3.3 mostra come possono essere organizzati questi registri. L'interfaccia di ogni dispositivo include pure un registro di controllo, di cui si discuterà nel Paragrafo 3.2. Sono stati identificati solo alcuni bit nei registri, quelli pertinenti alla discussione in questo capitolo. Altri bit possono essere usati per altri scopi, o forse semplicemente ignorati.



(a) Interfaccia della tastiera

(b) Interfaccia dello schermo

Figura 3.3 Registri nelle interfacce di tastiera e schermo.

Se si deve accedere ai registri nelle interfacce di I/O come se fossero locazioni di memoria, a ogni registro deve essere assegnato un indirizzo specifico che sarà riconosciuto dal circuito di interfaccia. In Figura 3.3 sono stati assegnati i numeri esadecimali 4000 e 4010 come indirizzi di base per tastiera e schermo, rispettivamente. Questi sono gli indirizzi dei registri per i dati. Gli indirizzi dei registri di stato sono quattro byte più in alto, mentre i registri di controllo sono otto byte più in alto. Questo rende tutti gli indirizzi allineati alla parola in un calcolatore avente una parola di 32 bit, che è ciò che avviene in pratica. Assegnare in questo modo gli indirizzi ai registri rende i registri di I/O accessibili da un programma eseguito dal processore. Questo è il punto di vista del programmatore del dispositivo.

È necessario un programma per effettuare la lettura di caratteri digitati dalla tastiera, immagazzinare questi caratteri in memoria e mandarli allo schermo. Per effettuare trasferimenti di I/O, il processore deve eseguire istruzioni macchina che controllino le condizioni di stato e che trasferiscano dati tra il processore e i dispositivi di I/O.

Si considerino i dettagli del processo di ingresso. Quando un tasto viene premuto, il circuito della tastiera inserisce il carattere codificato in ASCII nel registro KBD_DATA. Allo stesso tempo, il circuito pone la condizione di stato KIN a 1. Nel frattempo, il processore sta eseguendo il programma di I/O che controlla continuamente la condizione di stato KIN. Quando scopre che KIN è stato posto a 1, trasferisce il contenuto di KBD_DATA in un registro del processore. Una volta che il contenuto di KBD_DATA è stato letto, KIN deve essere posto a 0, il che in genere viene fatto automaticamente dal circuito di interfaccia. Se un secondo carattere viene digitato tramite la tastiera, KIN viene nuovamente posto a 1 e il processo si ripete. L'azione desiderata può essere ottenuta effettuando le operazioni:

```
ATTESALETTURA   Legge la condizione di stato KIN
                Salta a ATTESALETTURA se KIN = 0
                Trasferisce i dati da KBD_DATA a R5
```

che trasferiscono il carattere nel registro del processore R5.

Il seguente processo di operazioni, molto simile a quello visto sopra, guida il trasferimento del carattere al video:

```
ATTESASCRITTURA Legge la condizione di stato DOUT
                Salta a ATTESASCRITTURA se DOUT = 0
                Trasferisce i dati da R5 a DISP_DATA
```

Il ciclo di attesa viene eseguito di continuo finché il bit di stato DOUT non viene posto a 1 da parte del video, quando esso è pronto a visualizzare un nuovo carattere. L'operazione di trasferimento porta il carattere dal registro R5 di processore al registro buffer DISP_DATA, prendendolo dal quale il video lo visualizzerà; inoltre l'operazione stessa di scrittura in DISP_DATA dà al video il segnale di azzerare il bit di stato.

Si supponga che all'inizio i bit di stato KIN e DOUT siano posti a 0 e a 1, rispettivamente. Normalmente il bit di stato dell'unità di I/O viene impostato al valore iniziale da parte del sistema operativo, nel momento in cui l'unità viene collegata fisicamente al calcolatore o comunque agganciata logicamente al sistema; spesso ciò avviene al momento dell'attivazione del sistema operativo (*bootstrap*).

Nei calcolatori che usano l'unificazione degli spazi di indirizzamento di memoria e I/O, per cui alcuni indirizzi sono usati come riferimento ai registri nelle interfacce di I/O, i dati possono essere trasferiti tra questi registri e il processore usando istruzioni come Load, Store e Move. Per esempio, il contenuto del buffer di caratteri KBD_DATA può essere trasferito al registro R5 nel processore usando l'istruzione:

```
LoadByte   R5, KBD_DATA
```

Similmente, il contenuto del registro R5 può essere trasferito a DISP_DATA usando l'istruzione:

```
StoreByte   R5, DISP_DATA
```

I codici operativi LoadByte e StoreByte indicano che la dimensione dell'operando è un byte, per distinguerli dai codici operativi Load e Store utilizzati in precedenza per operandi di dimensioni pari a una parola.

L'operazione di lettura descritta sopra può essere realizzata dalle seguenti istruzioni in stile RISC:

```
ATTESALETTURA:   LoadByte          R4, KBD_STATUS
                 And               R4, R4, #2
                 Branch_if_[R4]=0  ATTESALETTURA
                 LoadByte          R5, KBD_DATA
```

L'istruzione And è usata per valutare la condizione di stato KIN, che è il bit b_1 del registro R4 che era stato letto dal registro KBD_STATUS. Finché $b_1 = 0$, il risultato dell'operazione AND lascia il valore in R4 uguale a zero, e il ciclo ATTESALETTURA continua a essere eseguito.

Analogamente, l'operazione di scrittura può essere effettuata per mezzo di:

```
ATTESASCRITTURA: LoadByte          R4, DISP_STATUS
                 And               R4, R4, #4
                 Branch_if_[R4]=0  ATTESASCRITTURA
                 StoreByte         R5, DISP_DATA
```

Si osservi che l'istruzione And in questo caso usa il valore immediato 4 per controllare il bit di stato b_2 dello schermo.

3.1.3 Un esempio di un programma per I/O in stile RISC

Adesso è possibile mettere insieme un programma completo per una tipica operazione di I/O, come mostrato in Figura 3.4. Il programma usa l'approccio suddetto di I/O controllato da programma per leggere, immagazzinare e visualizzare una linea di caratteri digitati da tastiera. A mano a mano che i caratteri sono letti, uno per uno, essi sono immagazzinati in memoria e quindi *riprodotti* sullo schermo. Il programma termina quando viene incontrato il carattere di ritorno carrello CR. L'indirizzo del primo byte della locazione di memoria a partire da cui la linea verrà memorizzata è LOC. Il registro R2 è usato per puntare a questa parte della memoria ed è inizialmente caricato con l'indirizzo LOC dalla prima istruzione del programma. R2 è incrementato per ciascun carattere letto e visualizzato.

3.1.4 Un esempio di un programma per I/O in stile CISC

Si effettua ora lo stesso compito usando istruzioni in stile CISC. Negli insiemi di istruzioni CISC è possibile effettuare alcune operazioni aritmetiche e logiche direttamente sugli operandi in memoria. Quindi è possibile avere l'istruzione:

```
TestBit   destinazione, #k
```

che controlla il bit b_k dell'operando di destinazione e pone a 1 il codice di condizione Z (Zero) se $b_k = 0$ altrimenti pone Z a 0. Poiché l'operando può essere in una locazione di memoria, si usa l'istruzione:

```
TestBit   KBD_STATUS, #1
```



```
        Move            R2, #LOC          Inizializza il registro puntatore R2 per
                                          puntare all'indirizzo della prima locazione
                                          nella memoria principale dove
                                          immagazzinare i caratteri
        MoveByte        R3, #CR           Carica in R3 il codice ASCII per il Ritorno
                                          Carrello
LEGGI:  LoadByte        R4, KBD_STATUS    Attendi l'immissione di un carattere
        And             R4, R4, #2        Controlla la condizione di stato KIN
        Branch_if_[R4]=0  LEGGI
        LoadByte        R5, KBD_DATA      Leggi il carattere da KBD_DATA (ciò azzera
                                          KIN)
        StoreByte       R5, (R2)          Scrivi il carattere nella memoria principale
                                          e incrementa il puntatore alla memoria
                                          principale
        Add             R2, R2, #1
ECO:    LoadByte        R4, DISP_STATUS   Attendi che lo schermo sia pronto
        And             R4, R4, #4        Controlla la condizione di stato DOUT
        Branch_if_[R4]=0  ECO
        StoreByte       R5, DISP_DATA     Trasferisci il carattere appena letto al registro
                                          buffer dello schermo (ciò azzera DOUT)
        Branch_if_[R5]≠[R3]  LEGGI        Controlla se il carattere appena letto sia
                                          il Ritorno carrello. Se non lo è, reitera la
                                          lettura di caratteri
```

Figura 3.4 Un programma in stile RISC che acquisisce una linea di caratteri e li visualizza.

per valutare la condizione di stato KIN nell'interfaccia della tastiera. Un'istruzione di salto che valuta la condizione di stato Z può essere quindi usata per effettuare un salto all'inizio del ciclo di attesa.

La Figura 3.5 fornisce un programma in stile CISC che legge e visualizza una linea di caratteri. Si osservi che la prima istruzione MoveByte trasferisce ciascun carattere direttamente da KBD_DATA alla locazione di memoria puntata da R2. L'istruzione:

```
Compare   destinazione, sorgente
```

effettua il confronto sottraendo il contenuto della sorgente dal contenuto della destinazione e quindi aggiorna i bit di esito nel registro della condizione di stato del processore in base al risultato. Tale istruzione non cambia il contenuto né di sorgente né di destinazione. Si noti che l'istruzione CompareByte nella Figura 3.5 usa il modo di indirizzamento con autoincremento, che incrementa automaticamente di 1 il valore del puntatore R2 dopo che il confronto è stato fatto. Nel programma in stile RISC in Figura 3.4 il puntatore deve essere incrementato usando un'istruzione Add separata.

Si è discusso lo schema di unificazione degli spazi di indirizzamento di memoria e I/O, che è usato nella maggior parte dei calcolatori. Un'alternativa può essere trovata in alcuni processori dove esistono istruzioni di ingresso e uscita speciali In e Out per effettuare trasferimenti di I/O. In questo caso, esiste uno spazio degli indirizzi separato usato solo da queste istruzioni. Quando si costruisce un calcolatore che usa questi processori, il progettista ha la possibilità di connettere dispositivi di I/O in modo da usare lo spazio degli indirizzi di I/O speciali, oppure può semplicemente incorporarli come parte dello spazio degli indirizzi di memoria.

	Move	R2, #BLOCCO	Inizializza il registro R2 per puntare all'indirizzo della prima locazione nella memoria principale dove immagazzinare i caratteri
LEGGI	TestBit	KBD_STATUS, #1	Monitorando la condizione di stato KIN, attendi l'immissione di un carattere nel registro di I/O KBD_DATA
	Branch=0	LEGGI	
	MoveByte	(R2), KBD_DATA	Scrivi nel byte di memoria puntato da R2 il carattere contenuto nel registro di I/O KBD_DATA (ciò azzera KIN)
ECO	TestBit	DISP_STATUS, #2	Attendi che lo schermo sia pronto monitorandone la condizione di stato DOUT
	Branch=0	ECO	
	MoveByte	DISP_DATA, (R2)	Scrivi il carattere puntato da R2 nel registro di I/O DISP_DATA (ciò azzera DOUT)
	CompareByte	(R2)+, #CR	Verifica se il carattere appena letto da tastiera sia il Ritorno Carrello: se non lo è, reitera la lettura di caratteri; in ogni caso incrementa il registro puntatore R2
	Branch≠0	LEGGI	

Figura 3.5 Un programma in stile CISC che acquisisce una linea di caratteri e li visualizza.

L'I/O controllato da programma richiede un continuo coinvolgimento del processore nelle attività di I/O. Quasi tutto il tempo di esecuzione per i programmi nelle Figure 3.4 e 3.5 viene trascorso nei due cicli di attesa, mentre il processore aspetta che un tasto sia premuto o che lo schermo diventi disponibile. Lo spreco di tempo di esecuzione del processore in questo modo può essere evitato usando il concetto di interruzione.

3.2 Tecnica di interruzione

I programmi di esempio mostrati nelle Figure 3.4 e 3.5 consistono in un ciclo di attesa perenne (quanto meno in linea di principio), durante cui si esamina ripetutamente lo stato della periferica. Quando il processore si trova nel ciclo non svolge altro lavoro utile, ma ci sono parecchie situazioni dove si potrebbero espletare numerose altre attività necessarie mentre si attende che una periferica data si renda disponibile. Per evitare di perdere tempo in attesa si può strutturare il sistema di I/O conferendo alla periferica la capacità di allertare il processore quando è il momento, cioè non appena essa è pronta per trasferire dati. Basta che la periferica possa mandare al processore un segnale apposito, chiamato segnale di *interruzione* (*interrupt*). Il bus di controllo contiene sempre almeno una linea dedicata a tale funzione, la linea di *richiesta di interruzione* INT_REQ (*interrupt request*). Siccome dunque il processore non ha più l'obbligo di esaminare in continuazione lo stato della periferica, nell'intervallo di tempo morto tra un'interruzione e quella successiva esso è libero di dedicarsi ad altri compiti utili. In effetti grazie al meccanismo di interruzione il tempo morto di attesa si riduce a zero.

Esempio 3.1

Si consideri un programma ciclico che debba fare qualche calcolo e visualizzarne il risultato. I risultati visualizzati devono essere aggiornati ogni dieci secondi. Gli intervalli di dieci secondi possono essere determinati dal semplice circuito di un cronometro, che genera un segnale appropriato.

Approfondimento applicativo 3.1

I/O nei processori NIOS II, ARM, ColdFire e IA-32

Si prosegue qui e nei successivi approfondimenti applicativi di questo capitolo l'analisi comparativa di caratteristiche delle quattro ISA considerate negli approfondimenti applicativi del Capitolo 2, ma con riferimento alle istruzioni di I/O. Per ulteriori dettagli su queste si rinvia alla bibliografia del Capitolo 2. Le caratteristiche generali dell'accesso ai dispositivi di I/O nei quattro processori sono conformi a quelle delineate nel Paragrafo 3.1: unificazione degli spazi di indirizzamento di memoria e I/O, trasferimenti controllati da programma o su interruzioni ecc. Negli esempi che seguono si incontrano alcune nuove istruzioni, non introdotte negli approfondimenti applicativi del Capitolo 2. Per il momento ci si limita a quelle qui usate negli esempi di controllo da programma di dispositivi di I/O, altre sono presentate nei successivi approfondimenti applicativi.

Se l'architettura di un processore include una memoria cache, allora i dati provenienti da registri di interfacce di I/O con unificazione degli spazi di indirizzamento non dovrebbero essere tenuti nella cache, poiché il contenuto di tali registri cambia sotto influenza esterna. Quindi i riferimenti a queste locazioni dovrebbero evitare la cache e accedere direttamente ai registri di I/O. Per esempio, l'ISA NIOS II mette a disposizione una variante speciale di ciascuna istruzione di Load o Store a tal fine, che ha la stessa forma di quella ordinaria eccetto che il codice mnemonico è esteso con il suffisso "io" a indicare che il riferimento a memoria riguarda un registro di un'interfaccia di I/O e che pertanto il trasferimento da o a tale locazione deve evitare la cache, se ne esiste una. Spesso si usano altri meccanismi per ottenere lo stesso effetto.

L'esempio di controllo da programma di I/O, mostrato nelle Figure 3.4 e 3.5 in linguaggio assemblativo generico, è tradotto nei programmi RISC NIOS II e ARM in Figura A3.1 e nei programmi CISC ColdFire e IA-32 in Figura A3.2. Si noti l'uso delle istruzioni ldbio e stbio nel programma NIOS II, varianti di I/O delle rispettive istruzioni di trasferimento di byte come si è detto. La notazione *&h*, con *h* una sequenza di cifre esadecimali, impiegata nel programma ARM è equivalente alla notazione 0x*h* (anch'essa disponibile nel linguaggio assemblativo ARM). Si noti anche l'uso di pseudoistruzioni per l'inizializzazione dei registri R0, R1 e R2 con indirizzi definiti nello stesso programma. Si veda l'Approfondimento applicativo 2.5 per questo e per le istruzioni TST e TEQ dell'ISA ARM. Per il resto i programmi in Figura A3.1 riflettono fedelmente la struttura del programma in Figura 3.4.

Similmente, i programmi in Figura A3.2 (nei quali si sono omesse le direttive di definizione di simboli) riflettono la struttura del programma in Figura 3.5, con le seguenti peculiarità.

Nel programma ColdFire risulta necessario caricare gli indirizzi dei registri di stato delle interfacce di I/O in registri di indirizzi, sebbene tali puntatori siano costanti, perché i vincoli sui modi di indirizzamento nell'ISA ColdFire vietano l'uso dell'indirizzamento assoluto per la destinazione nell'istruzione BTST.B con operando sorgente immediato. Questa istruzione (in questo caso con destinazione a 8 bit, come indicato dal suffisso del codice mnemonico) corrisponde all'istruzione assemblativa generica TestBit, descritta nel Paragrafo 3.1.4. Inoltre, nel programma ColdFire occorre preventivamente azzerare il registro dati D0, nel quale viene trasferito il byte letto dalla tastiera, perché l'istruzione di confronto, qui usata per controllare se il byte sia il codice ASCII CR, è disponibile solo per operandi a 32 bit, ovvero con suffisso L del suo codice mnemonico. Ciò non è necessario nel programma IA-32, poiché il registro AL è a 8 bit. L'istruzione IA-32 BT svolge un compito simile a quello dell'istruzione generica TestBit, ma pone il risultato nel bit di esito C invece che Z, il che spiega la successiva istruzione di salto condizionato JNC. Si noti anche che l'incremento del puntatore alla locazione di me-

	NIOS II		ARM	Commenti
	.equ KBD_DATA, 0x4000	KBD_DATA	EQU &4000	Indirizzi dei registri dati di tastiera e schermo
	.equ DISP_DATA, 0x4010	DISP_DATA	EQU &4010	
	movia r2, LOC		LDR R0, =LOC	Locazione dove sarà immagazzinata la linea
	movia r3, KBD_DATA		LDR R1, =KBD_DATA	Puntatore al registro dati della tastiera
	movia r4, DISP_DATA		LDR R2, =DISP_DATA	Puntatore al registro dati dello schermo
	addi r5, r0, 0x0D	CR	EQU &0D	Codice ASCII di Ritorno Carrello (CR)
LEGGI:	ldbio r6, 4(r3)	LEGGI	LDRB R3, [R1, #4]	Leggi il registro di stato della tastiera
	andi r6, r6, 2		TST R3, #2	Controlla se c'è un carattere
	beq r6, r0, LEGGI		BEQ LEGGI	Attendi finché arriva
	ldbio r7, (r3)		LDRB R3, [R1]	Leggi il carattere dalla tastiera (ciò azzera KIN)
	stb r7, (r2)		STRB R3, [R0], #1	Immagazzina il carattere in memoria
	addi r2, r2, 1			Incrementa il puntatore
ECO:	ldbio r6, 4(r4)	ECO	LDRB R4, [R2, #4]	Leggi il registro di stato dello schermo
	andi r6, r6, 4		TST R4, #4	Controlla se lo schermo è pronto
	beq r6, r0, ECO		BEQ ECO	Attendi finché non lo è
	stbio r7, (r4)		STRB R3, [R2]	Invia il carattere allo schermo (ciò azzera DOUT)
	bne r5, r7, LEGGI		TEQ R3, #CR BNE LEGGI	Se il carattere non è CR reitera la lettura di caratteri

Figura A3.1 Programmi NIOS II e ARM di lettura e visualizzazione di una linea di caratteri.

	ColdFire		IA-32	Commenti
	MOVEA.L #LOC, A2		LEA EBX, LOC	Inizializza il puntatore alla prima locazione di memoria dove immagazzinare i caratteri
	MOVEA.L #KBD_STATUS, A3			Inizializza il puntatore all'indirizzo del registro di stato della tastiera
	MOVEA.L #DISP_STATUS, A4			Inizializza il puntatore all'indirizzo del registro di stato dello schermo
	CLR.L D0			Azzera il registro dati destinato ai caratteri
LEGGI:	BTST.B #1, (A3)	LEGGI:	BT KBD_STATUS, 1	Attendi la lettura di un carattere nel buffer KBD_DATA della tastiera
	BEQ LEGGI		JNC LEGGI	
	MOVE.B KBD_DATA, D0		MOV AL, KBD_DATA	Trasferisci il carattere al registro (ciò azzera KIN)
	MOVE.B D0, (A2)+		MOV [EBX], AL INC EBX	Immagazzina il carattere in memoria e incrementa il puntatore
ECO:	BTST.B #2, (A4)	ECO:	BT DISP_STATUS, 2	Attendi che lo schermo sia pronto
	BEQ ECO		JNC ECO	
	MOVE.B D0, DISP_DATA		MOV DISP_DATA, AL	Trasferisci il carattere appena letto al buffer dello schermo (ciò azzera DOUT)
	CMPI.L #CR, D0		CMP AL, CR	Controlla se il carattere letto sia CR
	BNE LEGGI		JNE LEGGI	Se non lo è, reitera la lettura dei caratteri

Figura A3.2 Programmi ColdFire e IA-32 di lettura e visualizzazione di una linea di caratteri.

moria destinata al byte successivo necessita di un'istruzione apposita nel programma IA-32, al contrario che nel programma ColdFire, dove è realizzato dall'uso del modo di indirizzamento con autoincremento.

Il processore tratta il circuito del cronometro come un dispositivo di ingresso producente un segnale che può essere rilevato. Se questo viene fatto attraverso il controllo da programma, il processore sprecherà un tempo notevole per controllare lo stato del segnale. Una soluzione migliore è avere un cronometro che segnali una richiesta di interruzione ogni dieci secondi. In risposta, il processore visualizza i risultati più recenti. Il programma è costituito da due routine: CALCOLA, che fa il conto, e VISUALIZZA, che ne manda al video il risultato.

Quando il processore riceve una richiesta di interruzione dal cronometro, esso sospende l'esecuzione della routine CALCOLA ed esegue la routine VISUALIZZA, che manda i risultati più recenti

allo schermo. Al completamento della routine VISUALIZZA, il processore riprende l'esecuzione della routine CALCOLA. Poiché il tempo necessario a mandare i risultati allo schermo è molto piccolo se confrontato all'intervallo di dieci secondi, il processore in effetti passa quasi tutto il suo tempo eseguendo la routine CALCOLA.

L'esempio illustra dunque il concetto di interruzione. Il sottoprogramma attivato in risposta alla richiesta di interruzione è chiamato *routine di servizio di interruzione* o ISR (*Interrupt Service Routine*); nell'esempio è la routine VISUALIZZA. Come ben si intuisce il meccanismo di interruzione ha forte analogia con quello di chiamata a sottoprogramma, e la Figura 3.6 illustra la somiglianza. Si supponga che la richiesta di interruzione pervenga al processore mentre sta eseguendo l'istruzione macchina i^{esima}. Come prima cosa il processore deve comunque terminare l'esecuzione dell'istruzione i per non lasciare a metà le operazioni sui dati (ciò li renderebbe privi di senso riconoscibile). Poi il processore scrive nel registro contatore di programma PC l'indirizzo dell'istruzione iniziale della routine di servizio di interruzione; per ora si supponga che tale indirizzo sia fisso. Quando la routine di servizio termina, il flusso di esecuzione va riportato all'istruzione $(i + 1)^{esima}$. Ne consegue che il contenuto di PC al momento dell'arrivo della richiesta di interruzione, vale a dire l'indirizzo $i + 1$, va temporaneamente salvato da qualche parte per poterlo recuperare in seguito. La routine di servizio si conclude con l'apposita istruzione macchina Return-from-interrupt, che va a recuperare l'indirizzo di rientro nella posizione dove era stato salvato e lo riscrive nel registro PC e così riporta il flusso di esecuzione all'istruzione $(i + 1)^{esima}$. In molti processori l'indirizzo di rientro viene salvato impilandolo, proprio come si fa con il normale sottoprogramma, ma in alternativa lo si potrebbe salvare in un registro speciale designato apposta per tale funzione.

È necessario che il processore informi la periferica interrompente circa il fatto che la richiesta è stata recepita, cosicché la periferica possa disattivare il segnale di richiesta. Spesso il bus di controllo contiene una linea apposita, chiamata *conferma di interruzione* o INT_ACK (*INTerrupt ACKnowledge*), che serve appunto a tale scopo. Ancorché il segnale INT_ACK sia molto usato, non è tuttavia indispensabile, giacché la periferica (o meglio l'interfaccia di I/O) si può accorgere dell'avvenuta partenza del servizio di interruzione semplicemente osservando che il processore si dispone a trasferire il dato; anche questa seconda soluzione è piuttosto frequente. Infatti, se mentre la periferica tiene attiva le linea di richiesta il processore

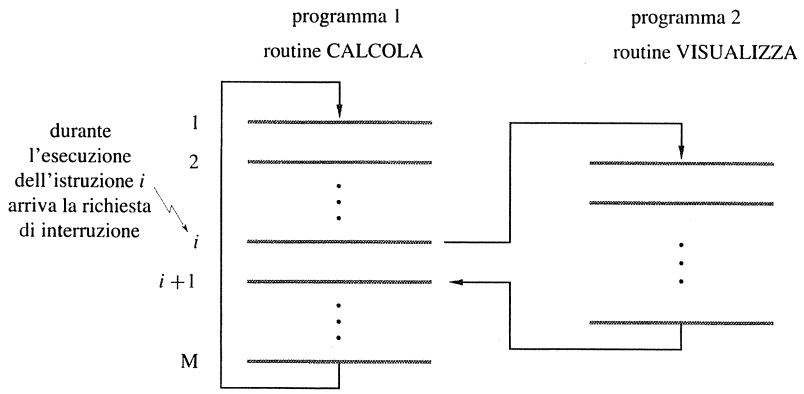


Figura 3.6 Trasferimento di flusso di esecuzione per servizio di interruzione.

esegue un'istruzione macchina di I/O che interessa uno dei registri di dato o di stato dell'interfaccia della periferica, allora l'interfaccia stessa, così come anche la periferica, può concludere che la richiesta è stata recepita e che dunque il servizio è ormai in corso.

Quanto detto finora non ha fatto che rafforzare l'analogia tra interruzione e chiamata a sottoprogramma, ma ci sono anche differenze. Per cominciare, lo stimolo a eseguire una routine normale viene dal programma chiamante stesso, non dall'esterno; pertanto la routine chiamata è nota a priori al programma chiamante, che dunque sa quale genere di compito vada a svolgere. Al contrario la routine di servizio di interruzione potrebbe essere del tutto estranea alla logica del programma interrotto. Non di rado, infatti, programma interrotto e routine di servizio non appartengono neppure allo stesso utente. Ne consegue che prima di passare a svolgere i propri calcoli la routine di servizio dovrebbe avere cura di preservare ogni elemento di informazione che potrebbe modificare, per ripristinarlo in seguito subito prima di rientrare al programma interrotto. Solo così il programma interrotto potrà ripartire senza alterazioni, tranne che per l'inevitabile ritardo. Gli elementi di informazione più sensibili, che quasi sempre vanno preservati, sono quelli contenuti nel processore, ovvero bit di esito e contenuto dei registri in uso da parte sia del programma interrotto sia della routine di servizio ed eventualmente anche parole di memoria condivise.

Il compito di preservare e ripristinare informazioni sensibili può essere espletato in modo automatico da parte del processore oppure diretto da istruzioni macchina della routine di servizio. La maggior parte dei processori odierni provvede a preservare automaticamente solo una quantità minima di informazione, assolutamente vitale per mantenere l'integrità del programma interrotto. Ciò dipende dal fatto che preservare informazioni comporta uso di memoria, allunga il tempo di esecuzione e pertanto rappresenta un costo aggiuntivo. Inoltre, se si devono preservare numerosi registri l'intervallo di tempo intercorrente tra arrivo della richiesta di interruzione e partenza effettiva del servizio si allunga. Tale intervallo si chiama *tempo di latenza di interruzione* (*interrupt time latency*); per applicazioni in *tempo reale* (*real time*) latenze di interruzione elevate sono inaccettabili. Ne consegue che quando l'interruzione viene servita di norma il processore salva automaticamente solo il contenuto del registro contatore di programma PC e del registro di stato del programma PS (*Program Status*), nient'altro. Se occorre preservare altri elementi di informazione ci devono pensare le istruzioni iniziali della routine di servizio, mentre quelle finali devono provvedere al rispettivo ripristino.

Taluni processori non recenti ma tuttora diffusi, in particolare quelli contenenti pochi registri, provvedono a salvare automaticamente l'intero banco di registri, speciali e di uso generale, non appena si sono disposti a servire l'interruzione. L'istruzione di rientro da interruzione provvede poi direttamente a ripristinare l'intero banco. Altri processori dispongono invece di due tipologie di interruzione: una prevede il meccanismo automatico completo di salvataggio, l'altra solo la versione minimale. La periferica può scegliere se mandare l'uno o l'altro tipo di richiesta secondo che si aspetti reazione immediata o differita. Un modo alternativo di agire consiste nel dotare il processore di un banco ausiliario di registri, indentico a quello principale e normalmente inattivo; però la routine di servizio funziona usando il banco ausiliario senza intaccare il contenuto di quello principale.

Il meccanismo di interruzione rappresenta ben più di una mera tecnica di coordinamento tra operazioni di I/O, benché questo ne sia l'uso più facile da immaginare. In senso generale il meccanismo di interruzione consente di trasferire il controllo dell'esecuzione da un programma a un altro per impulso di eventi esterni al calcolatore e di restituire il controllo al programma interrotto non appena quello interrompente si sia completato. Il meccanismo di interruzione gioca un ruolo essenziale in due ambiti caratteristici: nel sistema operativo, dove regola e media i rapporti tra processi e sistema operativo stesso, e anche in molti sistemi di controllo, dove occorre sincronizzare con precisione l'attivazione di routine specifiche quando si verificano determinati eventi esterni; quest'ultimo insieme di applicazioni va sotto il nome di *elaborazione in tempo reale* (*real time processing*).

3.2.1 Controllo di interruzione

Il calcolatore deve disporre di funzioni per dare al programmatore controllo completo sugli eventi che si possono verificare durante l'esecuzione del programma. L'arrivo di richiesta di interruzione generata da periferica forza il processore a sospendere l'esecuzione del programma corrente per passare a eseguirne un altro. Siccome l'interruzione è un evento impredicibile (giacché dipende da circostanze ignote al programmatore) è suscettibile di alterare la serie di eventi che, secondo le intenzioni del programmatore, dovrebbe avere luogo eseguendo il programma. Pertanto il processo di interruzione del programma corrente deve essere controllabile con precisione. Lo strumento principale per ottenere tale risultato e che si trova in qualsiasi calcolatore consiste nella facoltà di abilitare o disabilitare il meccanismo di interruzione secondo la volontà del programmatore. Qui di seguito si mostrerà come si può procedere.

Esistono parecchie situazioni in cui il processore non deve prestare ascolto alla richiesta di interruzione. Per esempio, il circuito del cronometro nell'Esempio 3.1 dovrebbe segnalare una richiesta di interruzione solo quando la routine CALCOLA è in esecuzione. Quando viene eseguito qualche altro compito, dovrebbe essere evitato che il cronometro segnali tale richiesta. In altri casi può essere necessario che una sequenza data di istruzioni macchina venga eseguita con continuità perché la routine di servizio di interruzione altererebbe parte dei dati elaborati dalle istruzioni in questione.[2] Sono queste le ragioni per cui il programmatore deve disporre di mezzi per attivare o disattivare il meccanismo di interruzione, e il modo più semplice di farlo è quello di prevedere istruzioni macchina per abilitarlo e disabilitarlo, rispettivamente.

È conveniente essere in grado di abilitare e disabilitare le interruzioni a entrambi gli estremi, processore e dispositivi di I/O. Il processore può sia accettare sia ignorare richieste di interruzioni. Un dispositivo di I/O può sia essere abilitato a segnalare richieste di interruzioni sia essere inibito a farlo. Un meccanismo comunemente usato per ottenere ciò consiste nell'avere alcuni bit di controllo nei registri, accessibili da istruzioni di programma.

Il registro di stato del programma, PS, contiene informazioni sullo stato corrente delle sue operazioni. Si supponga che un bit, IE, di questo registro sia assegnato per abilitare/disabilitare le interruzioni. Quindi, il programmatore può asserire o azzerare IE per ottenere l'azione desiderata. Quando IE = 1, le richieste di interruzione dai dispositivi di I/O sono accettate e servite dal processore. Quando IE = 0, il processore semplicemente ignora tutte le richieste dai dispositivi di I/O.

L'interfaccia di un dispositivo di I/O include un registro di controllo che contiene le informazioni che governano la modalità di funzionamento del dispositivo. Un bit in questo registro può essere dedicato al controllo delle interruzioni. Il dispositivo di I/O è abilitato a segnalare richieste di interruzioni solo quando questo bit è 1. Si discuterà questa organizzazione nel Paragrafo 3.2.3.

Si prenda come esempio il caso molto particolare di una sola richiesta di interruzione in un calcolatore cui sia collegata una sola periferica. Quando la periferica decide di lanciare la richiesta di interruzione, attiva la linea di richiesta e la tiene attivata finché non si accorge che il processore l'ha accettata e ha fatto partire la routine di servizio di interruzione. Ciò significa che la linea resta attivata anche durante l'esecuzione della routine di servizio o almeno durante la prima parte di essa, per esempio finché non venga eseguita un'istruzione macchina di ingresso o uscita per trasferire un dato con la periferica stessa, dandole così modo di accorgersi che il servizio è partito. È essenziale che lo stato della linea, rimasta attiva, non venga interpretato dal processore come seconda richiesta di interruzione e così via, dando luogo a una cascata di richieste tutte quante fasulle tranne la prima.

[2] La sequenza di istruzioni macchina non interrompibile si dice *atomica*, cioè indivisibile.

Una soluzione consiste nel disabilitare automaticamente il meccanismo di interruzione prima di fare partire la routine di servizio. Dopo avere impilato il contenuto dei due registri contatore di programma PC e di stato PS il processore disabilita il meccanismo, come se virtualmente avesse eseguito l'istruzione di disabilitazione. Spesso, come si è detto, il registro PS contiene un bit apposito, il bit di *abilitazione di interruzione* o IE (*Interrupt Enable*), che indica se il meccanismo è in funzione. Se tale bit ha valore 1 la richiesta in arrivo viene accettata e servita, o come si dice viene *onorata* subito; altrimenti viene ignorata. Dopo avere impilato i registri PC e PS, durante la quale operazione il bit IE di abilitazione vale ancora 1, il processore lo azzera e in tale modo disabilita il meccanismo ignorando richieste future. Rientrando al programma interrotto i valori di PC e PS prima impilati vengono spilati e riscritti nei rispettivi registri, così si ripristina anche il valore precedente del bit di abilitazione, che era 1; l'effetto complessivo è quello di riabilitare il meccanismo di interruzione (si noti come qui, sia l'istruzione di rientro stessa quella che riabilita).

Prima di esaminare gli aspetti più complessi dell'interruzione è bene riassumere la sequenza di eventi che devono avere luogo per gestire correttamente la richiesta proveniente da una sola periferica. Se il meccanismo di interruzione è abilitato sia nel processore, sia nella periferica, ecco che cosa deve succedere:

1. la periferica attiva la linea di richiesta di interruzione;
2. il processore sospende l'esecuzione del programma corrente e salva il contenuto dei registri PC e PS;
3. il meccanismo di interruzione viene disabilitato commutando il bit IE di abilitazione contenuto nel registro di stato PS;
4. la routine di servizio di interruzione va in esecuzione e svolge il compito richiesto dalla periferica; durante questo tempo il dispositivo viene informato che la sua richiesta è stata riconosciuta e, in risposta, disattiva il segnale di richiesta di interruzione;
5. al completamento della routine di servizio dell'interruzione, i contenuti dei registri PC e PS precedentemente salvati vengono ripristinati (e le interruzioni abilitate ponendo il bit IE a 1), quindi si rientra al programma interrotto riprendendone l'esecuzione.

3.2.2 Gestione di dispositivi multipli

Ora bisogna considerare situazioni dove ci siano diverse periferiche in grado di mandare al processore richieste di interruzione. Siccome le periferiche sono indipendenti non c'è alcun modo per prevedere o imporre ordine nell'arrivo delle richieste. Per esempio, la periferica P_2 potrebbe richiedere l'interruzione mentre è in corso il servizio di quella già richiesta prima dalla periferica P_1, oppure le due periferiche potrebbero mandare simultaneamente al processore ciascuna la rispettiva richiesta. Tutto ciò dà luogo a diversi interrogativi.

1. Come fa il processore a identificare la periferica che gli sta richiedendo l'interruzione?
2. Dato che ciascuna periferica richiede in generale un tipo di servizio diverso da quelli delle altre, come fa il processore a ottenere l'indirizzo iniziale della routine di servizio specifica per la periferica la cui richiesta di interruzione deve servire?
3. È lecito che una periferica interrompa il processore mentre esso sta servendo una richiesta di interruzione precedente?
4. Come si deve comportare il processore quando gli arrivano due richieste di interruzione simultanee?

Il modo di affrontare e risolvere questi problemi varia da un calcolatore all'altro ed è importante valutare la soluzione scelta perché influisce sensibilmente sull'applicazione.

Se la richiesta di interruzione raggiunge il processore viaggiando su una sola linea del bus di controllo, occorrono informazioni aggiuntive per identificare la specifica periferica interrompente. Inoltre, qualora due periferiche diverse abbiano attivata la linea di richiesta simultaneamente bisogna scegliere quale delle due servire per prima. Solo quando il servizio della prima sarà stato completato sarà possibile passare a servire la seconda.

Le informazioni sufficienti per stabilire quale periferica stia richiedendo l'interruzione si trovano nel registro di stato della periferica stessa. Quando essa manda la richiesta imposta a 1 un bit del suo registro di stato chiamato bit di IRQ. Il modo più semplice per identificare la periferica interrompente è che la routine di servizio vada a scandire i registri di stato delle periferiche collegate al bus; la periferica che presenta il bit IRQ attivato è quella da servire chiamando la routine appropriata. Tale modo di procedere si chiama *scansione* (*polling*).

Lo schema di scansione delle periferiche è facile da programmare, ma ha lo svantaggio di consumare tempo nell'esame di periferiche che potrebbero non avere affatto richiesto alcun servizio. L'alternativa consiste nell'interruzione vettorizzata, che ora verrà descritta.

Interruzione vettorizzata

Per ridurre il tempo consumato dal processo di scansione si può fare in modo che la periferica interrompente si indentifichi autonomamente nei confronti del processore, cosicché esso possa passare subito a eseguire la routine di servizio corrispondente. Tale soluzione va sotto il nome di *interruzione vettorizzata* (*vectored interrupt*).

La periferica interrompente può rendere se stessa nota al processore mandandogli tramite il bus un *codice univoco di identificazione*. Così il processore riesce a riconoscere la periferica interrompente anche se la linea di richiesta è una sola e condivisa tra tutte le periferiche. Il codice identificativo mandato al processore da parte della periferica è un numero binario e potrebbe rappresentare l'indirizzo iniziale della routine di servizio appropriata per la periferica. Di solito però il codice è costituito da pochi bit, da 4 a 8, e da solo non basta a specificare l'indirizzo. Il processore trasforma in indirizzo il codice in vari modi, per esempio il codice identificativo mandato dalla periferica interrompente viene interpretato dal processore come indice di una tabella in memoria chiamata *tabella dei vettori d'interruzione* (*interrupt vector table*), ciascuna posizione della quale contiene l'indirizzo iniziale effettivo della routine di servizio associata all'indice, ovvero al codice identificativo, in definitiva alla periferica. Per esempio, 128 byte possono essere allocati per tenere una tabella di 32 vettori di interruzioni. Generalmente, la tabella dei vettori di interruzioni risiede nell'intervallo degli indirizzi più basso. Le routine di servizio delle interruzioni possono essere collocate ovunque in memoria. L'indirizzo contenuto in tabella è il cosiddetto *vettore di interruzione* (*interrupt vector*): il processore lo legge dalla tabella e lo scrive nel registro contatore di programma PC, così passa a eseguire la routine di servizio.

Annidamento delle interruzioni

Nel Paragrafo 3.2.1 si è suggerito di disabilitare il meccanismo di interruzione mentre è in corso il servizio, per garantire che il processore non serva richieste multiple provenienti dalla stessa periferica. Si può adottare lo stesso accorgimento anche in presenza di due o più periferiche, onde garantire che il servizio per la prima venga portato a termine prima di prestare attenzione alla seconda. Siccome di solito la routine di servizio di interruzione è breve, il ritardo indotto dalla disabilitazione del meccanismo riesce accettabile per la maggior parte delle periferiche di tipo semplice.

Tuttavia per alcuni tipi di periferica attendere troppo a lungo prima di essere servita porta a situazioni di errore. Come esempio si consideri un calcolatore che tenga il conto del tempo per mezzo di un orologio in tempo reale, vale a dire tramite un'interruzione di orologio periodica generata e mandatagli da un dispositivo oscillatore. Il processore serve

la richiesta eseguendo una breve routine per aggiornare l'insieme di contatori che rappresentano ora, minuto, secondo ecc. Affinché il sistema funzioni correttamente è necessario che l'attesa prima del servizio sia molto breve a confronto del periodo dell'interruzione di orologio. Per soddisfare tale requisito, quando siano presenti anche altre periferiche capaci di generare interruzioni, potrebbe essere necessario passare a servire l'interruzione di orologio mentre è ancora in corso il servizio di un'interruzione di altro tipo, richiesta da un'altra periferica.

L'esempio suggerisce di attribuire alle periferiche livelli di priorità opportuni. Quando il processore ha in corso il servizio di una richiesta di interruzione a un determinato livello di priorità, dovrebbe accettare e passare a servire subito una seconda richiesta appena arrivata solo se questa avesse livello di priorità superiore.

Lo schema di priorità va strutturato a più livelli di urgenza, vale a dire che mentre è in corso il servizio certe nuove richieste vanno servite subito, ma altre no, secondo il livello di priorità assegnato alle varie periferiche. Ecco come si può fare: si assegna al processore un livello corrente, variabile in corso di esecuzione sotto controllo del programma; in pratica il livello corrente del processore va immaginato come quello del programma che esso sta eseguendo. Il processore accetta e passa a servire subito solo le richieste di interruzione provenienti da periferiche cui sia assegnato un livello maggiore di quello corrente. Non appena il processore accetta e passa a servire una data richiesta di interruzione, il livello corrente si alza parificandosi a quello della periferica. In questo modo si disabilitano tutte le richieste collocate a livello inferiore o uguale, mentre quelle provenienti da periferiche a livello superiore, se ve ne sono, sono ancora accettabili. Quando il servizio corrente termina e si rientra al servizio precedente, che era stato sospeso, il livello corrente si abbassa ritornando al valore di prima.

Di solito il livello di priorità corrente del processore è codificato in bit appositi contenuti nel registro di stato PS. Mentre questo schema è usato in alcuni processori, qui si userà uno schema più semplice in esempi successivi.

Infine, si dovrebbe notare che se sono permesse interruzioni annidate, allora ciascuna routine di servizio d'interruzione deve salvare in pila i contenuti salvati del contatore di programma e del registro di stato. Questo deve essere fatto prima che la routine di servizio dell'interruzione abiliti l'annidamento tramite l'asserzione a 1 del bit IE del registro di stato.

Richieste simultanee di interruzione

Ora bisogna considerare come comportarsi quando due o più richieste di interruzione arrivano contemporaneamente al processore; si tratta di un problema diverso da quello dell'annidamento. Il processore deve avere un criterio per decidere quale delle due richieste servire per prima.

Scandire il registro di stato delle varie periferiche interrompenti è la soluzione più semplice. Alle periferiche risulta così automaticamente associato il livello di priorità corrispondente all'ordine di scansione. Una seconda soluzione consiste invece nell'usare il meccanismo di interruzione vettorizzata, ma allora bisogna selezionare una sola periferica per volta permettendole di mandare il vettore al processore: questo viene fatto in hardware usando i circuiti di arbitraggio, di cui si discuterà nel Capitolo 7.

3.2.3 Controllo della richiesta

È importante garantire che solo le periferiche in uso da parte del programma corrente possano richiedere interruzione, mentre a quelle inattive ciò non va permesso sebbene possano prendere parte a operazioni di I/O di altro tipo. Occorre dunque perfezionare l'interfaccia di I/O della periferica per controllare se a questa si debba permettere o vietare di ricorrere all'interruzione.

Il controllo necessario è generalmente fornito sotto forma di un bit di *abilitazione di interruzione* nel circuito di interfaccia del dispositivo.

I dispositivi di I/O variano in complessità da semplici ad abbastanza complessi. Dispositivi semplici, come una tastiera, richiedono poco controllo. Dispositivi complessi possono avere varie possibili modalità di funzionamento, che devono essere controllate. Un approccio comunemente usato è quello di fornire un registro di controllo nell'interfaccia del dispositivo, che mantiene l'informazione necessaria a controllare il comportamento del dispositivo. Questo registro è accessibile come una locazione indirizzabile, esattamente come i registri di stato e di dati precedentemente discussi. Un bit nel registro serve da bit di abilitazione di interruzione, IE. Quando questo è asserito a 1 da un'istruzione che scrive sul registro di controllo, il dispositivo è posto nella modalità in cui gli è permesso di interrompere il processore quando è pronto per un trasferimento di I/O.

La Figura 3.3 mostra i registri che possono essere usati nelle interfacce di tastiera e schermo. Poiché questi dispositivi trasferiscono dati basati su caratteri, manipolando un carattere alla volta, risulta appropriato usare un registro di dati a otto bit. Si è assunto che i registri di stato e di controllo siano pure a otto bit. Solo uno o due bit in questi registri sono necessari per gestire i trasferimenti di I/O. I rimanenti bit possono essere usati per specificare altri aspetti del funzionamento del dispositivo, o ignorati se non necessari. Il registro di stato della tastiera include i bit KIN e KIRQ. Si è già discusso l'uso del bit KIN nel Paragrafo 3.1.2. Il bit KIRQ è asserito a 1 se una richiesta di interruzione è stata segnalata, ma non ancora servita. La tastiera può segnalare una richiesta di interruzione solo quando il bit di abilitazione di interruzione, KIE, nel suo registro di controllo è posto a 1. Quindi, quando entrambi i bit KIE e KIN sono uguali a 1, una richiesta di interruzione viene segnalata e il bit KIRQ è posto a 1. Analogamente, il bit DIRQ nel registro di stato dell'interfaccia dello schermo indica se sia stata segnalata una richiesta di interruzione. Il bit DIE nel registro di controllo di questa interfaccia è usato per abilitare le interruzioni. Si osservi che sono stati posti i bit KIN e KIE nella posizione 1, mentre i bit DOUT e DIE nella posizione 2. Questa è una scelta arbitraria che rende i seguenti programmi di esempio più facili da capire.

3.2.4 Registri di controllo del processore

Si è già discusso della necessità di avere un registro di stato nel processore. Per trattare le interruzioni è utile avere alcuni altri registri di controllo. La Figura 3.7 ne mostra un caso possibile, in cui si hanno quattro registri di controllo del processore. Il registro di stato PS include il bit di abilitazione delle interruzioni, IE, in aggiunta ad altre informazioni di stato. Si ricordi

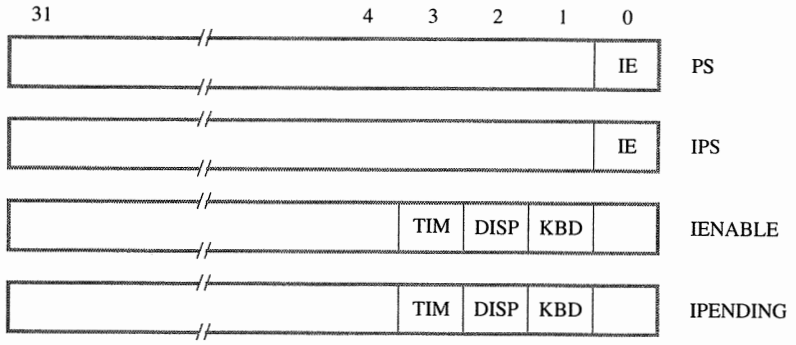


Figura 3.7 Registri di controllo nel processore.

che il processore accetterà interruzioni solo quando questo bit è posto a 1. Il registro IPS è usato per salvare automaticamente i contenuti di PS quando una richiesta di interruzione è ricevuta e accettata. Alla fine della routine di servizio delle interruzioni, il precedente stato del processore viene automaticamente ripristinato trasferendo il contenuto di IPS in PS. Poiché c'è un solo registro disponibile per memorizzare lo stato precedente, diventa necessario salvare in pila il contenuto di IPS se sono permesse interruzioni annidate.

Il registro IENABLE permette al processore di rispondere selettivamente a singoli dispositivi di I/O. Un bit può essere assegnato a ciascun dispositivo, come mostrato in figura per tastiera, schermo e per un circuito temporizzatore o *cronometro* (*timer*), che si userà in un esempio successivo. Quando un bit è posto a 1, il processore accetterà le richieste di interruzioni del dispositivo corrispondente. Il registro IPENDING indica le richieste di interruzioni attive. Questo è conveniente quando più dispositivi possono segnalare richieste allo stesso tempo. Quindi un programma può decidere quale interruzione dovrebbe essere servita per prima.

In un processore a 32 bit, i registri di controllo sono a 32 bit. Usando la struttura in Figura 3.7 è possibile gestire 32 dispositivi di I/O in maniera semplice.

Istruzioni del linguaggio assemblativo possono far riferimento ai registri di controllo del processore usando nomi come quelli mostrati in Figura 3.7. Tuttavia l'accesso a questi registri differisce da quello ai registri di uso generale. Essi non sono accessibili da istruzioni aritmetiche e logiche. Inoltre, essi non lo sono nemmeno da istruzioni Load e Store che usano il formato di codifica illustrato in Figura 2.32b, poiché in queste istruzioni si usa un campo a cinque bit per specificare un registro sorgente o destinazione, il che rende possibile specificare solo 32 registri di uso generale. Per accedere ai registri di controllo del processore si possono fornire istruzioni speciali o modi di indirizzamento speciali. In un processore in stile RISC, le istruzioni speciali possono essere del tipo:

MoveControl R2, PS

che carica il contenuto del registro di stato del programma nel registro R2, e:

MoveControl IENABLE, R3

che copia il contenuto di R3 nel registro IENABLE. Queste istruzioni effettuano trasferimenti tra i registri di controllo e quelli di uso generale.

3.2.5 Esempi di programmi con interruzioni

Dopo aver presentato gli aspetti di base delle interruzioni, è ora possibile fornire alcuni esempi illustrativi. Si useranno i dispositivi tastiera e schermo con la struttura di registri data in Figura 3.3.

Esempio 3.2

Si consideri nuovamente il compito di leggere una linea di caratteri digitata su una tastiera, di memorizzare i caratteri in memoria centrale e di mostrarli su uno schermo. Nelle Figure 3.4 e 3.5 si è mostrato come questo compito possa essere svolto usando l'approccio della scansione per rilevare quando i dispositivi di I/O sono pronti a trasferire i dati. Adesso si useranno le interruzioni per la tastiera ma la scansione per lo schermo.

Si assuma per adesso che una specifica locazione di memoria, ILOC, sia dedicata alla gestione delle interruzioni e che contenga la prima istruzione della routine di servizio delle interruzioni. Quando al processore arriva una richiesta di interruzione e le interruzioni del processore sono abilitate, automaticamente il processore:

- salverà il contenuto del contatore di programma, o in un registro del processore che mantiene l'indirizzo di rientro o nella pila del processore;

- salverà il contenuto del registro di stato PS trasferendolo nel registro IPS e azzererà il bit IE nel registro PS;
- caricherà l'indirizzo ILOC nel contatore del programma.

Si supponga che nel programma Main si desideri leggere una linea dalla tastiera e scrivere i caratteri in locazioni consecutive di byte di memoria, partendo dalla locazione LINEA. Inoltre, si supponga che la routine di servizio dell'interruzione sia stata caricata in memoria, a partire dalla locazione ILOC. Il programma Main deve inizializzare il processo delle interruzioni nel modo seguente:

1. carica l'indirizzo LINEA in una locazione di memoria, PNTR. La routine di servizio delle interruzioni userà questa locazione come un puntatore per immagazzinare in memoria i caratteri in ingresso;
2. abilita le interruzioni nell'interfaccia della tastiera ponendo a 1 il bit KIE nel registro KBD_CONT;
3. abilita il processore ad accettare interruzioni dalla tastiera ponendo a 1 il bit KBD nel suo registro di controllo IENABLE;
4. abilita il processore a rispondere alle interruzioni, in generale, ponendo a 1 il bit IE nel registro di stato del programma, PS.

Quando questa inizializzazione è stata completata, la digitazione di un carattere sulla tastiera causerà la generazione di una richiesta di interruzione da parte dell'interfaccia della tastiera. Il programma in esecuzione in quel momento sarà interrotto e verrà eseguita la routine di servizio dell'interruzione. Questa routine dovrà assolvere ai seguenti compiti:

1. leggere il carattere in ingresso dal registro dei dati in ingresso da tastiera; questo provocherà la rimozione della richiesta di interruzione da parte del circuito di interfaccia;
2. immagazzinare il carattere nella locazione di memoria puntata da PNTR e incrementare quest'ultimo;
3. visualizzare il carattere usando l'approccio della scansione;
4. quando si raggiunge la fine della linea, disabilitare le interruzioni da tastiera e informare il programma Main;
5. rientrare dall'interruzione.

Un programma in stile RISC che effettua questi compiti è mostrato in Figura 3.8. I commenti nel programma spiegano i dettagli rilevanti. Quando viene riconosciuta la fine della linea, la routine di servizio dell'interruzione azzera il bit KIE nel registro KBD_CONT, poiché non si aspettano altri dati in ingresso. Inoltre pone a 1 la variabile EOL di segnalazione di fine linea (*end of line*), che era stata inizialmente posta a 0. Si assuma che tale variabile sia controllata periodicamente dal programma Main per determinare quando la linea in ingresso è pronta per l'elaborazione. La variabile EOL realizza un modo di mettere in comunicazione il programma Main e la routine di servizio dell'interruzione.

Si osservi che le ultime tre istruzioni del programma Main sono usate per porre a 1 il bit di abilitazione di interruzioni in PS. Poiché solo istruzioni MoveControl possono accedere al contenuto di un registro di controllo, il contenuto di PS è caricato in un registro di uso generale, R2, modificato e quindi scritto nuovamente in PS. L'uso dell'istruzione Or per modificarne il contenuto agisce solo sul bit IE e lascia il resto dei bit in PS immutati.

Quando più dispositivi di I/O segnalano una richiesta di interruzione, è necessario determinare quale dispositivo abbia richiesto un'interruzione. Questo può essere risolto dal software controllando l'informazione nel registro di controllo IPENDING e scegliendo la routine di servizio dell'interruzione che dovrebbe essere eseguita.

Esempio 3.3

Nell'Esempio 3.2 sono state usate le interruzioni con la tastiera soltanto. Anche lo schermo può usare le interruzioni. Si supponga che un programma debba mostrare una pagina di testo immagazzinata in memoria. Questo può essere risolto facendo sì che il processore mandi un carattere quando l'interfaccia dello schermo è pronta, il che può essere indicato da una richiesta di interruzione. Si assuma che sia

Routine di servizio delle interruzioni

ILOC:	Subtract	SP, SP, #8	Salva i registri
	Store	R2, 4(SP)	
	Store	R3, (SP)	
	Load	R2, PNTR	Carica il puntatore all'indirizzo
	LoadByte	R3, KBD_DATA	Leggi un carattere dalla tastiera
	StoreByte	R3, (R2)	Scrivi il carattere in memoria
	Add	R2, R2, #1	Incrementa il puntatore
	Store	R2, PNTR	Aggiorna il puntatore in memoria
ECO:	LoadByte	R2, DISP_STATUS	Attendi che lo schermo sia pronto
	And	R2, R2, #4	
	Branch_if_[R2]=0	ECO	
	StoreByte	R3, DISP_DATA	Visualizza il carattere appena letto
	Move	R2, #CR	Codice ASCII per il Ritorno Carrello
	Branch_if_[R3]≠[R2]	RTRN	Rientra se non è CR
	Move	R2, #1	
	Store	R2, EOL	Indica la fine della linea
	Clear	R2	Disabilita le interruzioni nell'interfaccia della tastiera
	StoreByte	R2, KBD_CONT	
RTRN:	Load	R3, (SP)	Ripristina i registri
	Load	R2, 4(SP)	
	Add	SP, SP, #8	
	Return-from-interrupt		

Programma principale (Main)

INIZIO:	Move	R2, #LINEA	
	Store	R2, PNTR	Inizializza il puntatore del buffer
	Clear	R2	
	Store	R2, EOL	Azzera l'indicatore di fine linea
	Move	R2, #2	Abilita le interruzioni nell'interfaccia della tastiera
	StoreByte	R2, KBD_CONT	
	MoveControl	R2, IENABLE	
	Or	R2, R2, #2	Abilita le interruzioni da tastiera nel registro di controllo del processore
	MoveControl	IENABLE, R2	
	MoveControl	R2, PS	
	Or	R2, R2, #1	
	MoveControl	PS, R2	Poni a 1 il bit di abilitazione delle interruzioni in PS
	Prossima istruzione		

Figura 3.8 Un programma in stile RISC che acquisisce una linea di caratteri usando le interruzioni e li visualizza usando la scansione.

lo schermo sia la tastiera vengano usati da questo programma e che entrambi siano abilitati a segnalare richieste di interruzioni. Usando la struttura di registri mostrata nelle Figure 3.3 e 3.7, l'inizializzazione di interruzioni e l'elaborazione delle richieste possono essere fatte come indicato in Figura 3.9.

Il programma Main deve inizializzare ciascuna variabile necessaria alla routine di servizio delle interruzioni, quali i puntatori ai buffer di memoria, quindi abilita le interruzioni su entrambe le interfacce di tastiera e display. Successivamente, abilita le interruzioni nel registro di controllo IENABLE del processore. Si noti che il valore immediato 6, che è caricato in questo registro, pone a 1 i bit KBD e DISP. Infine, il processore viene abilitato a rispondere alle interruzioni in generale, ponendo a 1 il bit IE nel registro di stato del programma, PS.

Ancora una volta, si assuma che quando arriva una richiesta di interruzione il processore salvi automaticamente il contenuto del contatore di programma (PC), quindi carichi l'indirizzo ILOC in PC. Inoltre, salverà il contenuto del registro di stato (PS) trasferendolo nel registro IPS e disabiliterà le interruzioni. A differenza dell'Esempio 3.2, in cui si è assunto che ci fosse un solo dispositivo in grado di segnalare richieste di interruzioni, adesso non è possibile passare direttamente a eseguire la routine di servizio dell'interruzione desiderata. Per cominciare, è necessario identificare il dispositivo che ha generato l'interruzione. L'informazione utile si trova nel registro di controllo del processore IPENDING. Poiché la routine di servizio dell'interruzione usa i registri R2 e R3 in questo processo, il contenuto di questi registri deve essere salvato in pila e successivamente ripristinato. Inoltre, è necessario salvare il contenuto del registro di collegamento al sottoprogramma, LINK_reg, poiché un'interruzione potrebbe verificarsi mentre è in esecuzione qualche sottoprogramma, e la routine di servizio dell'interruzione chiama un sottoprogramma. Il circuito che rileva le interruzioni pone a 1 l'apposito bit di IPENDING per ciascuna richiesta pendente. In Figura 3.9 il contenuto di IPENDING viene caricato nel registro R2 e quindi esaminato per determinare quali interruzioni siano pendenti. Se lo schermo ha un'interruzione pendente, allora sarà eseguita la sua routine di servizio dell'interruzione, altrimenti si controlla la tastiera e, se occorresse, altri dispositivi che potrebbero avere richieste pendenti. L'ordine in cui i bit di IPENDING sono controllati stabilisce una priorità per i dispositivi che segnalano interruzioni in caso di richieste simultanee.

Le parti di programma che gestiscono richieste di interruzioni e forniscono il corrispondente servizio ai dispositivi richiedenti sono spesso dette *gestori di interruzioni* (*interrupt handler*). Si noti che mentre il gestore di interruzioni inizia all'indirizzo fisso ILOC, le singole routine di servizio delle interruzioni sono sottoprogrammi che possono essere collocati ovunque in memoria.

In Figura 3.9 si è usato un approccio tramite programma per determinare il dispositivo che ha segnalato un'interruzione. Nei processori che usano le interruzioni vettorizzate, il circuito che rileva richieste di interruzioni carica automaticamente un indirizzo differente nel contatore del programma per ciascuna interruzione a cui sia assegnata una locazione specifica nella tabella dei vettori d'interruzione. Una specifica routine di servizio dell'interruzione viene eseguita fino alla fine per ogni richiesta pendente, anche se altre richieste di interruzioni vengono segnalate allo stesso momento.

Esempi di interruzioni in stile CISC

I compiti suddetti possono essere effettuati usando istruzioni in stile CISC tramite lo stesso approccio di base. La differenza principale è che alcune operazioni, come controllare un bit in un registro di I/O, possono essere effettuate direttamente. I compiti degli Esempi 3.2 e 3.3 possono essere realizzati usando i programmi nelle Figure 3.10 e 3.11, rispettivamente. L'istruzione TestBit è usata per controllare le variabili di condizione di stato. Le istruzioni SetBit e ClearBit sono usate per porre a 1 e a 0, rispettivamente, un singolo bit in un registro di I/O. I commenti nei programmi forniscono spiegazioni su come i compiti voluti siano stati realizzati.

Generalmente, le operazioni di I/O in un calcolatore sono molto più complesse di quello che i semplici esempi esposti sopra suggeriscono. Come si descriverà nel Capitolo 4, il sistema operativo del calcolatore effettua queste operazioni per conto dei programmi degli utenti. Nel Capitolo 7 si discuterà in dettaglio l'hardware usato nelle operazioni di I/O.

Gestore delle interruzioni

ILOC:	Subtract	SP, SP, #12	Salva i registri
	Store	LINK_reg, 8(SP)	
	Store	R2, 4(SP)	
	Store	R3, (SP)	
	MoveControl	R2, IPENDING	Controlla il contenuto di IPENDING
	And	R3, R2, #4	Controlla se lo schermo ha segnalato la richiesta
	Branch_if_[R3]=0	TESTKBD	Se no, controlla se lo ha fatto la tastiera
	Call	DISR	Chiama la routine di servizio delle interruzioni da schermo (display ISR)
TESTKBD:	And	R3, R2, #2	Controlla se la tastiera ha segnalato la richiesta
	Branch_if_[R3]=0	PROSSIMO	Se no, controlla il prossimo dispositivo
	Call	KISR	Chiama la routine di servizio delle interruzioni da tastiera (keyboard ISR)
PROSSIMO:	...		Controlla le interruzioni da altri dispositivi di I/O
	Load	R3, (SP)	Ripristina i registri
	Load	R2, 4(SP)	
	Load	LINK_reg, 8(SP)	
	Add	SP, SP, #12	
	Return-from-interrupt		

Programma principale (Main)

INIZIO:	...		Inizializza i parametri per le ISR
	Move	R2, #2	Abilita le interruzioni nell'interfaccia della tastiera
	StoreByte	R2, KBD_CONT	
	Move	R2, #4	Abilita le interruzioni nell'interfaccia dello schermo
	StoreByte	R2, DISP_CONT	
	MoveControl	R2, IENABLE	
	Or	R2, R2, #6	Abilita le interruzioni nel registro di controllo del processore
	MoveControl	IENABLE, R2	
	MoveControl	R2, PS	
	Or	R2, R2, #1	
	MoveControl	PS, R2	Pone a 1 il bit di abilitazione delle interruzioni in PS
	prossima istruzione		

Routine di servizio delle interruzioni da tastiera

```
KISR:   ...
        ⋮
        Return
```

Routine di servizio delle interruzioni da schermo

```
DISR:   ...
        ⋮
        Return
```

Figura 3.9 Un programma in stile RISC che inizializza e gestisce le interruzioni.

Routine di servizio delle interruzioni

ILOC:	Move	−(SP), R2	Salva il registro
	Move	R2, PNTR	Carica puntatore all'indirizzo
	MoveByte	(R2), KBD_DATA	Scrivi il carattere in memoria
	Add	PNTR, #1	Incrementa il puntatore
ECO:	TestBit	DISP_STATUS, #2	Attendi che lo schermo sia pronto
	Branch=0	ECO	
	MoveByte	DISP_DATA, (R2)	Visualizza il carattere appena letto
	CompareByte	(R2), #CR	Controlla se il carattere appena letto sia CR
	Branch≠0	RTRN	Rientra se non è CR
	Move	EOL, #1	Indica la fine della linea
	ClearBit	KBD_CONT, #1	Disabilita le interruzioni nell'interfaccia della tastiera
RTRN:	Move	R2, (SP)+	Ripristina il registro
	Return-from-interrupt		

Programma principale (Main)

INIZIO:	Move	PNTR, #LINEA	Inizializza il puntatore al buffer
	Clear	EOL	Azzera l'indicatore di fine linea
	SetBit	KBD_CONT, #1	Abilita le interruzioni nell'interfaccia della tastiera
	Move	R2, #2	Abilita le interruzioni da tastiera nel registro di controllo del processore
	MoveControl	IENABLE, R2	
	MoveControl	R2, PS	
	Or	R2, #1	
	MoveControl	PS, R2	Poni a 1 il bit di abilitazione delle interruzioni in PS
	prossima istruzione		

Figura 3.10 Un programma in stile CISC che acquisisce una linea di caratteri usando le interruzioni e li visualizza usando la scansione.

3.2.6 Concetto di eccezione

L'interruzione è un evento che forza la sospensione dell'esecuzione del programma corrente per iniziarne un altro. Finora però sono state considerate solo interruzioni indotte da operazioni di ingresso e uscita; tuttavia il meccanismo di interruzione ha un ruolo centrale anche in svariate altre circostanze.

Il termine *eccezione* è spesso usato per riferirsi a un qualsiasi evento che causa un'interruzione. Quindi le interruzioni di I/O sono un esempio di eccezione.

Di seguito si elencano e illustrano i tipi di eccezione principali.

Ripristino da errore

Il calcolatore contiene svariati accorgimenti tecnologici atti a garantire che tutti i componenti fisici funzionino correttamente. Per esempio, spesso la memoria è dotata di codice per rilevare errori, che permette di stabilire se i dati siano alterati o integri; quando il sistema di rilevamento

Gestore delle interruzioni

ILOC:	Move	−(SP), R2	Salva i registri
	Move	−(SP), LINK_reg	
	MoveControl	R2, IPENDING	Controlla il contenuto di IPENDING
	TestBit	R2, #2	Controlla se lo schermo ha segnalato la richiesta
	Branch=0	TESTKBD	Se no, controlla se lo ha fatto la tastiera
	Call	DISR	Chiama la routine di servizio delle interruzioni da schermo (display ISR)
TESTKBD:	TestBit	R2, #1	Controlla se la tastiera ha segnalato la richiesta
	Branch=0	PROSSIMO	Se no, controlla il prossimo dispositivo
	Call	KISR	Chiama la routine di servizio delle interruzioni da tastiera (keyboard ISR)
PROSSIMO:	...		Controlla le interruzioni da altri dispositivi di I/O
	Move	LINK_reg, (SP)+	Ripristina i registri
	Move	R2, (SP)+	
	Return-from-interrupt		

Programma principale (Main)

INIZIO:	...		Inizializza i parametri per le ISR
	SetBit	KBD_CONT, #1	Abilita le interruzioni nell'interfaccia della tastiera
	SetBit	DISP_CONT, #2	Abilita le interruzioni nell'interfaccia dello schermo
	MoveControl	R2, IENABLE	
	Or	R2, #6	Abilita le interruzioni nel registro di controllo del processore
	MoveControl	IENABLE, R2	
	MoveControl	R2, PS	
	Or	R2, #1	Poni a 1 il bit di abilitazione delle interruzioni in PS
	MoveControl	PS, R2	
	prossima istruzione		

Routine di servizio delle interruzioni da tastiera

```
KISR:   ...
        ⋮
        Return
```

Routine di servizio delle interruzioni da schermo

```
DISR:   ...
        ⋮
        Return
```

Figura 3.11 Un programma in stile CISC che inizializza e gestisce le interruzioni.

Approfondimento applicativo 3.2

I/O con interruzioni NIOS II, ARM, ColdFire, IA-32

Per illustrare l'uso delle interruzioni per operazioni di I/O nei quattro processori in considerazione occorre premettere alcuni dettagli sulla loro architettura, riguardo ai loro registri di controllo, ad altri registri e ad alcune istruzioni non ancora trattate.

L'architettura NIOS II è dotata di sei registri di controllo del processore, denominati ctl*k* ($k = 0...5$), ma anche con nomi più significativi riconosciuti dal'assemblatore, quattro dei quali hanno funzioni qui rilevanti, come pure lo è l'uso specifico del registro r29. Eccone i sinonimi e brevi descrizioni:

ctl0	status	registro di stato del programma
ctl1	estatus	copia del registro di stato salvata in seguito a eccezione
ctl3	ienable	abilitazione di interruzioni in specifiche interfacce di I/O
ctl4	ipending	segnalazione delle interruzioni in attesa di servizio
r29	ea	*exception return address*, registro di collegamento a routine di servizio di eccezioni

Il termine "eccezione" nel presente contesto è sinonimo di interruzione di I/O, tuttavia più in generale non lo è: come spiegato nel Paragrafo 3.2.6 e come sarà illustrato nell'Approfondimento applicativo 3.3, le interruzioni hardware da interfacce di I/O sono un caso particolare del più generale concetto di eccezione. I quattro registri di controllo hanno struttura e funzioni come descritte nel Paragrafo 3.2.4 (si veda la Figura 3.7). Le istruzioni:

rdctl r*i*, ctl*j*
wrctl ctl*j*, r*j*

si usano per trasferire dati fra registri di controllo e registri di uso generale. Per il rientro da routine di servizio d'interruzione si usa l'istruzione di *rientro da eccezione*. Il contenuto del registro va decrementato di 4 prima del rientro perché, contro quanto indicato nel Paragrafo 3.2, non si completa l'esecuzione di un'istruzione interrotta da un'eccezione, che perciò è la prima da eseguire al rientro.

Abilitazione e segnalazione di interruzioni seguono uno schema un po' diverso nell'architettura ARM. Per il momento basti sapere che si abilitano le interruzioni nel registro di stato del programma, CPSR, azzerando il relativo bit di *disabilitazione*. Sono definiti due bit del genere, in corrispondenza dei due tipi di interruzioni di I/O previsti, IRQ e FIQ (*Fast Interrupt Request*). Le richieste di servizio d'interruzione FIQ hanno priorità più alta delle altre e l'eventuale salvataggio di registri d'uso generale è più rapido per esse, grazie alla disponibilità di un più esteso *banco di registri* (*banked registers*) ausiliario. Per altri dettagli sui banchi di registri ARM si veda l'Approfondimento applicativo 3.3. Per trasferire dati fra registro di stato e registri di uso generale si usano le istruzioni:

MRS R*i*, CPSR
MSR CPSR, R*i*

e similmente per il registro SPSR di salvataggio del registro di stato (*Saved Program Status Register*). In caso di richieste di servizio d'interruzione dello stesso tipo da più dispositivi, il processore individua quella da servire tramite scansione dei registri di stato delle rispettive interfacce; non sono cioè definiti i registri di controllo IENABLE e IPENDING, di cui alla Figura 3.7. Il salvataggio dei registri CPSR e PC (R15) rispettivamente nei registri SPSR e LR (R14) è effettuato automaticamente prima di trasferire il controllo alla routine di servizio, che poi ne compie esplicitamente

il ripristino prima del rientro. Il ripristino del PC richiede anche in questo caso una correzione, ma per una ragione diversa da quella vista nel caso del processore NIOS II. L'esecuzione dell'istruzione interrotta viene qui completata, ma le istruzioni ARM sono eseguite in *pipeline* (si veda il Paragrafo 2.3.3), perciò al termine della sua esecuzione è già stata prelevata l'istruzione successiva e il PC è stato ulteriormente incrementato di 4. Prima del rientro va dunque apportata al contenuto del registro di collegamento la stessa correzione che nel caso NIOS II.

La struttura dei programmi NIOS II e ARM in Figura A3.3 riflette quella del programma assemblativo generico mostrato in Figura 3.8. Si assume che i simboli KBD e DIS nel programma NIOS II siano rispettivamente definiti come KBD_DATA e DISP_DATA in quello di Figura A3.1,

	NIOS II		ARM	Commenti
	Routine di servizio delle interruzioni			
ILOC:	subi sp, sp, 16	ILOC	STMFD R13!, {R2, R3}	Salva i registri
	stw r2, 12(sp)			
	stw r3, 8(sp)			
	stw r4, 4(sp)			
	stw r5, (sp)			
	movia r2, PNTR			
	ldw r3, (r2)		LDR R2, PNTR	Puntatore all'indirizzo
	movia r4, KBD			
	ldbio r5, (r4)		LDRB R3, KBD_DATA	Leggi un carattere da tastiera, immagazzinalo in memoria e incrementa il puntatore, aggiornandolo in memoria
	stb r5, (r3)		STRB R3, [R2], #1	
	addi r3, r3, 1			
	stw r3, (r2)		STR R2, PNTR	
ECO:	movia r2,DISP	ECO	LDR BR2, DISP_STATUS	Attendi che lo schermo sia pronto, controllandone il bit di stato DOUT
	ldbio r3, 4(r2)			
	andi r3, r3, 4		TST R2, #4	
	beq r3, r0, ECO		BEQ ECO	
	stbio r5, (r2)		STRB R3, DISP_DATA	Invia carattere allo schermo, controlla se sia CR e rientra se non lo è, altrimenti segnala la fine della linea
	addi r3, r0, 0x0D		CMP R3, #CR	
	bne r5, r3, RTRN		BNE RTRN	
	movi r3, 1		MOV R2, #1	
	movia r5, EOL		STR R2, EOL	
	stw r3, (r5)			
	stbio r0, 8(r4)		MOV R2, #0 STRB R2, KBD_CONT	Disabilita le interruzioni nell'interfaccia di tastiera
RTRN:	ldw r5, (sp)	RTRN	LDMFD R13!, {R2, R3}	Ripristina i registri
	ldw r4, 4(sp)			
	ldw r3, 8(sp)			
	ldw r2, 12(sp)			
	addi sp, sp, 16			
	subi ea, ea, 4			Aggiusta indirizzo di rientro e rientra dall'interruzione
	eret		SUBS R15, R14, #4	

Figura A3.3 Programmi NIOS II e ARM per la lettura e visualizzazione di una linea di caratteri su interruzioni. (*Segue*)

	NIOS II	ARM	Commenti
	Programma principale		
START:	movia r2, LINE		
	movia r3, PNTR		
	stw r2, (r3)	LDR R2, =LINE	Inizializza puntatore al buffer
	movia r2, EOL	STR R2, PNTR	
	stw r0, (r2)	MOV R2, #0 STR R2, EOL	Azzera indicatore di fine linea
	movia r2, KBD		
	movi r3, 2	MOV R2, #2	Abilita le interruzioni nella interfaccia della tastiera
	stbio r3,8(r2)	STRB R2, KBD_CONT	
	rdctl r2, ienable		Abilita le sue interruzioni nel registro di controllo del processore
	ori r2, r2, 2		
	wrctl ienable, r2		
	rdctl r2, status		Abilita le interruzioni nel registro di stato del programma
	ori r2, r2, 1	MOV R2, #&50	
	wrctl status, r2	MSR CPSR, R2	
	...	...	Prossima istruzione

Figura A3.3 (*continua*).

e che siano definiti gli indirizzi delle locazioni di memoria PNTR, EOL e LINE. Nel programma ARM si assume che essi, tranne quello iniziale del buffer LINE, e gli indirizzi dei registri delle interfacce siano abbastanza prossimi da permetterne l'indirizzamento relativo a PC. In questo programma, salvataggio in pila e ripristino dei registri di uso generale usati dalla routine di servizio sono effettuati da istruzioni di trasferimento multiplo di registri (si veda l'Approfondimento applicativo 2.4), ma il ripristino del PC con la correzione del contenuto del registro di collegamento di cui si è detto è compiuto dall'istruzione SUBS. In genere, il suffisso "S" nelle istruzioni aritmetiche specifica che l'operazione abbia effetto sui bit di esito, ma se la destinazione è il PC prescrive il ripristino del registro di stato CPSR con il contenuto salvato nel registro SPSR.

Anche nei processori ColdFire e IA-32 l'abilitazione di interruzioni si effettua solo tramite alcuni bit nel registro di stato del programma e nei registri di controllo delle interfacce di I/O; non sono definiti registri quali IENABLE e IPENDING in Figura 3.7. I meccanismi di gestione delle interruzioni sono però diversi e qui se ne dà cenno.

I tre bit b_{10-8} del registro di stato ColdFire, SR, codificano il livello di priorità del processore nell'esecuzione corrente; tale codifica è anche detta *maschera delle interruzioni* (*interrupt mask*), perché disabilita le interruzioni di minore o uguale priorità. Sono però sempre abilitate le interruzioni *non mascherabili* (*nonmaskable interrupts*), anche quando il processore esegue a livello di priorità 7, il più alto. Il bit b_{13} del registro SR asserito seleziona il modo operativo *Supervisore* del processore, nel quale tutte le istruzioni sono eseguibili, altrimenti il processore è in modo *Utente*, nel quale non sono eseguibili le istruzioni *privilegiate*. Anche gli altri processori qui considerati dispongono di diversi modi operativi, si veda l'Approfondimendo applicativo 3.3). Per adesso basti dire che sono privilegiate, fra altre, le istruzioni ColdFire di accesso a bit del registro SR diversi dai bit di esito, il che spiega perché non si usino codici mnemonici distinti per tali istruzioni. L'istruzione RTE specifica il rientro da routine di servizio di eccezioni, interruzioni incluse.

Anche l'architettura IA-32 distingue fra interruzioni mascherabili e non. Il bit b_9 o IF (*Interrupt Flag*) del registro di stato abilita quelle mascherabili. L'istruzione STI provvede a questo, ponendolo a 1. Le interruzioni IA-32 sono vettorizzate: l'architettura prevede a tal fine un *controllore esterno delle interruzioni*, detto APIC (*Advanced Programmable Interrupt Controller*), che dialoga con le interfacce di I/O: individua le fonti delle richieste di interruzione, ne determina la priorità e ne ottiene l'indice del vettore di interruzione da inviare al processore. Registro di stato e contatore di programma EIP sono automaticamente salvati in pila prima di caricare EIP con l'indirizzo della routine di servizio; l'istruzione IRET di rientro da interruzione li ripristina.

I programmi ColdFire e IA-32 in Figura A3.4 hanno struttura simile a quella del programma assemblativo generico in Figura 3.10. Quali nuove istruzioni ColdFire troviamo ADDQ, variante di ADD analoga alla variante MOVEQ di MOVE descritta nell'Approfondimento applicativo 2.5, e le istruzioni BCLR e BSET che, come la già vista BTST, operano su un solo bit della destinazione, rispettivamente per porlo a 0 e a 1. Le analoghe istruzioni IA-32 sono BTR e BTS. Alcuni operandi di istruzioni IA-32 sono qualificati da parole-chiave: DWORD PTR specifica che l'operando contiene l'indirizzo di un dato a 32 bit in memoria; OFFSET specifica che l'operando non è il contenuto della locazione di memoria specificata bensì il suo indirizzo.

	ColdFire		IA-32	Commenti
	Routine di servizio delle interruzioni			
ILOC:	MOVE.L A2, (A7)	ILOC:	PUSH EAX	Salva i registri
	MOVE.L D0, (A7)		PUSH EBX	
	MOVEA.L PNTR, A2		MOV EBX, PNTR	Inizializza il puntatore all'indirizzo
	CLR.L D0			Azzera il registro dati destinato ai caratteri
	MOVE.B KBD_DATA, D0		MOV AL, KBD_DATA	Trasferisci il carattere al registro (ciò azzera KIN)
	MOVE.B D0, (A2)+		MOV [EBX], AL	Immagazzina il carattere in memoria e aggiorna il puntatore in memoria
	MOVE.L A2, PNTR		INC DWORD PTR PNTR	
	MOVEA.L #DISP_STATUS, A2			Indirizzo del registro di stato dello schermo
ECO:	BTST.B #2, (A2)	ECO:	BT DISP_STATUS, 2	Attendi che lo schermo sia pronto
	BEQ ECO		JNC ECO	
	MOVE.B D0, DISP_DATA		MOV DISP_DATA, AL	Invia il carattere allo schermo (ciò azzera DOUT)
	CMPI.L #CR, D0		CMP AL, CR	Controlla se il carattere letto sia CR
	BNE RTRN		JNE RTRN	Rientra se non lo è
	ADDQ.L #1, EOL		MOV DWORD PTR EOL, 1	Segnala la fine della linea
	MOVEA.L #KBD_CONT, A2			Indirizzo del registro di controllo della tastiera
	BCLR.B #1, (A2)		BTR KBD_CONT, 1	Disabilita interruzioni nell'interfaccia della tastiera
RTRN:	MOVE.L (A7)+, D0	RTRN:	POP EBX	Ripristina i registri
	MOVEA.L (A7)+, A2		POP EAX	
	RTE		IRET	Rientra dall'interruzione

Figura A3.4 Programmi ColdFire e IA-32 per la lettura e visualizzazione di una linea di caratteri su interruzioni. (*Segue*)

	ColdFire	IA-32	Commenti
	Programma principale		
START:	MOVEA.L #LINE, A0	MOV DWORD PTR PNTR, OFFSET LINE	Inizializza puntatore al buffer
	MOVE.L A0, PNTR		
	CLR.L EOL	MOV DWORD PTR EOL, 0	Azzera indicatore di fine linea
	MOVEA.L #KBD_CONT, A0		
	BSET.B #1, (A0)	BTS KBD_CONT, 1	Abilita le interruzioni nell'interfaccia della tastiera
	MOVE.W SR, D0		Carica lo stato del programma in D0
	ANDI.L #$F8FF, D0		Azzera la priorità nella maschera delle interruzioni
	MOVE.W D0, SR	STI	Abilita le interruzioni nel processore
	...	...	Prossima istruzione

Figura A3.4 (*continua*).

scopre la presenza di dati alterati segnala il fatto al processore mandandogli una richiesta di interruzione. Questo è un esempio di eccezione originatasi nell'unità di memoria.

Il processore può, in modo autonomo, interrompere l'esecuzione del programma corrente se questo incorre in errore o in uno stato insolito di funzionamento. Ne sono esempi: il tentativo di prelevare ed eseguire un'istruzione macchina con codice operativo inesistente, il verificarsi di errore nell'esecuzione di un'istruzione aritmetica (divisione per zero e simili), il tentativo di accedere a parti protette della memoria ecc. Anche questi sono esempi di eccezioni, ma originatesi internamente al processore.

Quando il servizio di interruzione parte a seguito di eccezione il processore si comporta in modo analogo all'interruzione proveniente da periferica, sospendendo il programma corrente. La routine di servizio di eccezione deve rimediare all'errore, se ciò è possibile, o almeno informarne l'utente. C'è però una differenza significativa: se l'interruzione proviene da periferica il processore termina comunque l'esecuzione dell'istruzione macchina corrente prima di passare a servire la richiesta, mentre se si tratta di eccezione di solito è impossibile completare l'istruzione corrente (si pensi al caso in cui l'eccezione sia indotta da errore causato dall'istruzione stessa, come la divisione per zero); allora il processore deve passare a servire l'eccezione immediatamente, rinunciando a completare l'istruzione corrente.

Correzione di errore

Un altro tipo di eccezione importante gioca ruolo significativo nella *simulazione* del programma, allo scopo di trovare errori (*debugging*). Di norma il software di sistema del calcolatore comprende il *debugger* (o *programma di simulazione controllata*), che serve al programmatore per scoprire errori algoritmici e di codifica. Il debugger fornisce due funzioni importanti: *esecuzione passo-passo* (*trace*); *punto di osservazione* (*breakpoint*). Queste funzioni sono descritte in dettaglio nel Capitolo 4.

Approfondimento applicativo 3.3

Eccezioni nei processori NIOS II, ARM, ColdFire, IA-32

I seguenti aspetti sono generalmente rilevanti alla gestione delle eccezioni: tipi di eccezione, rispettivi modi operativi e livelli di priorità del processore nel loro servizio; vettori d'interruzione; uso di banchi di registri specifici per diversi tipi di eccezione; rientro da servizio di eccezione. Verranno qui di seguito esaminati brevemente per i quattro processori in considerazione.

Nel processore NIOS II si distinguono le eccezioni hardware, ovvero le interruzioni da dispositivi di I/O, da *eccezioni software*, generate dall'esecuzione di un'istruzione trap, ed *eccezioni di istruzione non definita*. Secondo l'approccio delineato nell'Esempio 3.3, il gestore delle eccezioni, che per default inizia all'indirizzo 0x20 (questo può essere modificato in sede di configurazione hardware), trasferisce il controllo all'appropriata routine di servizio dell'eccezione. Nel caso di un'eccezione software, un uso frequente dell'istruzione trap è il trasferimento del controllo a un altro programma, quale un sistema operativo, come descritto nel Paragrafo 4.9.3. Anche il caso di eccezione di istruzione non definita può essere similmente gestito; per esempio se la configurazione hardware non è dotata dei circuiti per la moltiplicazione o la divisione, il gestore dell'eccezione può chiamare un sottoprogramma che la realizza in software. In entrambi i casi, l'istruzione da eseguire al rientro dal servizio dell'eccezione è quella successiva all'istruzione che ha generato l'eccezione; non va dunque effettuata la correzione del contenuto del registro di collegamento che è necessaria nel caso di eccezioni hardware, come mostrato nell'Approfondimento applicativo 3.2. Se si verifica una condizione di errore non prevista fra i tipi di eccezioni servite, il processore esegue un *reset* che lo porta in uno stato predefinito, quello di avvio, in cui va in esecuzione l'istruzione all'indirizzo 0.

Una configurazione hardware NIOS II può includere dei componenti opzionali, per funzionalità aggiuntive rispetto a quelle di base, quali l'*unità di gestione della memoria* o MMU (*Memory Management Unit*) e il *controllore esterno delle interruzioni* o EIC (*External Interrupt Controller*).

L'MMU fornisce la funzionalità di *memoria virtuale*, descritta nel Paragrafo 8.8. Questa permette di tenere in memoria solo una parte del programma in esecuzione, mentre l'intero programma risiede in un file su disco. Quando occorre caricare un'istruzione o un operando che non si trova in memoria, l'MMU genera un'eccezione che viene servita dal sistema operativo aggiornando il contenuto della memoria con le parti di programma richieste. Un bit del registro di stato NIOS II distingue il modo operativo Utente, in cui non tutte le istruzioni sono eseguibili, dal modo Supervisore (*Supervisor*), nel quale lo sono tutte, incluse le istruzioni dette appunto *privilegiate*, non eseguibili nel modo Utente. Nella configurazione hardware di base il processore NIOS II opera solo nel modo Supervisore. In presenza della MMU le applicazioni sono eseguite in modo Utente, mentre l'esecuzione del sistema operativo è in modo Supervisore.

L'EIC ha funzionalità simili a quelle dell'APIC nell'architettura IA-32, indicate nell'Approfondimento applicativo 3.2. Inoltre, per ridurre ulteriormente la latenza d'interruzione si possono includere nella configurazione hardware banchi di registri-ombra (*shadow register*) dei 32 registri di base, eliminando così la necessità di salvarli in pila al servizio di ogni richiesta. Alcuni bit riservati nel registro di stato del programma codificano il livello corrente di priorità e il banco di registri in uso.

Banchi di registri specifici per tipi diversi di eccezione sono disponibili nell'architettura ARM, che definisce diversi livelli di priorità del loro servizio in caso di eccezioni simultanee, due dei quali per il servizio di interruzioni di I/O, e sette modi operativi del processore. Tale informazione è codificata nel registro di stato del programma CPSR. Nei modi operativi Utente

e Sistema il processore usa il banco di registri principale; nei cinque *modi di eccezione* (*exception mode*) definiti, alcuni registri sono sostituiti da registri ausiliari, distinti per ciascun modo. In tutti i modi di eccezione, il banco ausiliario include sempre i registri SPSR, R13 (SP) e R14 (LR); nel modo FIQ esso include anche i registri R8–R12, per ridurre ulteriormente la latenza d'interruzione quando nella routine di servizio occorra usare registri generali. Inoltre, a ciascun modo di eccezione è associata una pila diversa in un'area di memoria allocata dal sistema operativo, che inizializza i registri SP ausiliari.

A tipi di eccezione diversi corrispondono distinti *vettori di eccezione*. La Tabella A3.1 descrive i tipi di eccezione ARM, i rispettivi livelli di priorità, modi operativi e indirizzi dei vettori di eccezione, nonché l'istruzione da impiegare per il rientro, come spiegato più avanti.

Istruzioni privilegiate, come quelle di trasferimento fra registri di stato e registri di uso generale, non sono eseguibili nel modo Utente, mentre lo sono nei modi di eccezione e nel modo Sistema. Quest'ultimo è accessibile solo da un modo di eccezione; la sua funzione consiste nel permettere la chiamata di sottoprogrammi in routine di servizio di eccezioni usando pila e banco di registri normali invece che ausiliari; il rientro alla routine di servizio di eccezione ripristina l'uso di registri e pila ausiliari, relativi al suo modo.

Con riguardo all'istruzione di rientro indicata, per ciascun tipo di eccezione, in Tabella A3.1, si rileva innanzitutto che alcuni tipi di eccezione impediscono il completamento dell'istruzione interrotta, per esempio un'istruzione privilegiata in modo Utente o un'istruzione non definita; altri ne comportano l'esecuzione al rientro dal servizio d'eccezione, quali gli errori di accesso a memoria quando la locazione di memoria riferita ha un indirizzo valido ma che richiede il caricamento in memoria di una parte del programma (l'eccezione è segnalata da una MMU e genera quindi operazioni di I/O al termine delle quali l'istruzione va rieseguita); altri ancora, quali le interruzioni di I/O o software, queste generate dall'istruzione SWI, comportano il rientro all'istruzione successiva. Inoltre, il contenuto del PC salvato nel registro di collegamento R14 dipende dall'effetto dell'esecuzione in pipeline sull'istruzione interrotta. Così si spiega l'istruzione di rientro da interruzioni di I/O, già esaminata nell'Approfondimento applicativo 3.2. Come per il processore NIOS II, le eccezioni di istruzione non definita sono analoghe a interruzioni software, purché esista una realizzazione in software dell'operazione richiesta; in questo caso, come in quello di rientro da servizio d'interruzione software, non va apportata alcuna correzione al contenuto del registro di collegamento (il suffisso "S" nel codice mnemonico MOVS ha la stessa funzione che nell'istruzione SUBS; si veda l'Approfondimento applicativo 3.2). La correzione indicata per le eccezioni di errore di accesso

Tabella A3.1 Tipi di eccezioni e loro caratteristiche nei processori ARM.

Tipo di eccezione	P	Modo	VA	Istruzione di rientro
Interruzione di I/O ordinaria	4	IRQ	28	SUBS R15, R14, #4
Interruzione di I/O urgente	3	FIQ	24	SUBS R15, R14, #4
Interruzione software	–	SVC	8	MOVS R15, R14
Avvio/reset	1	SVC	0	–
Errore di accesso a operando	2	Abort	16	SUBS R15, R14, #8
Errore di accesso a istruzione	5	Abort	12	SUBS R15, R14, #4
Istruzione non definita	6	Undefined	4	MOVS R15, R14

Legenda:
P Priorità (massima = 1);
VA Indirizzo del vettore di eccezione;
IRQ Interrupt request;
FIQ Fast Interrupt Request;
SVC Supervisor Call.

a memoria corrisponde al caso di indirizzo valido descritto sopra, ed è giustificata dal fatto che l'istruzione va rieseguita a caricamento avvenuto; la correzione differisce nei due tipi di errore di accesso indicati in Tabella A3.1 per il diverso effetto del prelievo in pipeline con l'esecuzione nei due casi. Il *reset* ha il significato descritto per il processore NIOS II, sebbene in quest'ultimo non sia propriamente considerato un'eccezione.

Non sono definiti banchi di registri ausiliari nei processori CISC qui considerati; invero, la disponibilità di numerosi registri è tipica delle architetture RISC, sia perché varie caratteristiche dello stile RISC, indicate nel Paragrafo 2.11, ne comportano un uso più frequente sia perché i programmi in stile RISC di solito constano di un maggior numero di istruzioni, ragion per cui risparmiare quelle di salvataggio dei registri nel servizio di eccezioni dà un più significativo contributo alla riduzione della latenza.

Si è già detto, nell'Approfondimento applicativo 3.2, dei due modi operativi del processore ColdFire rispetto all'esecuzione di istruzioni privilegiate. La commutazione dal modo Utente al modo Supervisore avviene solo in risposta a una richiesta di servizio d'interruzione ed è automatica, mentre quella opposta può ottenersi esplicitamente con un'istruzione (privilegiata) che modifica il contenuto del registro SR, oppure implicitamente al rientro da una routine di servizio d'interruzione a un programma eseguito in modo Utente. Il bit b_{15} del registro SR, ovvero quello nella posizione più significativa, seleziona il modo di esecuzione passo-passo, descritto nel Paragrafo 4.6, che genera un'eccezione apposita dopo l'esecuzione di ogni istruzione.

Oltre alle interruzioni di I/O, altre eccezioni nell'architettura IA-32 sono segnalate a seguito di eventi dovuti all'esecuzione del programma, quali per esempio codice operativo non valido, divisione per zero, trabocco. Anche queste eccezioni sono vettorizzate, ma i vettori d'eccezione, come nel caso dell'architettura ARM, sono predefiniti per diversi tipi di eccezione. Il registro di stato del programma, come nel processore ColdFire, ha un bit, b_8 o TF (*Trap Flag*), che seleziona il modo di esecuzione passo-passo, usato anche per l'esecuzione con punti di osservazione, discussi nel Paragrafo 4.6. Con riguardo ai modi operativi, i due bit b_{13-12} del registro di stato codificano il *livello di privilegio di I/O* o IOPL (*I/O Privilege Level*). Assieme a un'organizzazione in segmenti della memoria, questo permette una sofisticata struttura di gestione delle eccezioni in sistemi operativi che realizzano funzionalità di protezione. Nel modo operativo più semplice (*simple mode*) non sono definiti privilegi e tutti i programmi condividono lo stesso segmento della memoria; per ulteriori dettagli in proposito si veda il Caso di studio C2.

Eccezione e sistema operativo

Il sistema operativo ha in carico il coordinamento generale di tutte le attività del calcolatore. Esso fa uso molto esteso di eccezioni per interagire con i programmi applicativi e di utente controllandone l'esecuzione. Esso usa interruzioni hardware per effettuare operazioni di I/O. Questo argomento è discusso nel Capitolo 4.

Sommario

Nel capitolo sono state illustrate e discusse le due tecniche fondamentali di gestione di ingresso e uscita (I/O) impiegate nel calcolatore. La tecnica più semplice è quella a controllo di programma, in cui il processore, guidato dal programma, si incarica di effettuare il trasferimento dei dati, nonché tutte le operazioni di controllo accessorie. La seconda tecnica, più sofisticata, è quella di interruzione: alla periferica è data facoltà di interrompere il flusso di esecuzione normale del pro-

gramma e di forzare il processore a eseguire una routine di servizio apposita che si incarica di comunicare con la periferica stessa. Questa tecnica permette anche di realizzare schemi di priorità per tenere conto di richieste di interruzione a livelli di urgenza differenziati. Tutti i calcolatori, o quasi, dispongono di un meccanismo di interruzione, ma la complessità e la raffinatezza di tale meccanismo variano molto a seconda del calcolatore.

Sono state affrontate le questioni riguardanti l'I/O dal punto di vista del programmatore. Nel Capitolo 7 saranno considerati gli aspetti hardware e alcuni standard di I/O comunemente usati.

4 Software

In questo capitolo

- Software necessario per preparare ed eseguire programmi
- Assemblatori
- Caricatori (*loader*)
- Collegatori (*linker*)
- Compilatori
- *Debugger*
- Interazioni tra linguaggio assemblativo e linguaggio C
- Sistemi operativi

Il Capitolo 2 ha introdotto l'insieme di istruzioni di un calcolatore e ha illustrato come scrivere programmi in linguaggio assemblativo. Il Capitolo 3 ha mostrato come scrivere programmi che effettuano operazioni di ingresso/uscita. Questo capitolo fornirà una visione d'insieme del software necessario per preparare ed eseguire programmi.

I programmi in linguaggio assemblativo sono scritti usando una notazione simbolica che è facilmente comprensibile dai programmatori. Questi programmi devono essere tradotti in codice di linguaggio macchina prima che essi possano essere eseguiti in un calcolatore, come spiegato nel Paragrafo 2.5. Questa traduzione viene effettuata dal programma assemblatore, che interpreta i codici mnemonici che rappresentano istruzioni macchina e le direttive di assemblatore per le dichiarazioni dei dati e per il controllo del processo assemblativo.

Avendo presentato come scrivere programmi in linguaggio assemblativo, si discuterà ora il processo completo di preparazione di programmi per la loro esecuzione. Sarà descritto:

- come l'assemblatore traduce un programma sorgente scritto in linguaggio assemblativo in un programma oggetto consistente di istruzioni macchina e dati in formato binario;
- come programmi oggetto vengono caricati nella memoria di un calcolatore;
- come si fa iniziare e terminare l'esecuzione dei programmi;
- come formare programmi più grandi collegando programmi interrelati;
- come individuare errori di programmazione durante l'esecuzione di un programma.

Quindi si esamineranno alcune questioni riguardanti programmi preparati in un linguaggio di alto livello come il C e, infine, si prenderà in considerazione il ruolo del sistema operativo nella gestione e nel coordinamento dell'uso delle risorse del calcolatore.

4.1 Processo assemblativo

Per preparare un programma sorgente, il programmatore usa un programma di utilità chiamato *editore di testi* che permette di inserire, attraverso una tastiera, le istruzioni di un pro-

gramma sorgente e di salvarle in un *file*. Il file contenente il programma sorgente è una sequenza di caratteri alfanumerici codificati in binario ed è identificato da un nome scelto dall'utente. I file sono in genere immagazzinati in un dispositivo di memoria secondaria, come un disco magnetico.

Dopo aver preparato il file sorgente, il programmatore usa un altro programma di utilità chiamato *assemblatore*. Questo traduce i programmi sorgente scritti in un linguaggio assemblativo in programmi oggetto che consistono di istruzioni macchina. Questo processo è spesso chiamato *assemblaggio* di un programma. Inoltre l'assemblatore converte la rappresentazione in linguaggio assemblativo dei dati in sequenze binarie che sono parte del programma oggetto. Dopo aver caricato un file sorgente dal disco alla memoria e averlo tradotto in un programma oggetto, l'assemblatore salva il programma oggetto in un singolo file su disco.

Il programma sorgente usa codici mnemonici per rappresentare i codici operativi di istruzioni macchina. Un insieme di regole sintattiche regolamentano le specifiche dei modi di indirizzamento per gli operandi di queste istruzioni. L'assemblatore genera la codifica binaria per il codice operativo e gli altri campi delle istruzioni; inoltre l'assemblatore riconosce le direttive che specificano numeri e caratteri e le direttive che allocano spazio di memoria per aree dati. Usando le direttive EQU (*equate*), il programmatore può definire dei nomi per rappresentare costanti. Questi nomi possono quindi apparire nel programma sorgente come operandi delle istruzioni. Possono essere definiti nomi anche per le etichette di indirizzi per destinazioni di salti, punti di ingresso di sottoprogrammi o locazioni di dati in memoria. Alle etichette degli indirizzi sono assegnati valori in base alla loro posizione relativa rispetto all'inizio di un programma assemblato.

Via via che l'assemblatore analizza il programma sorgente, tiene traccia di tutti i nomi e dei valori corrispondenti nella *tabella dei simboli* (*symbol table*). Ogni volta che compare un nome, esso è sostituito con il suo valore reperito nella tabella.

4.1.1 Assemblatore a due passi

Il processo di assemblaggio avviene in modo sequenziale, una riga di programma alla volta. L'assemblatore tiene traccia nella tabella dei simboli di tutti i simboli che incontra e dei rispettivi valori numerici che riesce ad associare loro. In tal modo, se il simbolo si ripete, il valore associatogli è subito noto consultando tabella. Si presenta un intoppo quando si incontra il simbolo prima di avergli potuto associare un valore numerico. Ciò accade tipicamente quando si incontra un salto in avanti la cui destinazione sia indicata in modo simbolico (cioè quanto si fa di solito); l'assemblatore non sa ancora come risolvere numericamente il simbolo perché non lo trova nella tabella non avendolo ancora incontrato come etichetta di un'istruzione. La soluzione più semplice (e più diffusa) consiste nell'organizzare il processo di assemblaggio *in due passate*, ovvero scandire due volte il programma. Naturalmente per fare ciò l'assemblatore conosce gli ingombri di ciascun tipo di istruzione e li addiziona scandendo il programma. Durante il primo passo ci si limita a raccogliere in tabella tutti i simboli, in particolare etichette, e ad associare loro i rispettivi valori numerici, mentre nel secondo passo si sostituisce ogni simbolo con il rispettivo valore numerico reperibile in tabella. Tale modo di operare si chiama appunto *assemblaggio in due passate* (*two pass assembling*), e produce un programma oggetto completo.

4.2 Caricare ed eseguire programmi oggetto

I programmi oggetto generati dall'assemblatore vengono immagazzinati in file su disco. Per eseguire uno specifico programma oggetto, questo viene prima caricato dal disco e trasferito

in memoria, quindi l'indirizzo della prima istruzione da eseguire viene caricato nel contatore del programma. Un programma di utilità chiamato *caricatore* (*loader*) viene usato per effettuare queste operazioni e si attiva quando un utente inserisce un comando per eseguire un programma oggetto che è memorizzato sul disco. Il comando dell'utente specifica il nome del file oggetto, abilitando quindi il caricatore a trovare il file sul disco. Il caricatore trasferisce il programma oggetto dal disco a una locazione di memoria specificata; per fare ciò deve conoscere la lunghezza del programma e l'indirizzo di memoria dove caricare il programma. L'assemblatore in genere inserisce queste informazioni nell'intestazione (*header*) del file oggetto, prima delle istruzioni macchina e dei dati del programma oggetto.

Un modo per inserire i comandi dell'utente consiste nel digitarli sulla tastiera. Un'alternativa più comunemente usata è impiegare un'*interfaccia utente grafica* o GUI (*Graphical User Interface*). In questo caso, l'utente usa un *mouse* per selezionare il file oggetto desiderato, quindi il software GUI passa al caricatore l'informazione sulla locazione del file oggetto sul disco.

Una volta che il programma oggetto sia stato caricato in memoria, il caricatore comincia la sua esecuzione saltando alla prima istruzione da eseguire. Nel programma sorgente, il programmatore indica la prima istruzione con una speciale etichetta come INIZIO. L'assemblatore include il valore di questa etichetta di indirizzo nell'intestazione del programma oggetto.

Quando un programma oggetto completa il suo lavoro, la sua esecuzione deve essere terminata in un modo ben definito, il che permette di recuperare lo spazio di memoria che contiene il programma oggetto e abilita l'utente a inserire un nuovo comando per eseguire un altro programma oggetto. Tali questioni sono in genere risolte dal software del sistema operativo (SO), che sarà discusso nel Paragrafo 4.9.

4.3 Il collegatore

Nelle sezioni precedenti si è assunto che tutte le istruzioni e i dati di un particolare programma siano specificate in un singolo file sorgente da cui l'assemblatore genera un programma oggetto. In molti casi, un programmatore potrebbe desiderare di chiamare un sottoprogramma creato da altri programmatori. Non è conveniente e neppure pratico raccogliere tutti i sottoprogrammi necessari, possibilmente da molti file sorgente separati, in un singolo file sorgente per farlo elaborare all'assemblatore.

In alternativa, una prassi comune è usare l'assemblatore su ogni file sorgente separatamente. In questo caso, ogni singolo file in uscita non sarà un programma oggetto completo. Ogni programma può contenere riferimenti a *nomi esterni*, che tipicamente sono etichette di indirizzi definiti in altri file sorgente. Quando elabora un file sorgente, l'assemblatore identifica i riferimenti esterni e costruisce una lista di questi nomi e di istruzioni che hanno riferimenti a essi. L'assemblatore include questa lista nel file oggetto che genera dal file sorgente.

Un programma di utilità chiamato *collegatore* (*linker*) viene usato per combinare i contenuti di file oggetto separati in un solo programma oggetto. Esso risolve i riferimenti a nomi esterni usando l'informazione registrata su ciascun file oggetto. Al collegatore servono le posizioni relative delle etichette degli indirizzi definite nei file sorgenti, così che possa determinare i valori degli indirizzi assoluti quando combina file oggetto separati. Per questo compito, l'informazione sulle etichette degli indirizzi alle quali si possa fare riferimento in altri file sorgente deve essere *esportata* da ciascun file sorgente. Normalmente, si richiede al programmatore di indicare le etichette che devono essere esportate. I nomi esportati sono inclusi

dall'assemblatore in ogni file oggetto che genera, insieme con una lista dei nomi esterni usati nel programma e di istruzioni che vi si riferiscono.

Il collegatore usa l'informazione in ciascun file oggetto e la dimensione conosciuta dei programmi in linguaggio macchina per costruire una mappa della memoria del file oggetto complessivo finale. I valori finali corrispondenti a etichette di indirizzi esportate sono determinati quando tutti i singoli file oggetto sono stati raccolti insieme e assegnati alle loro locazioni finali in memoria. A questo punto, i riferimenti a nomi esterni possono essere risolti. I valori finali degli indirizzi determinati dal collegatore sono sostituiti nelle istruzioni che contengono riferimenti esterni. Quando tutti i riferimenti esterni sono stati risolti, il programma oggetto finale è completo.

Il programmatore può scegliere di fissare alcuni degli indirizzi di istruzioni e dati in un file oggetto. Questo può essere fatto usando direttive come ORIGIN in un file sorgente in linguaggio assemblativo. In questo caso, il programmatore deve assicurarsi che istruzioni e dati provenienti da file oggetto diversi non si sovrappongano in memoria. Un approccio più flessibile è non usare direttive ORIGIN, dando al collegatore la libertà di selezionare l'indirizzo di partenza per il programma oggetto e di assegnare concordemente indirizzi assoluti. Il collegatore si assicura che file oggetto differenti non si sovrappongano tra loro o con locazioni di memoria speciali come vettori di interruzioni.

4.4 Librerie

I sottoprogrammi scritti per un programma possono essere utili per altri programmi. È pratica comune raccogliere file oggetto contenenti tali sottoprogrammi in un file di *libreria* immagazzinato sul disco. I sottoprogrammi della libreria possono essere quindi collegati ad altri file oggetto per qualsiasi programma. Per creare un file di libreria viene usato un programma di utilità *archiviatore* (*archiver*). Il file di libreria comprende informazioni necessarie al collegatore per risolvere i riferimenti a nomi esterni in un programma che chiama routine della libreria.

Quando invoca il collegatore, il programmatore specifica i file di libreria desiderati. Il collegatore estrae i file oggetto rilevanti dalla libreria e li include nel programma oggetto finale.

4.5 Il compilatore

La programmazione in linguaggio assemblativo richiede la conoscenza di dettagli specifici della macchina, che variano da un calcolatore a un altro, mentre programmare con un linguaggio di alto livello come C, C++ o Java non richiede tale conoscenza. Affinché un programma scritto in un linguaggio di alto livello possa essere eseguito in un calcolatore, esso deve essere prima tradotto in linguaggio assemblativo e quindi nel linguaggio macchina del calcolatore. Un programma di utilità chiamato *compilatore* effettua il primo lavoro di traduzione. Un file sorgente in un linguaggio di alto livello viene preparato dal programmatore e memorizzato sul disco. Da questo file sorgente il compilatore genera istruzioni e direttive in linguaggio assemblativo e le scrive in un file in uscita. Quindi il compilatore invoca l'assemblatore per assemblare questo file.

Spesso è conveniente ripartire un programma sorgente di alto livello in più file, raggruppando insieme i sottoprogrammi in base alle relazioni tra i loro compiti. In ciascun file sorgente devono essere dichiarati i nomi di sottoprogrammi esterni e le variabili per i dati presenti in altri file. Questo è necessario per abilitare il compilatore a controllare i tipi di dati e controllare

errori. Per ogni file sorgente, il compilatore genera un file in linguaggio assemblativo, quindi invoca l'assemblatore per generare un file oggetto. Il collegatore combina tutti i file oggetto, includendo le routine di libreria, per creare il programma oggetto finale.

Un vantaggio importante dato dal programmare in un linguaggio di alto livello è che il compilatore automatizza molti dei noiosi compiti che un programmatore deve fare quando programma in linguaggio assemblativo. Per esempio, quando genera la rappresentazione in linguaggio assemblativo di sottoprogrammi, il compilatore effettua tutti i compiti relativi alla gestione delle aree di attivazione.

4.5.1 Ottimizzazioni del compilatore

Se il compilatore usa un approccio diretto per generare un programma in linguaggio assemblativo da un file sorgente scritto in un linguaggio di alto livello, non necessariamente produrrà il programma più efficiente in termini di tempo di esecuzione o di dimensione. Miglioramenti alle prestazioni possono essere ottenuti se il compilatore usa tecniche come il riordinamento delle istruzioni prodotte da un approccio diretto. Un compilatore con tali capacità è detto compilatore *ottimizzante*.

Poiché molto del tempo d'esecuzione di un programma viene trascorso nei cicli, i compilatori possono applicare ottimizzazioni particolarmente efficaci per i cicli. Per esempio, un programma sorgente di alto livello può usare una variabile in memoria come contatore in un ciclo. Questa variabile deve essere letta e scritta per incrementare il suo valore a ogni passata del ciclo. Un'implementazione diretta in linguaggio assemblativo di questo compito consiste di istruzioni Load, Add e Store dentro un ciclo. Un'implementazione migliore viene prodotta da un compilatore che riconosce che il valore del contatore può essere tenuto in un registro mentre si esegue il ciclo. In questo caso, le istruzioni Load e Store non sono più necessarie dentro il ciclo. Un'istruzione Load può essere usata prima dell'ingresso nel ciclo per inserire un valore iniziale nel registro; un'istruzione Store può essere necessaria dopo l'uscita dal ciclo per registrare il valore finale del contatore.

4.5.2 Combinare programmi scritti in linguaggi diversi

Il Paragrafo 4.3 descrive il collegatore, che collega vari file oggetto al fine di generare il programma oggetto. In alcuni casi un programmatore potrebbe desiderare di combinare file oggetto prodotti da file sorgenti scritti in un linguaggio di alto livello e file sorgenti scritti in linguaggio assemblativo. Per esempio, il programmatore può realizzare sottoprogrammi speciali in linguaggio assemblativo che sono stati attentamente progettati per ottenere alte prestazioni. Un programma sorgente in un linguaggio di alto livello può quindi chiamare questi sottoprogrammi in linguaggio assemblativo. Analogamente, un programma in linguaggio assemblativo può chiamare sottoprogrammi in un linguaggio di alto livello.

La Figura 4.1 illustra il flusso completo per generare un programma oggetto da vari file sorgente e da routine di libreria.

4.6 Il debugger

Un programma oggetto viene generato con successo quando non ci sono errori di sintassi o nomi sconosciuti nei file sorgenti del programma. Tali problemi vengono rilevati e riferiti dall'assemblatore, compilatore o collegatore. Il programmatore quindi effettua le correzioni necessarie nei file sorgenti.

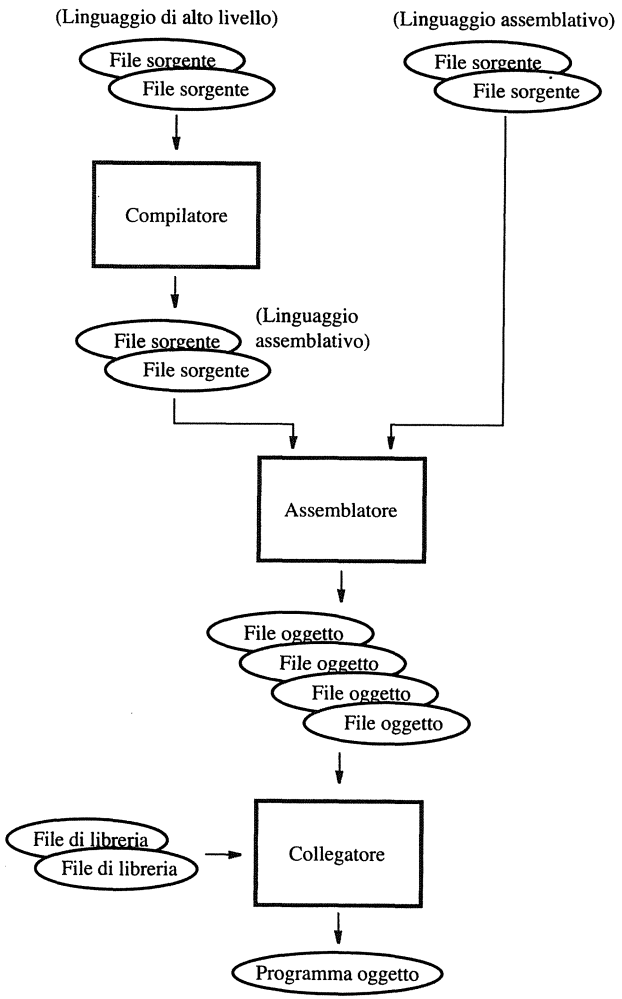


Figura 4.1 Flusso della generazione di un programma oggetto.

Tuttavia, quando un programma oggetto viene eseguito, esso può produrre risultati incorretti dovuti a errori di programmazione, o *bug*, spesso difficili da isolare. Per aiutare il programmatore a individuare tali errori, può essere usato un programma di utilità detto *debugger* (o *programma di simulazione controllata*). Esso abilita il programmatore a fermare l'esecuzione del programma oggetto in alcuni punti di interesse e a esaminare il contenuto di vari registri del processore e di locazioni di memoria. In questo modo, il programmatore può confrontare i valori calcolati con i valori attesi in qualsiasi punto dell'esecuzione per determinare dove può esservi un errore di programmazione. Con queste informazioni, il programmatore può quindi riesaminare e correggere il file sorgente erroneo.

A supporto delle funzioni del debugger, i processori hanno solitamente modalità di funzionamento speciali e interruzioni speciali. Due esempi di supporti al debug sono il *modo di esecuzione passo-passo* (*trace mode*) e i *punti di osservazione* (*breakpoint*).

Modo di traccia di esecuzione

Se il processore funziona in modo di esecuzione passo-passo, detto anche *modo di traccia di esecuzione* (*trace mode*), si genera internamente un'eccezione ogniqualvolta l'istruzione macchina termina e il programma debugger stesso si attiva funzionando come routine di servizio dell'eccezione. Il debugger permette al programmatore di prendere visione dello stato corrente dei registri del processore, di parole di memoria a scelta e così via, di solito tramite interfaccia grafica apposita, pertanto permette di scoprire le cause dell'errore. Quando l'utente immette un comando per riprendere l'esecuzione del programma oggetto, verrà eseguita un'istruzione di rientro da interruzione. La prossima istruzione del programma su cui si sta effettuando il debug viene quindi eseguita e successivamente il debugger viene nuovamente attivato con un'altra eccezione. Le interruzioni del modo di traccia di esecuzione vengono disabilitate automaticamente quando si entra nella routine del debugger e riabilitate al rientro verso il programma oggetto.

Punti di osservazione

I *punti di osservazione* (*breakpoint*) forniscono un supporto simile al debug basato su interruzioni, ma il programma oggetto viene interrotto solo quando il flusso di esecuzione raggiunge certi punti prestabiliti dal programmatore. Per esempio, il programmatore può fissare un punto di osservazione per determinare se un certo sottoprogramma venga mai raggiunto. Quando ciò accade, il debugger verrà attivato da un'interruzione. Il programmatore può quindi esaminare lo stato dell'elaborazione in quel punto. Il vantaggio di usare un punto di osservazione è che l'esecuzione procede alla massima velocità finché non si raggiunga il punto di osservazione.

Un'istruzione speciale Trap, detta anche di *trappola* o *interruzione software* (*software interrupt*), è in genere usata per realizzare i punti di osservazione. L'esecuzione di questa istruzione provoca le stesse attività intraprese quando viene ricevuta una richiesta di interruzione hardware. Quando il debugger ha il controllo dell'esecuzione, esso permette all'utente di inserire un punto di osservazione che interrompe l'esecuzione subito prima dell'istruzione i nel programma oggetto. Il debugger salva l'istruzione i in una locazione temporanea e la sostituisce con un'istruzione di interruzione software, quindi l'utente inserisce un comando per riprendere l'esecuzione del programma e il debugger esegue un'istruzione di rientro da interruzione. Le istruzioni del programma vengono elaborate normalmente fino a quando si incontra l'istruzione di interruzione software. A quel punto, l'elaborazione dell'interruzione provoca la riattivazione del debugger, permettendo all'utente di esaminare lo stato dell'elaborazione.

Quando l'utente inserisce il comando per riprendere l'esecuzione, il debugger deve svolgere vari compiti, non solo per eseguire l'istruzione i ma anche per inserire nuovamente lo stesso punto di osservazione; prima, però, dovrà ripristinare l'istruzione i alla sua locazione originaria nel programma. Questa sarà la prima istruzione da eseguire quando il programma riprende l'esecuzione. Quindi il debugger deve reinserire il punto di osservazione, perciò dovrà far in modo che una seconda interruzione avvenga dopo che l'istruzione i sia stata eseguita. Per far questo, può abilitare il modo di esecuzione passo-passo, se disponibile. In alternativa, può inserire un punto di osservazione temporaneo alla locazione dell'istruzione $i + 1$, quindi riprendere l'esecuzione del programma. Dopo che l'istruzione i sia stata eseguita, una seconda interruzione avviene a causa del punto di osservazione temporaneo al posto dell'istruzione $i + 1$. Questa volta il debugger ripristina l'istruzione $i + 1$, reinserisce il punto di osservazione sull'istruzione i e riprende l'esecuzione del programma interrotto.

4.7 Uso di un linguaggio di alto livello per operazioni di I/O

Il compilatore, l'assemblatore e il collegatore offrono considerevole flessibilità al programmatore. I programmi sorgente possono essere scritti interamente in linguaggio assemblativo, o interamente in un linguaggio di alto livello, o tramite un misto di linguaggi. L'uso di un linguaggio di alto livello è preferibile in molte applicazioni, poiché il tempo di sviluppo è più breve e il codice desiderato è più facile da generare e mantenere. In questo paragrafo e nel prossimo saranno mostrati alcuni programmi di esempio per operazioni di I/O che usano il linguaggio C per illustrare questo approccio.

Si considerino le seguenti attività di I/O. Un programma usa l'approccio della scansione per leggere caratteri a 8 bit da una tastiera e mandarli a uno schermo via via che vengono inseriti da un utente. Il Capitolo 3 presenta esempi di interfacce con l'unificazione degli spazi di indirizzamento di memoria e I/O per tali dispositivi. La Figura 4.2 mostra un programma in linguaggio assemblativo per le operazioni di I/O usando le interfacce della Figura 3.3.

La Figura 4.3 mostra un programma in linguaggio C che effettua lo stesso compito. In questo linguaggio, a un puntatore può essere assegnata una qualsiasi locazione di memoria, incluse le locazioni relative ai dispositivi di I/O. Il valore del puntatore è quindi l'indirizzo di una tale locazione. Se il contenuto di questa locazione deve essere trattato come un carattere, il puntatore dovrebbe essere dichiarato di tipo puntatore a carattere. Questo definisce che il contenuto è della lunghezza di un byte, che è la dimensione dei registri di I/O in Figura 3.3. I costrutti *define* in Figura 4.3 sono usati per associare i necessari indirizzi costanti ai nomi simbolici dei puntatori. Questi costrutti hanno lo stesso scopo dell'istruzione EQU in Figura 4.2. Essi abilitano il compilatore a sostituire i nomi simbolici con valori numerici e indicano,

```
KBD_DATA     EQU            0x4000          Registro dati della tastiera (8 bit)
KBD_STATUS   EQU            0x4004          Registro di stato della tastiera (il bit 1 è
                                             la condizione di stato KIN)
DISP_DATA    EQU            0x4010          Registro dati dello schermo (8 bit).
DISP_STATUS  EQU            0x4014          Registro di stato dello schermo (il bit 2
                                             è la condizione di stato DOUT)
             Move           R2, #KBD_DATA   Puntatore all'interfaccia del dispositivo
                                             tastiera
             Move           R3, #DISP_DATA  Puntatore all'interfaccia del dispositivo
                                             schermo
CICLO_KBD :  LoadByte       R4, 4(R2)       Controlla se c'è in ingresso un carattere
             And            R4, R4, #2       dalla tastiera
             Branch_if_[R4]=0 CICLO_KBD
             LoadByte       R5, (R2)        Leggi il carattere ricevuto
CICLO_DISP : LoadByte       R4, 4(R3)       Controlla se lo schermo è pronto
             And            R4, R4, #4       per un nuovo carattere
             Branch_if_[R4]=0 CICLO_DISP
             StoreByte      R5, (R3)        Scrivi il carattere ricevuto sullo schermo
             Branch         CICLO_KBD
```

Figura 4.2 Programma assemblativo per trasferire caratteri da una tastiera a uno schermo.


```
/* Definizione di indirizzi di registri */
#define KBD_DATA        (volatile char *) 0x4000
#define KBD_STATUS      (volatile char *) 0x4004
#define DISP_DATA       (volatile char *) 0x4010
#define DISP_STATUS     (volatile char *) 0x4014

void main()
{
   char ch;

   /* Trasferisci i caratteri */
   while (1) {                                  /* Ciclo senza fine */
   while ((*KBD_STATUS & 0x2) == 0);            /* Attendi un nuovo carattere */
   ch = *KBD_DATA;                              /* Leggi il carattere dalla tastiera */
   while ((*DISP_STATUS & 0x4) == 0);           /* Attendi che lo schermo sia pronto */
   *DISP_DATA = ch;                             /* Trasferisci il carattere allo schermo */
```

Figura 4.3 Programma C che esegue lo stesso compito del programma assemblativo in Figura 4.2.

inoltre, il tipo di dati per i puntatori. Il compilatore può generare istruzioni in linguaggio assemblativo con valori noti e dimensioni corrette dei dati.

Si noti che i puntatori KBD_STATUS e DISP_STATUS sono dichiarati *volatili*. Questo è necessario poiché il programma effettua solamente la lettura del contenuto delle locazioni corrispondenti. Nessun dato è scritto in queste locazioni. Un compilatore ottimizzante potrebbe rimuovere istruzioni che apparentemente non hanno impatto, incluse istruzioni che si riferiscono a locazioni in memoria lette ma mai scritte. Poiché il contenuto dei registri KBD_STATUS e DISP_STATUS cambia sotto l'influenza esterna al programma, è essenziale informare di questo il compilatore, il quale non rimuoverà istruzioni che coinvolgono puntatori o altre variabili dichiarate volatili.

In un calcolatore dotato di una memoria cache, alcuni compilatori hanno un'interpretazione ulteriore per puntatori o variabili volatili. La cache è una memoria piccola e veloce che tiene copie di dati in memoria principale. Le istruzioni con riferimenti a locazioni in memoria sono eseguite più velocemente quando i dati sono disponibili nella cache. Tuttavia, dati provenienti da registri di dispositivi di I/O che usano l'unificazione degli spazi di indirizzamento non dovrebbero essere tenuti nella cache poiché il contenuto di questi registri cambia sotto l'influenza esterna. Quindi, i riferimenti a queste locazioni dovrebbero evitare la cache e accedere direttamente ai registri di I/O. La dichiarazione di puntatori a tali locazioni come volatili permette di informare un compilatore non solo per prevenire ottimizzazioni non volute, ma anche per generare istruzioni di accesso a memoria che evitano la cache, si veda l'Approfondimento applicativo 3.1.

La Figura 4.3 include costanti numeriche per i valori specifici che rappresentano posizioni di bit nei due registri di stato. Per esempio, la costante 0x2 nell'istruzione:

```
while ((*KBD_STATUS & 0x2) == 0)
```

è usata per controllare se il bit b_1 nel registro KBD_STATUS sia asserito. Questo approccio è qui usato per facilitare il confronto fra i valori dati e la specifica delle interfacce dei dispositivi in Figura 3.3. Un approccio più usuale nella scrittura di programmi C impiega istruzioni *define* per associare nomi significativi a tali valori costanti, per usare nomi nel resto del programma.

4.8 Interazioni tra linguaggio assemblativo e linguaggio C

Occasionalmente un programma potrebbe necessitare di accedere ai registri di controllo del processore, per esempio per l'inizializzazione di una routine di servizio delle interruzioni. Sulla base di istruzioni in un linguaggio di alto livello, un compilatore non può generare istruzioni in linguaggio assemblativo che accedono ai registri di controllo del processore. Poiché a tale scopo sono necessarie istruzioni in linguaggio assemblativo, il compilatore permette di avere istruzioni in linguaggio assemblativo direttamente nel programma in un linguaggio di alto livello. Questo paragrafo illustra tale approccio.

Si consideri un'attività di I/O che trasferisce caratteri da una tastiera a uno schermo, assumendo che si usino le interruzioni per ricevere caratteri dall'interfaccia della tastiera. Per mantenere semplice l'esempio si assuma anche che la routine di servizio dell'interruzione mandi ciascun carattere ricevuto direttamente all'interfaccia dello schermo senza la scansione del suo stato. Qui si sta assumendo che i caratteri siano ricevuti con una frequenza sufficientemente bassa da permettere allo schermo di gestirla.

L'inizializzazione nel programma per svolgere quest'attività richiede l'accesso ai registri di I/O e ai registri di controllo del processore. L'interfaccia di I/O in Figura 3.3 dovrebbe essere configurata per segnalare una richiesta di interruzione quando KIN = 1. Il bit di abilitazione delle interruzioni corrispondente nel registro KBD_CONT, KIE, dovrebbe essere posto a 1. Inoltre è necessario abilitare le interruzioni del processore ponendo a 1 il bit IE nel registro di stato del programma (PS) e il bit KBD nel registro di controllo IENABLE in Figura 3.7.

Il Capitolo 3 descrive diversi metodi per identificare l'indirizzo iniziale di una routine di servizio d'interruzione che deve essere eseguita quando viene segnalata una particolare interruzione. Il metodo delle interruzioni vettorizzate da diverse fonti usa locazioni di memoria predeterminate che mantengono gli indirizzi delle corrispondenti routine di servizio delle interruzioni. In questo paragrafo, si assumerà per semplicità che ci sia un solo vettore delle interruzioni, IVETT, all'indirizzo 0x20 per tutte le interruzioni. Questo vettore deve essere inizializzato con l'indirizzo della routine di servizio delle interruzioni.

La Figura 4.4 mostra un programma in linguaggio assemblativo che usa le interruzioni per leggere caratteri dalla tastiera. Il programma principale carica l'indirizzo della routine di servizio dell'interruzione nella locazione IVETT. Esso pone a 1 il bit KIN nel registro di controllo dell'interfaccia della tastiera, come pure i bit di abilitazione delle interruzioni nei registri del processore IENABLE e PS. Per ciascuna interruzione dall'interfaccia della tastiera, la routine di servizio dell'interruzione legge il carattere in ingresso e lo manda allo schermo.

Si consideri adesso l'uso di un programma C per compiere la stessa attività di I/O. Un linguaggio di alto livello come il C non è progettato per gestire caratteristiche dell'hardware come le interruzioni. Per scrivere un programma C che usa le interruzioni bisogna rispondere a due domande:

- Come si accede ai registri di controllo del processore?
- Come si scrive una routine di servizio delle interruzioni?

L'approccio delle interruzioni richiede di asserire i bit di controllo nei registri IENABLE e PS come parte dell'inizializzazione. L'approccio basato sui puntatori usato in Figura 4.3 per accedere ai registri di I/O con l'unificazione degli spazi di indirizzi di memoria e I/O non può essere usato poiché i registri di controllo IENABLE e PS non hanno indirizzi. Invece, questi registri sono accessibili includendo apposite istruzioni in linguaggio assemblativo di-


```
IVETT        EQU        0x20          Vettore per routine di servizio d'interruzione
KBD_DATA     EQU        0x4000        Registro dati della tastiera (8 bit)
KBD_STATUS   EQU        0x4004        Registro di stato della tastiera (il bit 1 è
                                       la condizione di stato KIN)
KBD_CONT     EQU        0x4008        Registro di controllo della tastiera il (bit 1 è
                                       la condizione di stato KIE)
DISP_DATA    EQU        0x4010        Registro dati dello schermo (8 bit)
DISP_STATUS  EQU        0x4014        Registro di stato dello schermo (il bit 2 è
                                       la condizione di stato DOUT)
Programma principale
MAIN:        Move       R2, #KBD_DATA Puntatore all'interfaccia della tastiera
             Move       R3, #0x2
             StoreByte  R3, 8(R2)     Configura la tastiera per segnalare
                                       interruzioni
             Move       R2, #IVETT    Puntatore al vettore delle interruzioni
             Move       R3, #INTSERV  Inizio della routine di servizio d'interruzione
             Store      R3, (R2)      Inizializza il vettore delle interruzioni
             Move       R2, #0x2      Abilita il processore a ricevere interruzioni
             MoveControl IENABLE, R2  da tastiera
             Move       R2, #0x1      Poni a 1 il bit che abilita le interruzioni
                                       per il processore
             MoveControl PS, R2
CICLO :      Branch     CICLO         Ciclo di attesa continua
Routine di servizio d'interruzione
INTSERV:     Subtract   SP, SP, #8    Salva i registri
             Store      R2, 4(SP)
             Store      R3, (SP)
             Move       R2, #KBD_DATA Puntatore all'interfaccia della tastiera
             LoadByte   R3, (R2)      Leggi il prossimo carattere
             Move       R2, #DISP_DATA Puntatore all'interfaccia dello schermo
             StoreByte  R3, (R2)      Scrivi il carattere ricevuto sullo schermo
             Load       R2, 4(SP)     Ripristina i registri
             Load       R3, (SP)
             Add        SP, SP, #8
             Return-from-interrupt
```

Figura 4.4 Programma assemblativo per trasferire caratteri usando le interruzioni.

rettamente nel programma C, il che è reso possibile da una direttiva speciale al compilatore. Per esempio, l'istruzione:

asm("MoveControl PS, R2")

fa sì che il compilatore C inserisca nel codice compilato l'istruzione in linguaggio assemblativo tra virgolette. Poiché il registro R2 potrebbe già essere usato da istruzioni generate dal compilatore, il suo contenuto non dovrà essere alterato dalle istruzioni in linguaggio assemblativo iniettate. Una semplice soluzione è salvare in pila il contenuto di R2 prima che questo

sia modificato per essere usato dall'istruzione MoveControl, quindi ripristinarlo dopo questa istruzione. Qui si userà questo approccio, ma si noti che i compilatori forniscono metodi più sofisticati per gestire l'uso dei registri specificati nelle direttive *asm*.

Il secondo problema è la routine di servizio dell'interruzione. Il linguaggio C richiede che questa routine sia scritta come funzione. Tuttavia, il compilatore traduce tutte le funzioni C come sottoprogrammi che implicitamente finiscono con un'istruzione di rientro da sottoprogramma. La Figura 4.5 fornisce un esempio. C'è una funzione principale che effettua qualche attività non specificata. La funzione chiamata *intserv* trasferisce un carattere dalla tastiera allo schermo. Il codice generato dal compilatore per la funzione *intserv* è:

```
<salva registri>
LoadByte    R2, 0x4000(R0)
StoreByte   R2, 0x4010(R0)
<ripristina registri>
Return
```

Poiché gli indirizzi dei registri di I/O entrano in 16 bit, il compilatore può usare il modo di indirizzamento assoluto, con il registro R0 che contiene sempre il valore zero, come discusso nel Paragrafo 2.4.3.

Per poter usare la funzione *intserv* come routine di servizio delle interruzioni, essa dovrà terminare con un'istruzione di rientro da interruzione. Questa istruzione è necessaria per ripristinare il contenuto del contatore del programma e il registro di stato del programma ai valori che avevano al momento in cui l'interruzione era avvenuta. Possiamo inserire un rientro da interruzione come ultima istruzione della funzione *intserv* nel programma usando l'istruzione:

```
asm("Return-from-interrupt")
```

Con questa istruzione il codice compilato per l'intera funzione sarà:

```
<salva registri>
LoadByte    R2, 0x4000(R0)
StoreByte   R2, 0x4010(R0)
Return-from-interrupt
<ripristina registri>
Return
```

Il compilatore include ancora il codice per ripristinare i registri e l'istruzione di rientro da sottoprogramma alla fine come avviene per tutte le funzioni. Comunque, l'inserimento di un'istruzione di rientro da interruzione implica che il codice successivo a essa non sarà mai eseguito. Poiché le interruzioni possono avvenire in qualsiasi punto nel programma, non riuscire a ripristinare il valore originale di un registro che viene modificato in una funzione rende incorretta l'esecuzione della parte restante del programma. Ancor più grave è il mancato ripristino del valore corretto del puntatore alla pila, che provoca errori nelle aree di attivazione di sottoprogrammi annidati.

Esistono due approcci per fornire un corretto supporto alle interruzioni in un linguaggio di alto livello come il C. Il primo richiede l'estensione della sintassi del linguaggio con una parola chiave speciale che identifica le routine di servizio delle interruzioni. Per esempio, un compilatore C può riconoscere la parola chiave *interrupt* all'inizio di una definizione di funzione, come:

```
interrupt void intserv() { ... }
```

per la funzione in Figura 4.5. Questa parola chiave istruisce il compilatore a usare l'istruzione di rientro da interruzione anziché da sottoprogramma. I registri sono sempre salvati e ripristinati. Non tutti i compilatori C forniscono però questo supporto.


```
#define   KBD_DATA     (volatile char *) 0x4000
#define   DISP_DATA    (volatile char *) 0x4010
 void main()
 {
        ⋮
 }
 void intserv()
 {
    *DISP_DATA = *KBD_DATA;   /* Trasferisci un carattere */
```

Figura 4.5 Rappresentazione di una routine di servizio d'interruzione come funzione in un programma C.

Il secondo approccio consiste nel predisporre un gestore di interruzioni in linguaggio assemblativo e usare il collegatore per collegare il gestore al programma C. In questo caso, il gestore deve prima salvare il registro di collegamento, poiché l'interruzione potrebbe avvenire dopo una chiamata a sottoprogramma nel programma principale, dopo di che il gestore dell'interruzione può chiamare un sottoprogramma in linguaggio C che serve l'interruzione. In questo modo, non sono necessarie parole chiave speciali nel file sorgente del linguaggio di alto livello. Al rientro dal sottoprogramma, il registro di collegamento viene ripristinato e si esegue l'istruzione di rientro da interruzione.

È ora possibile scrivere un programma C che usa le interruzioni per trasferire caratteri dalla tastiera allo schermo. La Figura 4.6 fornisce un possibile programma equivalente a quello di Figura 4.4. Usiamo l'approccio basato sulla parola chiave speciale interrupt per il compilatore C, poiché essa permette di avere l'intero programma in un singolo file sorgente nel linguaggio di alto livello. Si noti che i puntatori ai registri nella disposizione con unificazione degli spazi di indirizzamento di memoria e I/O sono puntatori a caratteri poiché essi puntano a locazioni che corrispondono ai registri a otto bit nelle interfacce dei dispositivi. Il puntatore IVETT è di tipo unsigned integer, poiché esso punta a una locazione di memoria che immagazzina un vettore d'interruzione di quattro byte.

4.9 Sistema operativo

I paragrafi precedenti hanno descritto come preparare ed eseguire applicazioni con l'aiuto di vari programmi di utilità. Tutti i compiti descritti in questo capitolo sono facilitati dal *sistema operativo* (SO), che è un componente software determinante nella maggior parte dei calcolatori. Esso è responsabile del coordinamento di tutte le attività in un calcolatore. Generalmente il SO consiste di routine essenziali che risiedono sempre nella memoria del calcolatore e di vari programmi di utilità che sono immagazzinati su un disco magnetico e caricati in memoria quando è necessario eseguirli.

Il SO gestisce le risorse di elaborazione, memoria e ingresso/uscita di un calcolatore durante l'esecuzione di programmi, interpreta comandi di utente, assegna memoria e spazio su disco, trasferisce informazioni fra la memoria e il disco, e gestisce operazioni di I/O; inoltre rende possibile all'utente l'uso di editori di testi, compilatori, assemblatori e collegatori, per preparare applicazioni. In genere il caricatore fa parte del SO e viene richiamato quando un utente inserisce un comando per eseguire un programma. L'obiettivo in questo paragrafo è fornire una comprensione di base delle funzioni importanti svolte dal SO. Una discussione più completa è al di là degli obiettivi di questo libro, quindi si rinvia alla bibliografia [1].

```
#define  IVECT        (volatile unsigned int *) 0x20
#define  KBD_DATA     (volatile char *) 0x4000
#define  KBD_CONT     (volatile char *) 0x4008
#define  DISP_DATA    (volatile char *) 0x4010
#define  DISP_STATUS  (volatile char *) 0x4014

 interrupt void intserv();     /* Dichiarazione anticipata di funzione */

 void main()
 {
   /* Inizializzazioni per trasferimenti di caratteri tramite interruzioni */
   *KBD_CONT = 0x2;                /* Abilita le interruzioni da tastiera */
   *IVETT = (unsigned int) &intserv;  /* Inizializza il vettore delle interruzioni */
   asm ("Subtract SP, SP, #4");    /* Salva il registro R2 */
   asm ("Store R2, (SP)");
   asm ("Move R2, #0x2");          /* Abilita il processore a ricevere interruzioni da tastiera */
   asm ("MoveControl IENABLE, R2");
   asm ("Move R2, #0x1");          /* Abilita le interruzioni per il processore */
   asm ("MoveControl PS, R2");
   asm ("Load R2, (SP)");          /* Ripristina il registro R2 */
   asm ("Add SP, SP, #4");

   while (1)                       /* Ciclo continuo */
   {
                                   /* Trasferisci i caratteri usando la routine di servizio
                                      d'interruzione */
   }
}

interrupt void intserv()  /* La parola chiave indica al compilatore di trattare la funzione come
                             routine di servizio d'interruzione */
{
   *DISP_DATA = *KBD_DATA;     /* Trasferisci un carattere */
   /* Il compilatore inserirà l'istruzione di rientro da interruzione alla fine della funzione */
}
```

Figura 4.6 Programma C per trasferire caratteri usando le interruzioni.

4.9.1 Processo di avvio

Il SO per un calcolatore di uso generale è un insieme grande e complesso di componenti software. Tutte le parti del SO, inclusa la porzione che risiede sempre in memoria, sono in genere immagazzinate sul disco. Un processo di *avvio* (*boot-strapping*) viene usato per caricare in memoria la porzione del SO residente in memoria così che la sua esecuzione possa cominciare e assumere il controllo sulle risorse del calcolatore.

Il processo di avvio inizia quando il calcolatore viene acceso e il processore preleva la prima istruzione da una locazione predeterminata. Tale locazione deve essere in una parte permanente della memoria che mantiene il suo contenuto quando il calcolatore viene spento. Un piccolo programma presente in quella locazione abilita il processore a trasferire parti progressivamente sempre più grandi del SO dal disco alla porzione di memoria che non è permanente. Ogni programma eseguito in questa sequenza di avvio trasferisce un po' di più del SO dal disco alla memoria ed effettua le necessarie inizializzazioni di memoria e I/O del calcolatore. Infine, vengono trasferiti in memoria il caricatore e la porzione di SO responsabile

dell'elaborazione dei comandi di utente. Questo abilita il SO a iniziare l'accettazione dei comandi per caricare ed eseguire applicazioni immagazzinate in file sul disco.

4.9.2 Gestione dell'esecuzione di programmi di applicazioni

Per capire i principi essenziali dei sistemi operativi, si consideri un calcolatore con un processore e alcuni dispositivi di I/O, ovvero una tastiera, uno schermo, un disco e una stampante. Si discuteranno dapprima i passi coinvolti nell'esecuzione di una singola applicazione, poi come il SO gestisce l'esecuzione di più applicazioni.

Per eseguire un programma immagazzinato in un file su disco, l'utente inserisce un comando che fa sì che il caricatore trasferisca questo file in memoria. Quando il trasferimento è completo, inizia l'esecuzione del programma. Supponendo che tra i compiti del programma ci sia dapprima il caricamento in memoria centrale di un file di dati, poi l'esecuzione di alcuni calcoli sui dati in memoria e infine la stampa dei risultati prodotti, si hanno i passi seguenti:

- raggiunto il punto dove occorre usare i dati, il programma chiama il SO richiedendo il servizio di caricamento di file da disco a memoria centrale; sospendendo il programma, il SO esegue il servizio e quando ha terminato lo riattiva;
- il programma continua eseguendo i calcoli sui dati in memoria centrale;
- completati i calcoli e raggiunto il punto in cui deve stamparne i risultati, il programma chiama il SO richiedendo il servizio di stampa; sospendendo di nuovo il programma, il SO esegue il servizio e quando ha terminato lo riattiva.

Ogniqualvolta il SO viene chiamato, il modulo componente appropriato va in esecuzione per svolgere il servizio desiderato; si comprende dunque come il controllo dell'esecuzione venga spartito a intervalli tra programma e moduli (o routine) componenti del SO. Quando si ha una successione di attività svolte da entità differenti un buon modo per rappresentarla è dato dal diagramma temporale, come quello mostrato in Figura 4.7. Eccone la spiegazione:

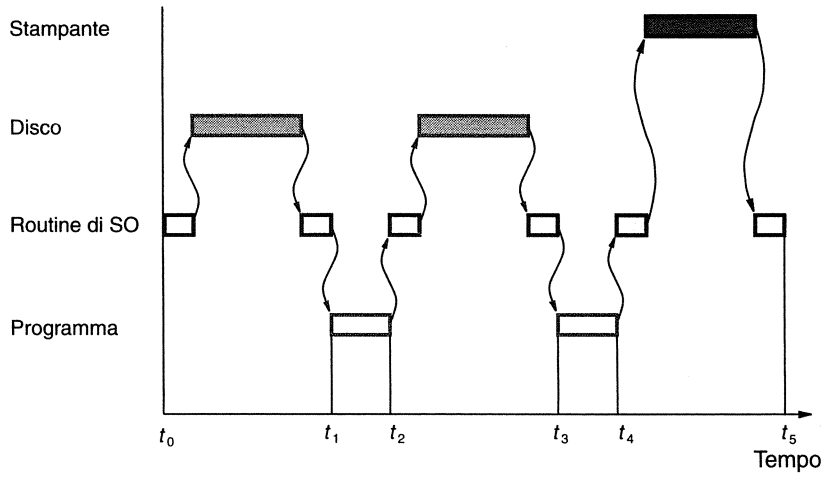


Figura 4.7 Come il programma, il SO e le unità funzionali si spartiscono il tempo.

- nell'intervallo iniziale di tempo $[t_0, t_1]$ un modulo di SO dà inizio al caricamento del programma da disco a memoria centrale, attende finché l'operazione non sia terminata e poi mette in esecuzione il programma così caricato;
- durante gli intervalli $[t_2, t_3]$ e $[t_4, t_5]$ hanno luogo attività analoghe per il caricamento del file di dati da disco a memoria centrale e la stampa dei risultati, rispettivamente;
- dopo l'istante di tempo t_5 il SO è libero di caricare e mettere in esecuzione un nuovo programma.

Si deve notare come gli intervalli di tempo $[t_0, t_1]$, $[t_2, t_3]$ e $[t_4, t_5]$ si dividano a loro volta in tempo di esecuzione del modulo del SO e tempo di attesa affinché l'unità funzionale coinvolta svolga il proprio compito.

E ora bisogna vedere come usare le risorse del calcolatore in modo efficiente quando due o più programmi vanno eseguiti in contemporanea. L'osservazione di partenza è che il processore e l'unità disco restano inattivi per la maggior parte dell'intervallo di tempo $[t_4, t_5]$; ne segue che il SO può benissimo caricare da disco a memoria il prossimo programma da eseguire mentre la stampante è ancora attiva per conto di quello precedente. In modo analogo, durante l'intervallo $[t_0, t_1]$ il SO può effettuare le operazioni di stampa di pertinenza del programma precedente mentre quello corrente è ancora in fase di caricamento da disco. Naturalmente tutto ciò si generalizza in schemi di alternanza e sovrapposizione di attività ben più complessi di questo semplice esempio. In tale modo il SO riesce a gestire l'esecuzione in contemporanea di svariati programmi, a tratti intercalandone o sovrapponendone le attività in modo da evitare interferenze indesiderate. Si tratta, beninteso, di un'attuazione parziale dell'effettiva esecuzione contemporanea, in quanto in realtà i programmi sono in attività uno alla volta a scatti; ma se gli scatti sono frequenti e brevi tutto l'insieme avanza come se davvero i programmi vengano eseguiti contemporaneamente, o almeno ciò è quanto si rileva esternamente. Programmi in esecuzione contemporanea si dicono *concorrenti* e il SO capace di gestirli in tale modo si chiama *SO concorrente* (*concurrent OS*) o *multiprogrammato* (*multitasking OS*). Per approfondimenti sul SO si rimanda alla bibliografia [1].

4.9.3 Uso delle interruzioni in un sistema operativo

Il sistema operativo fa largo uso delle interruzioni per effettuare operazioni di I/O e anche per comunicare con i programmi e controllarne l'esecuzione. Il meccanismo delle interruzioni abilita il SO ad assegnare priorità, commutare da un programma a un altro, terminare programmi, garantire caratteristiche di sicurezza e protezione, coordinare attività di I/O. Si discuteranno brevemente alcuni di questi aspetti per illustrare come sono usate le interruzioni.

Il SO incorpora le routine di servizio delle interruzioni per tutti i dispositivi collegati a un calcolatore che sono in grado di segnalare interruzioni. In un calcolatore di uso generale con un sistema operativo, le applicazioni non effettuano direttamente operazioni di I/O. Quando un'applicazione ha bisogno di un'operazione di ingresso o uscita, essa indica i dati che devono essere trasferiti e chiede al SO di effettuare l'operazione. Spesso la richiesta da parte dell'applicazione avviene tramite un sottoprogramma di libreria che segnala un'interruzione software per accedere alle routine del SO. Il SO sospende temporaneamente l'esecuzione del programma richiedente, quindi avvia l'operazione di I/O richiesta. Quando l'operazione di I/O viene completata, il SO è in genere informato di tale condizione da un'interruzione hardware, quindi permette al programma sospeso di riprendere l'esecuzione. Il SO e l'applicazione si passano il controllo a vicenda usando interruzioni software.

Il SO fornisce una varietà di servizi alle applicazioni. Per facilitare la realizzazione di questi servizi, un processore può avere diverse istruzioni di interruzione software, ognuna con il proprio vettore di interruzioni. Tali istruzioni possono essere usate per chiamare diverse parti del SO

in base al servizio che viene richiesto. In alternativa, un processore può avere soltanto un'istruzione di interruzione software, con un operando immediato che indica il servizio desiderato.

Il SO deve assicurare che l'esecuzione di un'applicazione sia terminata in modo appropriato. L'esecuzione di un'appropriata istruzione di interruzione software alla fine dell'applicazione comunica al SO che deve assumere il controllo e completare la terminazione. Si rammenti che le informazioni su locazione d'inizio e lunghezza del programma in memoria sono incluse nell'intestazione di un programma oggetto. Il SO usa queste informazioni per recuperare lo spazio precedentemente allocato al programma. Lo spazio recuperato è quindi disponibile per un'altra applicazione.

Per ottenere la multiprogrammazione, il SO accetta un nuovo comando dall'utente in qualsiasi momento. Il caricamento e l'inizio dell'esecuzione del programma richiesto avverranno quando tutte le risorse necessarie al programma saranno disponibili.

Esempio di multiprogrammazione

Per illustrare l'interazione tra applicazioni e SO, si consideri un esempio che comporta la multiprogrammazione. Una comune tecnica del SO che la rende possibile è la *divisione in quanti di tempo* (*time slicing*). Ogni programma esegue per un piccolo periodo, τ, chiamato quanto di tempo. Quindi un altro programma esegue per il suo quanto di tempo, e così via. L'intervallo τ è determinato da un cronometro hardware continuamente in esecuzione, che genera un'interruzione ogni τ secondi.

La Figura 4.8 descrive le routine necessarie a realizzare alcune delle funzioni essenziali di un ambiente multiprogrammato. Quando il sistema operativo è avviato, esegue una routine di inizializzazione, chiamata OSINIT in Figura 4.8a. Tra le altre cose, questa routine inizializza le locazioni in memoria dei vettori d'interruzione. I valori scritti nelle locazioni dei vettori sono gli indirizzi iniziali delle routine di servizio delle interruzioni per le interruzioni corrispondenti. Per esempio, OSINIT carica l'indirizzo di partenza di una routine chiamata SCHEDULER nel vettore d'interruzione corrispondente all'interruzione del cronometro. Quindi, alla fine di ogni quanto di tempo, l'interruzione del cronometro provocherà l'esecuzione di questa routine.

Un programma, insieme con le informazioni che descrivono il suo stato di esecuzione corrente, è considerato un'entità chiamata *processo*. Un processo può trovarsi in uno dei tre stati Esecuzione, Eseguibile, Bloccato (*running, runnable, blocked*). Lo stato Esecuzione significa che il programma è attualmente in esecuzione. Il processo è Eseguibile se il programma è pronto e in attesa di essere selezionato per l'esecuzione. Il terzo stato, Bloccato, significa che il programma non è pronto a riprendere l'esecuzione per qualche ragione. Per esempio, può essere in attesa del completamento di un'operazione di I/O che era stata precedentemente richiesta.

Si assuma che il programma A sia nello stato Esecuzione durante un certo quanto di tempo. Alla fine di quel quanto di tempo, il cronometro interrompe l'esecuzione di questo programma e avvia l'esecuzione di SCHEDULER. Questa è una routine del sistema operativo la cui funzione è determinare quale programma utente dovrebbe essere eseguito per il prossimo quanto di tempo. Essa comincia salvando tutte le informazioni che saranno successivamente necessarie quando l'esecuzione del programma A verrà riavviata. Le informazioni salvate includono il contenuto dei registri, compreso il contatore del programma e il registro di stato del programma. I registri devono essere salvati poiché essi possono contenere risultati intermedi di elaborazioni in corso al momento dell'interruzione. Il contatore del programma punta alla locazione da cui l'esecuzione verrà successivamente ripresa. Il registro di stato del programma riflette lo stato corrente del programma.

Quindi, lo SCHEDULER seleziona per l'esecuzione qualche altro programma, B, che sia stato precedentemente sospeso e che si trovi nello stato Eseguibile. Ripristina tutte le informazioni salvate nel momento in cui il programma B era stato sospeso, inclusi il contenuto del contatore del programma e il registro di stato, ed esegue un'istruzione di rientro da interruzione. Di conseguenza, il programma B riprende l'esecuzione per τ secondi, alla fine dei

OSINIT	Inizializza i vettori delle interruzioni: Interruzione dal cronometro ← SCHEDULER Interruzione software ← OSSERVICES Interruzione di I/O ← IODATA ⋮
OSSERVICES	Esamina la pila o registri del processore per individuare l'operazione richiesta Chiama la routine appropriata
SCHEDULER	Salva lo stato corrente del processo in esecuzione Seleziona un altro processo eseguibile Ripristina lo stato salvato del nuovo processo Rientra dall'interruzione
	(a) Inizializzazione di SO, servizi e pianificatore (scheduler)
IOINIT	Poni a Bloccato lo stato del processo richiedente Inizializza il puntatore all'indirizzo del buffer in memoria e il contatore di byte Chiama il driver del dispositivo per inizializzarlo e abilitarne le interruzioni nell'interfaccia Rientra da sottoprogramma
IODATA	Controlla i dispositivi per individuare la fonte dell'interruzione Chiama il driver appropriato Se END = 1, commuta a Eseguibile lo stato del processo bloccato da I/O Rientra da interruzione
	(b) Routine di I/O
KBDINIT	Abilita le interruzioni Rientra da sottoprogramma
KBDDATA	Controlla lo stato del dispositivo Se è pronto, trasferisci il carattere Se il carattere è CR, poni End = 1 e disabilita le interruzioni, altrimenti poni End = 0 Rientra da sottoprogramma.
	(c) Driver della tastiera

Figura 4.8 Esempi di routine del sistema operativo.

quali il cronometro segnala nuovamente un'interruzione, e avverrà un *cambio di contesto* verso un altro programma eseguibile.

Si supponga che il programma A sia al momento in esecuzione e necessiti di leggere una linea di caratteri dalla tastiera. Invece di effettuare l'operazione da se stesso, richiede il servizio di I/O al sistema operativo. Il programma A usa la pila o i registri del processore per passare al SO l'informazione che descrive l'operazione richiesta, il dispositivo di I/O e l'indirizzo di un buffer nell'area dati del programma dove i caratteri della tastiera dovranno essere immagazzinati. Quindi segnala un'interruzione software. Il corrispondente vettore dell'interruzione punta alla routine OSSERVICES in Figura 4.8a. Questa routine esamina l'informazione sulla pila o nei registri e inizia l'operazione richiesta chiamando un'appropriata routine di SO. Nell'esempio, viene chiamata IOINIT in Figura 4.8b, che è una routine generale responsabile per l'avvio di operazioni di I/O.

Mentre un'operazione di I/O è in esecuzione, il programma che l'ha richiesta non può continuare l'esecuzione. Quindi la routine IOINIT pone il processo associato con il programma A nello stato Bloccato. La routine IOINIT effettua i preparativi necessari per l'operazione

di I/O come l'inizializzazione di puntatori a indirizzi e del conteggio di byte, quindi chiama una routine che inizializza il dispositivo specifico per l'operazione di I/O richiesta.

Nella progettazione dei SO è pratica comune incapsulare tutti i componenti software di pertinenza a un particolare dispositivo di I/O in un modulo a sé stante chiamato *driver del dispositivo* (*device driver*). Tale modulo può essere facilmente aggiunto al SO o rimosso da esso. Si è assunto che il driver del dispositivo per la tastiera consista di due routine, KBDINIT e KBDDATA, come mostrato in Figura 4.8c. La routine IOINIT chiama KBDINIT, che effettua le operazioni di inizializzazione necessarie al dispositivo o al suo circuito di interfaccia. Inoltre, KBDINIT abilita le interruzioni nel circuito di interfaccia ponendo a 1 l'apposito bit nel suo registro di controllo, e quindi rientra in IOINIT, che rientra in OSSERVICES. L'interfaccia della tastiera è adesso pronta a partecipare a un'operazione di trasferimento dati. Essa genererà una richiesta di interruzione quando viene premuto un tasto.

A seguito del rientro a OSSERVICES, la routine SCHEDULER seleziona un altro programma di utente da eseguire. Ovviamente, lo schedulatore non selezionerà il programma A, poiché tale programma ha richiesto un'operazione di I/O e si trova adesso nello stato Bloccato. Invece viene selezionato il programma B o qualche altro programma nello stato Eseguibile. L'istruzione di rientro da interruzione che causa l'inizio dell'esecuzione del programma di utente selezionato riabiliterà inoltre le interruzioni nel processore caricando il contenuto salvato nel registro di stato del programma. Quindi, una richiesta di interruzione generata dall'interfaccia della tastiera sarà accettata. Il vettore d'interruzione per questa interruzione punta a una routine di SO chiamata IODATA. Poiché ci potrebbero essere tanti dispositivi che richiedono un'interruzione, IODATA inizia a scandire questi dispositivi per individuare quello che ha richiesto il servizio, quindi, chiama l'appropriato driver del dispositivo per servire la richiesta. Nell'esempio, il driver chiamato sarà KBDDATA, che trasferirà un carattere di dati. Se il carattere è il Ritorno Carrello, una condizione di stato chiamata END sarà posta a 1, per informare IODATA che l'operazione di I/O richiesta è stata completata. A questo punto, la routine IODATA cambia lo stato del processo A da Bloccato a Eseguibile, così che lo schedulatore possa selezionarlo per l'esecuzione in qualche quanto di tempo successivo.

Sommario

Il software è un fattore significativo che contribuisce alla versatilità e all'usabilità di un calcolatore. Programmi di utilità permettono all'utente di creare, eseguire e fare il debug di applicazioni. I programmatori possono fruire della flessibilità di combinare file sorgenti in linguaggi di alto livello, file sorgenti in linguaggio assemblativo e file di libreria usando il compilatore, l'assemblatore e il collegatore per generare programmi oggetto. Quando è necessario, istruzioni in linguaggio assemblativo possono essere incluse in un file sorgente in un linguaggio di alto livello.

La potenza di un calcolatore è notevolmente aumentata grazie al software del sistema operativo, che gestisce e coordina tutte le attività. La multiprogrammazione del sistema operativo permette a differenti attività di procedere in concorrenza per più applicazioni, ottenendo quindi il miglior uso del calcolatore.

Note bibliografiche

1. Silbershatz A., Galvin P.B., Gagne G., *Sistemi Operativi – Concetti ed esempi*, 8/Ed., Pearson Italia, Milano, 2009.

5 Struttura di base del processore

In questo capitolo

- Esecuzione delle istruzioni tramite un processore
- Unità funzionali di un processore e loro interconnessioni
- Hardware per generare segnali di controllo
- Controllo microprogrammato

Il capitolo è dedicato all'*unità centrale di elaborazione* o CPU (*central processing unit*), chiamata più brevemente *processore* (*processor*). In realtà oggigiorno non pochi calcolatori contengono diverse unità di elaborazione o processori con compiti variamente specializzati, e pertanto la denonominazione "unità centrale" non è più così calzante come in passato. Comunque qui si illustrerà la struttura interna del processore e si vedrà come esso svolga le operazioni di prelievo, decodifica ed esecuzione di istruzioni macchina.

La struttura e il funzionamento del processore sono andati modificandosi sensibilmente nel corso degli anni, a seguito dei mutamenti tecnologici e della spinta all'aumento delle prestazioni. Un tratto comune alle numerose strategie ideate per potenziare il processore consiste nel fare lavorare il più possibile in parallelo le varie unità funzionali componenti. Per esempio spesso il processore funziona in tipo *pipeline*, perciò è suddiviso in stadi, perché così si riesce ad avviare l'esecuzione di un'istruzione macchina ancora prima di avere terminato quella precedente. Oppure il processore ha struttura superscalare per potere avviare in parallelo due o anche più istruzioni macchina. Le metodologie di *pipelining* e quella *superscalare* saranno trattate nel Capitolo 6. Per cominciare nel capitolo presente si considerano solo le caratteristiche di base comuni a tutti i processori moderni.

5.1 Alcuni concetti fondamentali

Un tipico lavoro di elaborazione consiste di una serie di operazioni specificate da una sequenza di istruzioni in linguaggio macchina che costituiscono un programma. Quando il processore è attivo, preleva dalla memoria un'istruzione macchina alla volta e la pone in esecuzione. Il prelievo delle istruzioni va in ordine di indirizzi consecutivi, ovvero è sequenziale, tranne quando si incontra un'istruzione di salto. Il registro *contatore di programma* o PC (*program counter*) contiene in ogni istante il puntatore alla parola di memoria dove si trova la prossima istruzione da prelevare. Ogni istruzione definisce come aggiornare il registro PC individuando l'istruzione che le deve succedere. Di solito l'aggiornamento del PC va in sequenza, ma come si è accennato prima un'istruzione di salto è più articolata e comporta una modifica meno banale del contenuto del PC.

Un altro componente chiave del processore è il *registro di istruzione* o IR (*instruction register*). Per esempio, si supponga che ogni istruzione macchina sia codificata (in forma nu-

merica) mediante una parola di quattro byte, pari alla dimensione di parola di memoria, come nell'architettura dell'insieme di istruzioni in stile RISC. Per eseguire una tale istruzione generica il processore deve attuare in successione i tre passi seguenti:

1. prelievo della parola di memoria puntata dal registro PC. Tale parola di memoria codifica l'istruzione da eseguire e viene pertanto scritta nel registro IR interno al processore. Ecco l'operazione in formalismo RTN:

$$IR \leftarrow [[PC]]$$

2. incremento di quattro unità del contenuto del PC, supponendo che la memoria sia indirizzabile per byte, vale a dire l'operazione seguente (in formalismo RTN):

$$PC \leftarrow [PC] + 4$$

3. esecuzione dell'istruzione prelevata, effettuandone le operazioni.

Spesso i primi due passi sono chiamati collettivamente *fase di prelievo* (*fetch phase*), mentre il terzo costituisce la *fase di esecuzione* (*execution phase*).

Tranne poche eccezioni, l'operazione specificata da un'istruzione può essere realizzata effettuando una o più delle seguenti azioni:

- lettura del contenuto di una data locazione di memoria e caricamento in un registro del processore;
- lettura dei dati da uno o più registri del processore;
- esecuzione di un'operazione aritmetica o logica e collocamento del risultato in un registro del processore;
- immagazzinamento di dati da un registro del processore in una certa locazione di memoria.

I componenti hardware necessari per effettuare queste azioni sono mostrati in Figura 5.1. Il processore comunica con la memoria attraverso l'interfaccia processore-memoria, che trasferisce dati da e verso la memoria durante operazioni Read e Write. Il generatore di indirizzi delle istruzioni aggiorna il contenuto del PC dopo avere prelevato ciascuna istruzione. Il banco di registri (*register file*) è un'unità di memoria le cui locazioni di memoria sono organizzate in modo da formare i registri di uso generale del processore. Durante l'esecuzione, i contenuti dei registri menzionati in un'istruzione che effettua un'operazione aritmetica o logica vengono inviati all'unità aritmetica e logica o ALU (*arithmetic and logic unit*), che effettua l'elaborazione richiesta. I risultati dell'elaborazione vengono immagazzinati in un registro nel banco di registri.

Prima di esaminare in dettaglio queste unità e le loro interazioni, è opportuno considerare la struttura generale di un qualunque sistema di elaborazione dati.

Hardware per elaborazione di dati

Un tipico calcolo opera su dati immagazzinati nei registri. Questi dati vengono elaborati da circuiti combinatori, quali i sommatori (*adder*), e i risultati vengono posti in un registro. La Figura 5.2 illustra questa struttura. Un segnale dal clock è usato per controllare la temporizzazione del trasferimento di dati. I registri consistono di flip-flop a commutazione sul fronte nei quali i nuovi dati vengono caricati sul fronte attivo del clock. In questo capitolo, si assume che il fronte di salita del clock sia il fronte attivo (si veda l'Appendice A). Il periodo di clock, che è il tempo tra due successivi fronti di salita, deve essere sufficientemente lungo da permettere al circuito combinatorio di produrre il risultato corretto.

L'operazione effettuata dal blocco combinatorio in Figura 5.2 può essere abbastanza complicata. Essa può essere spesso suddivisa in parecchi passi più semplici, in cui ogni passo è effettuato da un sottocircuito del circuito originale. Questi sottocircuiti possono essere quindi

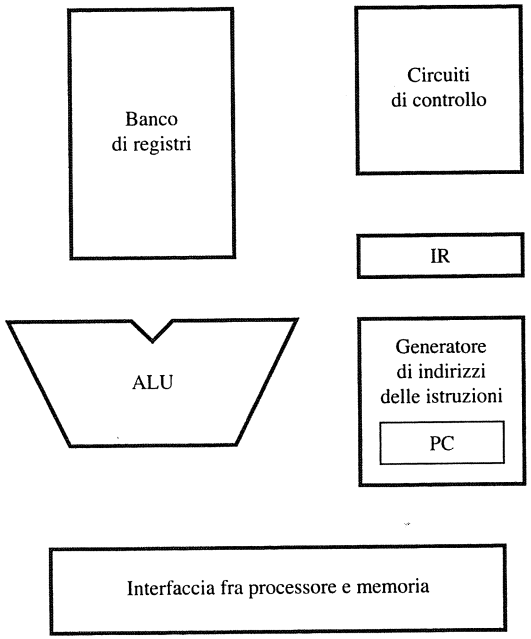


Figura 5.1 Principali componenti hardware di un processore.

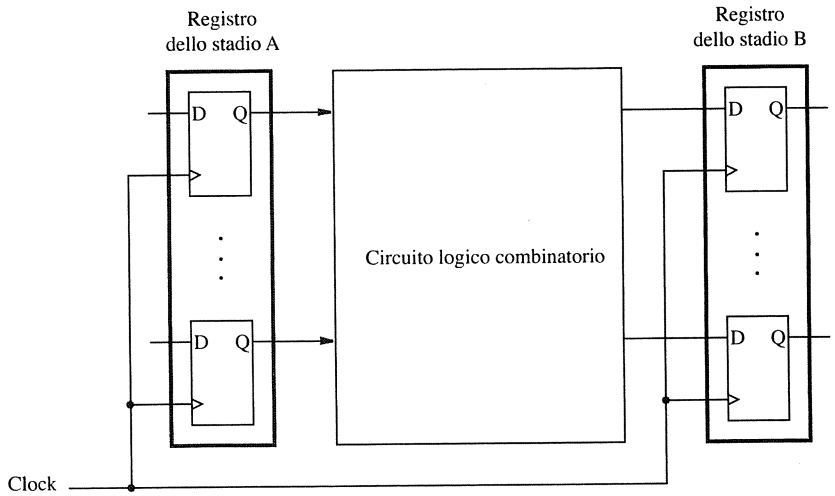


Figura 5.2 Struttura di base per l'elaborazione di dati.

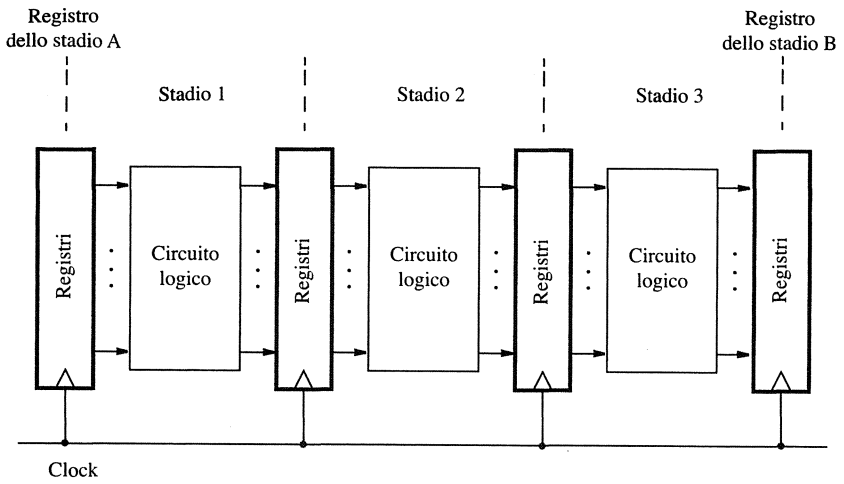


Figura 5.3 Una struttura hardware a più stadi.

usati in cascata in una struttura a più stadi, come mostrato in Figura 5.3. Quindi, se vengono usati *n* stadi, l'operazione sarà completata dopo *n* cicli di clock. Poiché questi sottocircuiti combinatori sono più piccoli, essi possono completare la loro operazione in meno tempo, quindi può essere usato un periodo di clock più corto. Un vantaggio significativo della struttura a più stadi è che si adatta a funzionare in *pipeline*, come sarà discusso nel Capitolo 6. Tale struttura è particolarmente utile per realizzare processori che hanno un insieme di istruzioni in stile RISC. La discussione nel resto di questo capitolo si concentrerà sui processori che usano una struttura a più stadi di questo tipo. Nel Paragrafo 5.7 si analizzerà un'alternativa più tradizionale che è più adatta ai processori in stile CISC.

5.2 Esecuzione di istruzioni

Si esaminino adesso le azioni coinvolte nel prelievo ed esecuzione di istruzioni. Si illustrano queste azioni usando alcune istruzioni rappresentative dello stile RISC.

5.2.1 Istruzioni di caricamento

Si consideri l'istruzione:

Load R5, X(R7)

che usa il modo di indirizzamento con indice e spiazzamento per caricare una parola di dati dalla locazione di memoria X + [R7] nel registro R5. L'esecuzione di questa istruzione coinvolge le seguenti azioni:

- prelievo dell'istruzione dalla memoria;
- incremento del contatore del programma;
- decodifica dell'istruzione per determinare quale operazione effettuare;

- lettura del registro R7;
- somma del valore immediato X al contenuto di R7;
- uso della somma X + [R7] come indirizzo effettivo dell'operando sorgente e lettura del contenuto di quella locazione di memoria;
- caricamento dei dati ricevuti dalla memoria nel registro di destinazione, R5.

A seconda di come l'hardware è organizzato, alcune di queste azioni possono essere svolte allo stesso tempo. Nella discussione che segue, si assumerà che il processore abbia cinque stadi hardware, un'organizzazione comunemente usata nei processori in stile RISC. L'esecuzione di ciascuna istruzione è suddivisa in cinque passi, così che ciascun passo sia effettuato da uno stadio hardware. In questo caso, il prelievo e l'esecuzione della suddetta istruzione Load può essere completato come segue:

1. prelievo dell'istruzione e incremento del contatore del programma;
2. decodifica dell'istruzione e lettura del contenuto del registro R7 dal banco di registri;
3. calcolo dell'indirizzo effettivo;
4. lettura dell'operando sorgente dalla memoria;
5. caricamento dell'operando nel registro di destinazione R5.

5.2.2 Istruzioni aritmetiche e logiche

Le istruzioni che coinvolgono un'operazione aritmetica o logica possono essere eseguite usando passi simili. Esse differiscono dall'istruzione Load per due aspetti:

- vi sono due registri sorgenti, o un registro sorgente e un operando sorgente immediato;
- non è richiesto l'accesso a operandi in memoria.

Una tipica istruzione di questo genere è

Add R3, R4, R5

Essa richiede i seguenti passi:

1. prelievo dell'istruzione e incremento del contatore del programma;
2. decodifica dell'istruzione e lettura dei contenuti dei registri sorgenti R4 e R5;
3. calcolo della somma [R4] + [R5];
4. caricamento del risultato nel registro di destinazione, R3.

L'istruzione Add non richiede l'accesso a un operando in memoria, quindi potrebbe essere completata in quattro passi anziché nei cinque passi necessari per l'istruzione Load. Tuttavia, come si vedrà nel prossimo capitolo, è vantaggioso usare lo stesso hardware di elaborazione a più stadi per quante più istruzioni è possibile. Questo può essere realizzato se facciamo sì che tutte le istruzioni siano eseguite nello stesso numero di passi. A questo scopo, l'istruzione Add dovrebbe essere estesa a cinque passi, analogamente ai passi dell'istruzione Load. Poiché non viene richiesto l'accesso a operandi in memoria, si può inserire un passo in cui non viene svolta nessuna azione fra i suddetti passi 3 e 4. L'istruzione Add sarebbe quindi effettuata come segue:

1. prelievo dell'istruzione e incremento del contatore del programma;
2. decodifica dell'istruzione e lettura dei registri R4 e R5;
3. calcolo della somma [R4] + [R5];
4. nessuna azione;
5. caricamento del risultato nel registro di destinazione R3.

Se l'istruzione usa un operando immediato, come in

Add R3, R4, #1000

il valore immediato verrà fornito nella parola dell'istruzione. Una volta che l'istruzione venga caricata nell'IR, il valore immediato è disponibile per essere usato nell'operazione di somma. La stessa sequenza di cinque passi può essere usata, con i passi 2 e 3 modificati in:

2. decodifica dell'istruzione e lettura del registro R4;
3. calcolo della somma [R4] + 1000.

5.2.3 Istruzioni di immagazzinamento

La sequenza di cinque passi usata per le istruzioni Load e Add è inoltre adatta per le istruzioni Store, eccetto che per l'ultimo passo di caricamento del risultato in un registro di destinazione che non è richiesto. Lo stadio hardware responsabile di questo passo non effettua nessuna azione. Per esempio, l'istruzione

Store R6, X(R8)

immagazzina il contenuto del registro R6 nella locazione di memoria X + [R8]. Essa può essere realizzata come segue:

1. prelievo dell'istruzione e incremento del contatore del programma;
2. decodifica dell'istruzione e lettura dei registri R6 e R8;
3. calcolo dell'indirizzo effettivo X + [R8];
4. immagazzinamento del contenuto del registro R6 nella locazione di memoria X + [R8];
5. nessuna azione.

Dopo aver letto il registro R8 nel passo 2, viene calcolato l'indirizzo di memoria nel passo 3 usando il valore immediato, X, nell'IR. Nel passo 4, il contenuto di R6 viene inviato in memoria per essere immagazzinato. Nessuna azione viene effettuata nel passo 5.

Riassumendo, la sequenza di cinque passi esibita in Figura 5.4 è adatta a tutte le istruzioni di un insieme in stile RISC. Le istruzioni in stile RISC occupano una parola e solo le istruzioni Load e Store accedono a operandi in memoria, come spiegato nel Capitolo 2. Le istruzioni che effettuano calcoli usano dati che sono collocati nei registri di uso generale o forniti come valori immediati nell'istruzione.

La sequenza di cinque passi è adatta a tutte le istruzioni Load e Store, poiché i modi di indirizzamento che possono essere usati in queste istruzioni sono casi speciali del modo con indice e spiazzamento. La maggior parte dei processori in stile RISC fornisce un registro di uso generale, di solito il registro R0, che contiene sempre il valore zero. Quando R0 viene usato come registro indice, l'indirizzo effettivo dell'operando è il valore immediato X. Questo

Passo	Azione
1	Preleva un'istruzione e incrementa il contatore del programma
2	Decodifica l'istruzione e leggi registri dal banco di registri
3	Esegui un'operazione dell'ALU
4	Leggi o scrivi dati in memoria se l'istruzione coinvolge un operando in memoria
5	Scrivi il risultato nel registro di destinazione, se necessario

Figura 5.4 Una sequenza di azioni in cinque passi per prelevare ed eseguire un'istruzione.

è il modo di indirizzamento assoluto. In alternativa, se lo spiazzamento X è posto a zero, l'indirizzo effettivo è il contenuto del registro indice R*i*. Questo è il modo di indirizzamento indiretto. Quindi, solo un modo di indirizzamento, il modo con indice e spiazzamento, deve essere realizzato, di conseguenza l'hardware del processore risulta notevolmente semplificato. Il compito di selezionare R0 come registro indice o di azzerare X viene lasciato all'assemblatore o al compilatore. Questo è in accordo alla filosofia RISC che tende ad avere hardware semplice e veloce a discapito di una più alta complessità del compilatore e tempi di compilazione più lunghi. Il risultato è un guadagno nel tempo necessario per effettuare varie attività in un calcolatore, poiché i programmi vengono compilati meno frequentemente di quanto essi vengano eseguiti.

5.3 Componenti hardware

La precedente discussione indica che tutte le istruzioni di un processore in stile RISC possono essere eseguite usando la sequenza di cinque passi della Figura 5.4. Quindi l'hardware del processore può essere organizzato in cinque stadi, cosicché ogni stadio effettua le azioni necessarie in uno dei passi. Si esaminino ora le componenti in Figura 5.1 per vedere come esse possano essere organizzate nella struttura a più stadi della Figura 5.3.

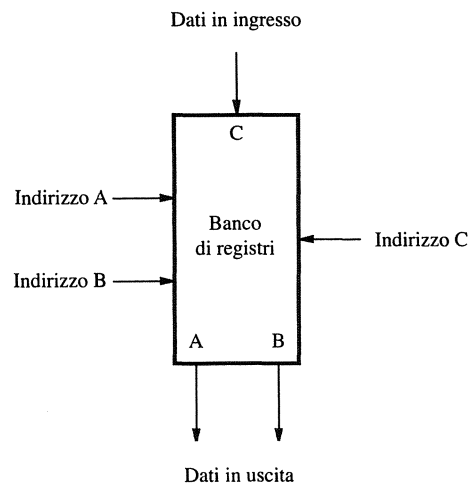
5.3.1 Banco di registri

I registri di uso generale sono in genere realizzati come un banco di registri, ovvero un blocco di memoria piccolo e veloce. Esso consiste di una serie di elementi di memorizzazione, con un circuito per l'accesso che permette la lettura e la scrittura di dati in qualsiasi registro. Il circuito per l'accesso è progettato per consentire la lettura simultanea di due registri, rendendo i contenuti disponibili su due uscite separate, A e B. Il banco di registri ha due ingressi per gli indirizzi che selezionano i due registri da leggere. Questi ingressi sono collegati ai campi dell'IR che specificano i registri sorgenti, così che i registri richiesti possano essere letti. Il banco di registri ha anche un ingresso per i dati (C) e un corrispondente ingresso per l'indirizzo che seleziona il registro in cui i dati devono essere scritti. Questo ingresso per l'indirizzo è collegato al campo di IR che specifica il registro di destinazione dell'istruzione.

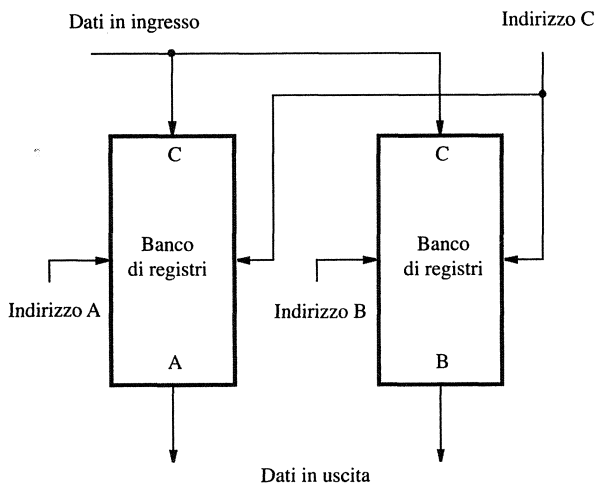
Gli ingressi e le uscite di un'unità di memoria sono spesso chiamate *porte* di ingresso e uscita. Un'unità di memoria che ha due porte di uscita viene detta *a doppia porta* (*dual-ported*). La Figura 5.5 mostra due modi per realizzare un banco di registri a doppia porta. Una possibilità è usare un singolo insieme di registri e duplicare sia i percorsi dei dati sia i circuiti di accesso per permettere a due registri di essere letti simultaneamente. Un'alternativa è usare due blocchi di memoria, ognuno contenente una copia del banco di registri. Quando si scrive un dato in un registro, esso viene scritto in entrambe le copie del registro. Quindi i due banchi hanno contenuti identici. Quando un'istruzione richiede dati da due registri, si accede a un registro in ciascuno dei banchi. In pratica, insieme i due banchi di registri funzionano come un singolo banco di registri a doppia porta.

5.3.2 ALU

L'unità aritmetica e logica viene usata per manipolare dati. Essa effettua operazioni aritmetiche come addizione e sottrazione e operazioni logiche come AND, OR e XOR. Concettualmente, il banco di registri e l'ALU possono essere collegati come mostrato in Figura 5.6. Quando un'istruzione che effettua un'operazione aritmetica o logica è in esecuzione, i contenuti dei



(a) Blocco di memoria singolo



(b) Due blocchi di memoria

Figura 5.5 Due alternative per realizzare un banco di registri a doppia porta.

due registri specificati nell'istruzione vengono letti dal banco di registri e diventano disponibili alle uscite A e B. L'uscita A è collegata direttamente al primo ingresso dell'ALU, InA, e l'uscita B è collegata a un multiplatore (*multiplexer*), MuxB. Il multiplatore seleziona l'uscita B del banco di registri o il valore immediato nell'IR da collegare al secondo ingresso dell'ALU,

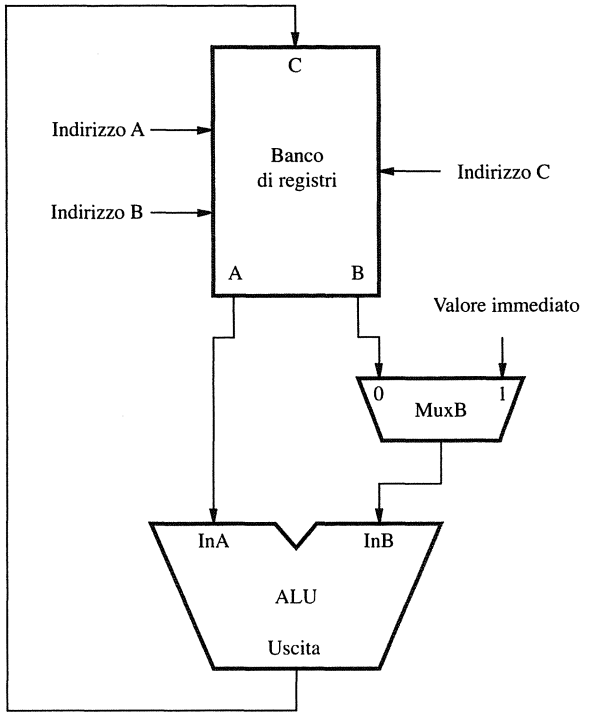


Figura 5.6 Schema concettuale dell'hardware necessario per il calcolo.

InB. L'uscita dell'ALU è collegata all'ingresso dati C del banco di registri, cosicché i risultati di un calcolo possano essere caricati nel registro di destinazione.

5.3.3 Percorso dati

L'elaborazione delle istruzioni consiste di due fasi: la fase di prelievo e la fase di esecuzione. È conveniente suddividere l'hardware del processore in due sezioni corrispondenti. Una sezione preleva istruzioni e l'altra le esegue. La sezione che preleva istruzioni è anche responsabile per la loro decodifica e per la generazione dei segnali di controllo che fanno svolgere azioni appropriate nella sezione di esecuzione. La sezione di esecuzione legge gli operandi dati specificati in un'istruzione, effettua i calcoli richiesti e immagazzina i risultati.

Adesso si deve organizzare l'hardware in una struttura a più stadi simile a quella in Figura 5.3, con stadi che corrispondono ai cinque passi in Figura 5.4. Una possibile struttura è mostrata in Figura 5.7. Le azioni svolte in ciascuno dei cinque stadi vengono completate in un ciclo di clock. Un'istruzione viene prelevata nel passo 1 dallo stadio hardware 1 e posta nell'IR. Essa viene decodificata e i suoi registri sorgenti vengono letti nel passo 2. L'informazione in IR viene usata per generare i segnali di controllo per tutti i successivi passi. Quindi IR deve continuare a tenere l'istruzione fino a quando la sua esecuzione viene completata.

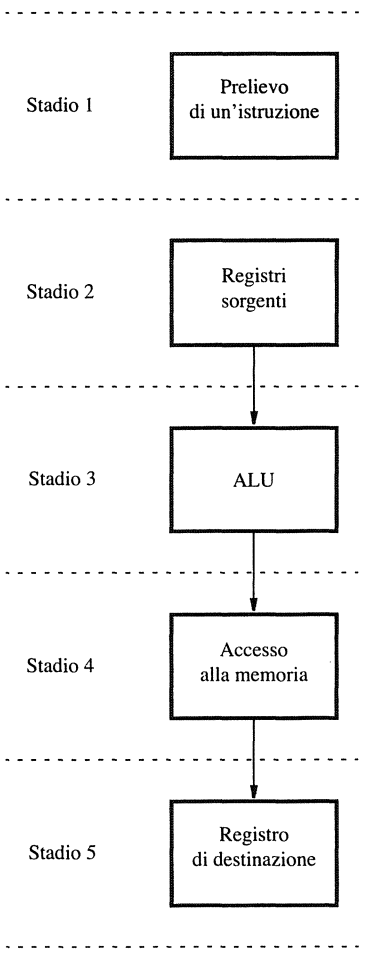


Figura 5.7 Un'organizzazione a cinque stadi.

Sarà necessario inserire registri fra stadi. I registri *interstadi* (*inter-stage register*) tengono i risultati prodotti da uno stadio cosicché essi possano essere usati come ingressi per il prossimo stadio durante il ciclo di clock successivo. Questo porta all'organizzazione in Figura 5.8. L'hardware in figura è spesso chiamato *percorso dati* (*datapath*). Esso corrisponde agli stadi da 2 a 5 in Figura 5.7. I dati letti dal banco di registri vengono posti nei registri RA e RB. Il registro RA fornisce i dati all'ingresso InA dell'ALU. Il multiplatore MuxB invia il contenuto di RB o il valore immediato in IR al secondo ingresso dell'ALU, InB. L'ALU costituisce lo stadio 3 e il risultato del calcolo effettuato dall'ALU viene posto nel registro RZ.

Si rammenti che per le istruzioni di calcolo, come un'istruzione Add, nessuna elaborazione sarà effettuata al passo 4. Durante tale passo, il multiplatore MuxY in Figura 5.8 seleziona il registro RZ per trasferire il risultato del calcolo a RY. Il contenuto di RY viene

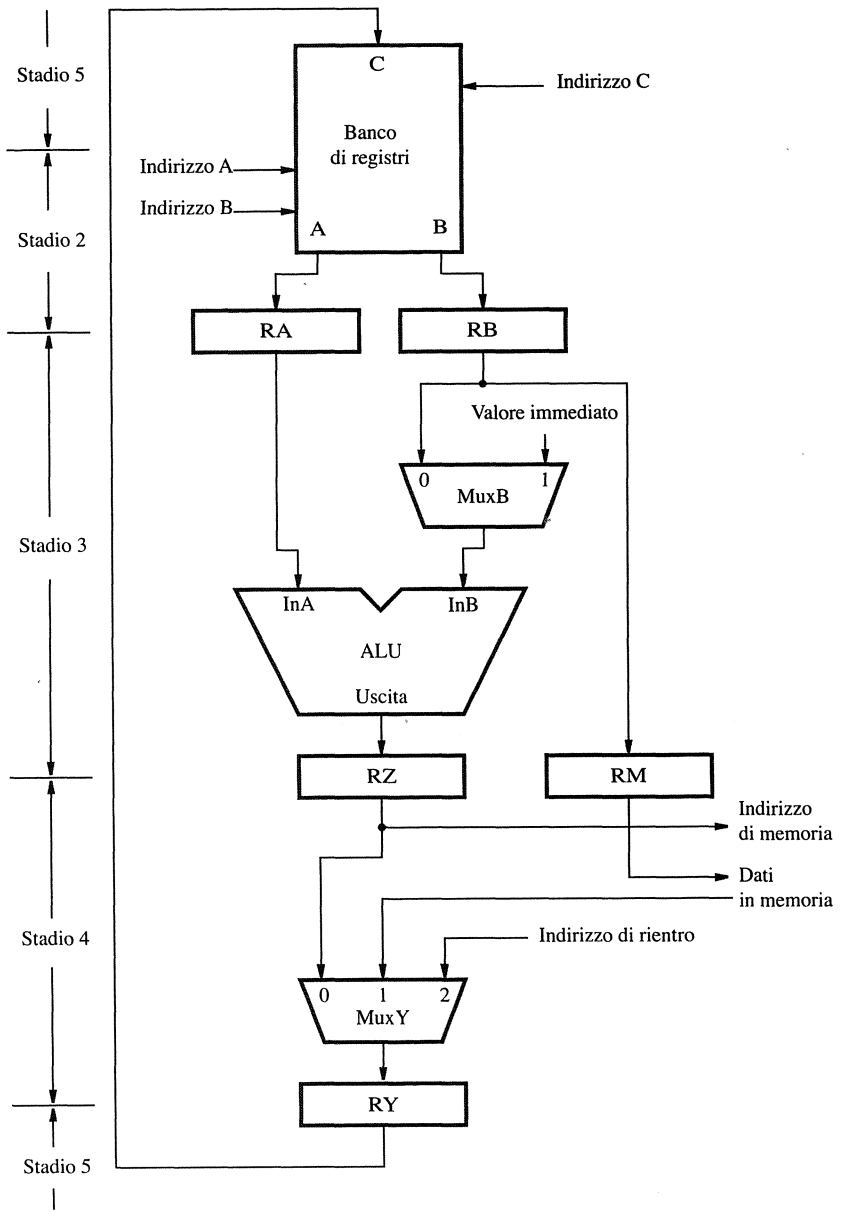


Figura 5.8 Percorso dati in un processore.

trasferito al banco di registri nel passo 5 e caricato nel registro destinazione. Per questo motivo, il banco di registri è in entrambi gli stadi 2 e 5. È parte dello stadio 2 poiché contiene i registri sorgenti ed è parte dello stadio 5 poiché contiene il registro destinazione.

Per le istruzioni Load e Store, l'indirizzo effettivo dell'operando in memoria viene calcolato dall'ALU nel passo 3 e caricato nel registro RZ. Da lì, viene inviato alla memoria, nel passo 4. Nel caso di un'istruzione Load, i dati letti dalla memoria vengono selezionati dal multiplatore MuxY e posti nel registro RY, per essere trasferiti al banco di registri nel ciclo di clock successivo. Per un'istruzione Store, i dati sono letti dal banco di registri, che è parte dello stadio 2, e posti nel registro RB. Poiché l'accesso alla memoria viene effettuato nello stadio 4, è necessario un altro registro interstadio per mantenere il corretto flusso di dati nella struttura a più stadi. Il registro RM viene introdotto a questo scopo. I dati da immagazzinare sono trasferiti da RB a RM nel passo 3, quindi alla memoria nel passo 4. Nessuna azione viene svolta nel passo 5 in questo caso.

Le istruzioni di chiamata a sottoprogramma introdotte nel Paragrafo 2.7 salvano l'indirizzo di rientro in un registro di uso generale, che si chiamerà LINK per semplicità. Analogamente, l'elaborazione delle interruzioni necessita di salvare un indirizzo di rientro, come descritto nel Paragrafo 3.2. Si assume che un altro registro di uso generale, IRA, sia usato per questo scopo. Entrambe queste azioni richiedono che il contenuto del contatore del programma sia inviato al banco di registri. Per questo motivo, il multiplatore MuxY ha un terzo ingresso attraverso cui l'indirizzo di rientro può essere diretto verso il registro RY, da dove esso può essere inviato al banco di registri. L'indirizzo di rientro viene prodotto dal generatore di indirizzi delle istruzioni, come sarà spiegato più avanti.

5.3.4 Sezione di prelievo delle istruzioni

L'organizzazione della sezione del processore dedita al prelievo delle istruzioni è illustrata in Figura 5.9. Gli indirizzi usati per accedere alla memoria provengono dal PC, quando si prelevano istruzioni, e dal registro RZ nel percorso dati quando si accede agli operandi delle

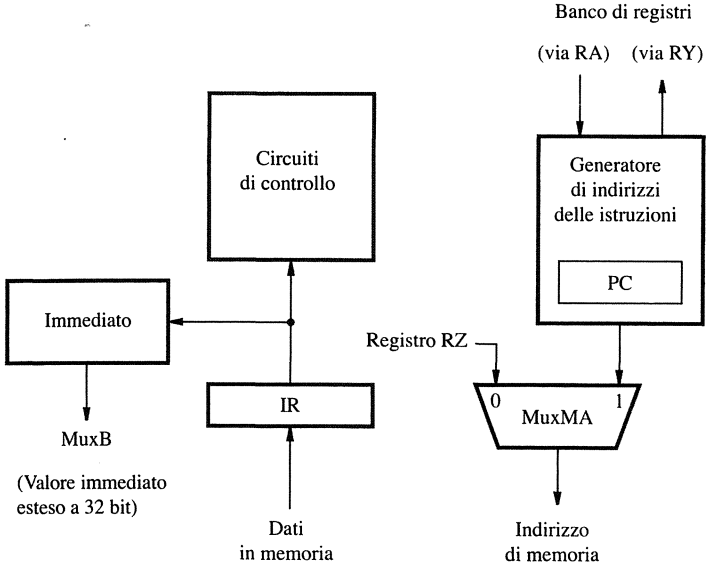


Figura 5.9 Sezione di prelievo dell'istruzione in Figura 5.7.

istruzioni. Il multiplatore MuxMA seleziona una di queste sorgenti per inviarla all'interfaccia fra processore e memoria. Il PC è incluso all'interno di un blocco più grande, il generatore di indirizzi delle istruzioni, che aggiorna il contenuto del PC dopo che ogni istruzione sia stata prelevata. L'istruzione letta dalla memoria viene caricata nell'IR, dove rimane fino a quando la sua esecuzione venga completata e la prossima istruzione venga prelevata.

Il contenuto dell'IR viene esaminato dal circuito di controllo per generare i segnali necessari a controllare tutto l'hardware del processore. Tale contenuto è anche usato dal blocco etichettato Immediato. Come descritto nel Capitolo 2, un valore immediato può essere incluso in alcune istruzioni. Un valore immediato a 16 bit viene esteso a 32 bit. Il valore esteso è quindi usato direttamente come operando o per calcolare l'indirizzo effettivo di un operando. Per alcune istruzioni, come quelle che effettuano operazioni aritmetiche, al valore immediato viene applicata l'estensione del segno; per altre, come le istruzioni logiche, viene riempito con zeri. Il blocco Immediato in Figura 5.9 genera il valore esteso e lo inoltra a MuxB nella Figura 5.8 per essere usato in un calcolo dell'ALU. Inoltre, esso genera il valore esteso che viene usato per calcolare l'indirizzo di destinazione in istruzioni di salto.

Il circuito di generazione degli indirizzi è mostrato in Figura 5.10. Un sommatore viene usato per incrementare il PC di 4 durante l'esecuzione sequenziale. Esso è anche usato per calcolare un nuovo valore da caricare nel PC quando si eseguono istruzioni di salto e chiamate a sottoprogrammi. Un ingresso del sommatore è collegato al PC. Il secondo ingresso è collegato a un multiplatore, MuxINC, che seleziona la costante 4 o lo spiazzamento da aggiungere al PC per il salto. Lo spiazzamento per il salto è fornito nel campo immediato dell'IR. Il blocco Immediato in Figura 5.9 gli applica l'estensione del segno a 32 bit. L'uscita del sommatore è instradata al PC attraverso un secondo multiplatore, MuxPC, che seleziona tra il sommatore e l'uscita del registro RA. Quest'ultima connessione è necessaria quando si eseguono istruzioni di collegamento a sottoprogrammi. Il registro PC-Temp è necessario per mantenere temporaneamente il contenuto del PC durante il processo di salvataggio dell'indirizzo di rientro da sottoprogramma o da interruzione.

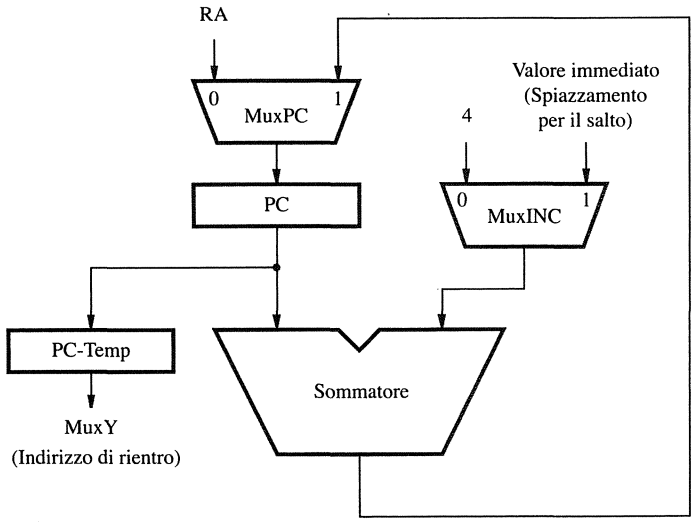


Figura 5.10 Generatore di indirizzi delle istruzioni.

5.4 Prelievo di istruzioni e passi di esecuzione

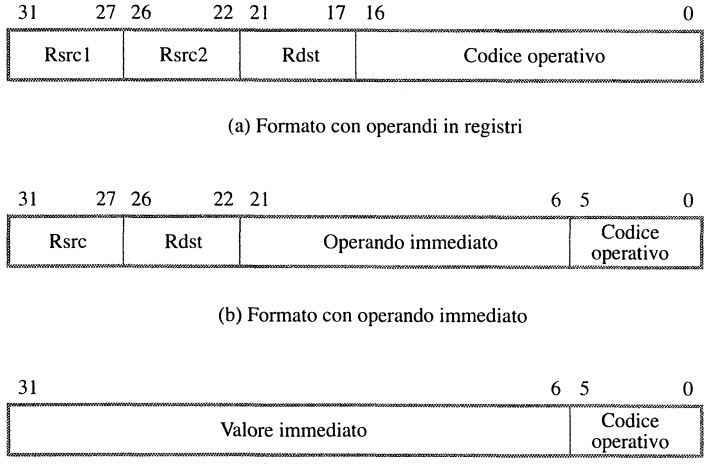
Si esamina adesso il processo di prelievo ed esecuzione di istruzioni in maggior dettaglio, usando il percorso dati in Figura 5.8. Si consideri nuovamente l'istruzione

Add R3, R4, R5

I passi per prelevare ed eseguire questa istruzione sono elencati in Figura 5.11. Si assuma che l'istruzione sia codificata usando il formato in Figura 2.32, che è riprodotto qui come Figura 5.12. Dopo che l'istruzione sia stata prelevata dalla memoria e posta nell'IR, gli indirizzi dei registri sorgenti sono disponibili nei campi IR_{31-27} e IR_{26-22}. Questi due campi sono collegati agli ingressi degli indirizzi per le porte A e B del banco di registri. Di conseguenza, i registri R4 e R5 vengono letti e i loro contenuti posti rispettivamente nei registri RA e RB, alla fine del passo 2. Nel passo successivo, il circuito di controllo fa sì che MuxB selezioni l'ingresso 0, quindi connettendo il registro RB all'ingresso InB dell'ALU. Allo stesso tempo, questo provoca l'esecuzione da parte dell'ALU dell'operazione di somma. Poiché il registro

Passo	Azione
1	Indirizzo di memoria ← [PC], Leggi memoria, IR ← Dati da memoria, PC ← [PC] + 4
2	Decodifica istruzione, RA ← [R4], RB ← [R5]
3	RZ ← [RA] + [RB]
4	R3 ← [RY]

Figura 5.11 Sequenza di azioni necessarie per prelevare ed eseguire l'istruzione Add R3, R4, R5.



(a) Formato con operandi in registri

(b) Formato con operando immediato

(c) Formato chiamata

Figura 5.12 Codifica di istruzioni.

RA è collegato all'ingresso InA, l'ALU produce la somma richiesta [RA] + [RB], che viene caricata nel registro RZ alla fine del passo 3.

Nel passo 4, il multiplatore MuxY seleziona l'ingresso 0, facendo sì che il contenuto di RZ sia trasferito a RY. Il circuito di controllo collega il campo dell'indirizzo di destinazione dell'istruzione Add, IR_{21-17}, all'ingresso dell'indirizzo per la porta C del banco di registri. Nel passo 5, emette un comando Write al banco di registri, provocando la scrittura del contenuto del registro RY nel registro R3.

Le istruzioni Load e Store vengono eseguite in modo simile. In questo caso l'indirizzo del registro di destinazione è dato dai bit IR_{26-22}. L'hardware di controllo connette questo campo all'ingresso dell'indirizzo corrispondente alla porta C del banco di registri. I passi coinvolti nell'esecuzione di queste istruzioni sono elencati nelle Figure 5.13 e 5.14. In entrambi gli esempi, l'indirizzo di memoria è specificato usando il modo con indice e spiazzamento, in cui il valore X dello spiazzamento è fornito come valore immediato nell'istruzione. Il campo immediato di IR, opportunamente esteso dal blocco Immediato in Figura 5.9, viene selezionato da MuxB nel passo 3 e aggiunto al contenuto del registro RA. La somma risultante è l'indirizzo effettivo dell'operando.

Alcune osservazioni

Nella precedente discussione, si è assunto che le operazioni di Read e Write in memoria possano essere completate in un ciclo di clock. È un'assunzione realistica? In generale, l'accesso alla memoria principale di un calcolatore richiede notevolmente più tempo che la lettura del contenuto di un registro del banco di registri. Tuttavia, buona parte dei processori moderni usano le memorie cache, che saranno discusse in dettaglio nel Capitolo 8. Una memoria cache è molto più veloce della memoria principale. In genere è realizzata sullo stesso chip del pro-

Passo	Azione
1	Indirizzo di memoria ← [PC], Leggi memoria, IR ← Dati da memoria, PC ← [PC] + 4
2	Decodifica istruzione, RA ← [R7]
3	RZ ← [RA] + Valore immediato X
4	Indirizzo di memoria ← [RZ], Leggi memoria, RY ← Dati da memoria
5	R5 ← [RY]

Figura 5.13 Sequenza di azioni necessarie per prelevare ed eseguire l'istruzione Load R5, X (R7).

Passo	Azione
1	Indirizzo di memoria ← [PC], Leggi memoria, IR ← Dati da memoria, PC ← [PC] + 4
2	Decodifica istruzione, RA ← [R8], RB ← R6
3	RZ ← [RA] + Valore immediato X, RM ← [RB]
4	Indirizzo di memoria ← [RZ], Dati per memoria ← [RM], Scrivi in memoria
5	Nessuna azione

Figura 5.14 Sequenza di azioni necessarie per prelevare ed eseguire l'istruzione Store R6, X (R8).

cessore, rendendo così la sua velocità prossima a quella del banco di registri. Quindi, un'operazione di Read o Write in memoria può essere completata in un ciclo di clock quando i dati coinvolti siano disponibili nella cache. Quando l'operazione richiede l'accesso alla memoria principale, il processore deve aspettare che tale operazione sia completata. Si discuterà la gestione di accessi lenti alla memoria nel Paragrafo 5.4.2.

Si è inoltre assunto che il processore legga i registri sorgenti dell'istruzione durante il passo 2 mentre sta ancora decodificando il codice operativo dell'istruzione che ha appena caricato sull'IR. Questi due compiti possono essere completati nello stesso passo? Come può l'hardware di controllo sapere quali registri leggere prima di completare la decodifica dell'istruzione? Questo è possibile poiché gli indirizzi dei registri sorgenti sono specificati nelle stesse posizioni di bit in tutte le istruzioni. L'hardware legge i registri i cui indirizzi sono nei bit di queste posizioni non appena l'istruzione è caricata nell'IR. I loro contenuti sono caricati nei registri RA e RB alla fine del passo 2. Se questi dati sono necessari all'istruzione, essi saranno disponibili per l'uso nel passo 3. Altrimenti, essi saranno ignorati dagli stadi successivi dell'hardware.

Si noti che le azioni descritte nelle Figure 5.11, 5.13 e 5.14 non mostrano la lettura di due registri nel passo 2 in ciascun caso. Per evitare confusione, sono stati indicati solo i registri necessari alla specifica istruzione descritta nella relativa figura, sebbene siano letti sempre due registri.

5.4.1 Salti

Le istruzioni vengono prelevate da parole in successive locazioni di memoria durante l'esecuzione sequenziale del programma. Quando un'istruzione viene prelevata, il processore incrementa il PC di 4 per puntare alla prossima parola. Questo modello di esecuzione continua finché un'istruzione di salto o di chiamata a sottoprogramma carica un nuovo indirizzo nel PC. Istruzioni di chiamata a sottoprogramma salvano inoltre l'indirizzo di rientro, da usare quando si ritorna al programma chiamante. In questo paragrafo si esaminano le azioni necessarie per realizzare queste istruzioni. Le interruzioni dai dispositivi di I/O e le istruzioni di interruzione software sono gestite in modo simile.

Le istruzioni di salto specificano l'indirizzo di destinazione del salto relativamente al PC. Uno spiazzamento di salto fornito come valore immediato nell'istruzione viene aggiunto al contenuto corrente del PC. Il numero di bit usati per questo spiazzamento è notevolmente più piccolo della lunghezza della parola del calcolatore, poiché un po' di spazio dell'istruzione è necessario per specificare il codice operativo e la condizione per il salto. Quindi l'intervallo di indirizzi che può essere raggiunto da un'istruzione di salto è limitato.

Istruzioni di chiamata a sottoprogramma possono raggiungere un intervallo di indirizzi più ampio. Poiché esse non includono una condizione, sono disponibili più bit per specificare l'indirizzo di destinazione. Inoltre, la maggior parte dei calcolatori in stile RISC ha istruzioni Jump e Call che impiegano un registro di uso generale per specificare un indirizzo a 32 bit. I dettagli variano da un calcolatore a un altro, come illustrano gli esempi di tali istruzioni nei processori trattati negli Approfondimenti applicativi 2.4 e 2.5.

Istruzioni di salto

La sequenza di passi che realizza un'istruzione di salto non condizionato è mostrata in Figura 5.15. L'istruzione viene prelevata e il PC viene incrementato come al solito nel passo 1. Dopo che l'istruzione è stata decodificata nel passo 2, il multiplatore MuxINC seleziona lo spiazzamento del salto nell'IR da aggiungere al PC nel passo 3. Questo è l'indirizzo che sarà usato per prelevare la prossima istruzione. L'esecuzione di un'istruzione Branch è completata con il passo 3. Nessuna azione viene svolta nei passi 4 e 5.

Passo	Azione
1	Indirizzo di memoria ← [PC], Leggi memoria, IR ← Dati da memoria, PC ← [PC] + 4
2	Decodifica istruzione
3	PC ← [PC] + Spiazzamento per il salto
4	Nessuna azione
5	Nessuna azione

Figura 5.15 Sequenza di azioni necessarie per prelevare ed eseguire un'istruzione di salto incondizionato.

Si è spiegato nel Paragrafo 2.13 che lo spiazzamento per il salto è la distanza tra la destinazione del salto e la locazione di memoria che segue l'istruzione di salto. Il motivo di ciò può essere facilmente visto in Figura 5.15. Il PC viene incrementato di 4 nel passo 1, nel momento in cui l'istruzione di salto viene prelevata. Quindi l'indirizzo di destinazione del salto viene calcolato nel passo 3 aggiungendo lo spiazzamento del salto al contenuto già aggiornato del PC.

La sequenza in Figura 5.15 può essere modificata per realizzare istruzioni di salto condizionato. Nei processori che non usano codici di condizione, l'istruzione di salto specifica un'operazione di confronto che determina la condizione per il salto. Per esempio, l'istruzione

Branch_if_[R5]=[R6] CICLO

provoca un salto se i contenuti dei registri R5 e R6 sono identici. Quando quest'istruzione viene eseguita, i contenuti dei registri vengono confrontati e se sono uguali viene effettuato un salto alla locazione CICLO.

La Figura 5.16 mostra come può essere eseguita questa istruzione. I registri R5 e R6 vengono letti nel passo 2, come al solito, e confrontati nel passo 3. Il confronto potrebbe essere effettuato tramite l'operazione di sottrazione [R5] − [R6] nell'ALU. L'ALU genera segnali che indicano se il risultato della sottrazione è positivo, negativo o zero. L'ALU può anche generare segnali per indicare se è avvenuto un trabocco e se l'operazione ha prodotto un riporto. Il circuito di controllo esamina questi segnali per valutare la condizione data nell'istruzione di salto. Nell'esempio di cui sopra, esso controlla se il risultato della sottrazione è uguale a zero. Se è così, l'indirizzo di destinazione del salto viene caricato nel PC, per essere usato nella successiva istruzione. Altrimenti, il contenuto del PC rimane pari al valore incrementato calcolato nel passo 1 e l'esecuzione sequenziale continua.

Passo	Azione
1	Indirizzo di memoria ← [PC], Leggi memoria, IR ← Dati da memoria, PC ← [PC] + 4
2	Decodifica istruzione, RA ← [R5], RB ← [R6]
3	Confronta [RA] con [RB], Se [RA] = [RB], allora PC ← [PC] + Spiazzamento per il salto
4	Nessuna azione
5	Nessuna azione

Figura 5.16 Sequenza di azioni necessarie per prelevare ed eseguire l'istruzione: Branch_if_[R5] = [R6] CICLO.

Come indica la sequenza di passi in Figura 5.16, le due azioni di confronto dei contenuti dei registri e di controllo del risultato sono entrambe svolte nel passo 3. Quindi il ciclo di clock deve essere sufficientemente lungo per completare le due azioni, una dopo l'altra. Per questo motivo, è auspicabile che il confronto venga effettuato il più velocemente possibile. Un'operazione di sottrazione nell'ALU impiega del tempo e in questo caso non è necessaria. Un circuito comparatore, più semplice e veloce, può esaminare i contenuti dei registri RA e RB e produrre i segnali di controllo richiesti, che indicano le condizioni di maggiore, uguale, minore ecc. Non si mostra separatamente un comparatore nella Figura 5.8 poiché esso è parte del blocco ALU. L'Esempio 5.3 mostra come può essere progettato un circuito comparatore.

Istruzioni di chiamata a sottoprogrammi

Chiamate a sottoprogramma e rientri possono essere realizzati in modo simile alle istruzioni di salto. L'indirizzo del sottoprogramma può essere calcolato usando un valore immediato fornito nell'istruzione o può essere fornito da uno dei registri di uso generale. La Figura 5.17 mostra la sequenza di azioni per l'istruzione

Call_Register R9

che chiama un sottoprogramma il cui indirizzo si trova nel registro R9. Il contenuto del registro viene letto e posto in RA nel passo 2. Durante il passo 3, il multiplatore MuxPC seleziona il suo ingresso 0, quindi trasferisce il dato nel registro RA perché sia caricato nel PC.

Si supponga che l'indirizzo di rientro da un sottoprogramma, che è il contenuto precedente del PC, debba essere salvato in un registro di uso generale chiamato LINK nel banco di registri. I dati vengono scritti nel banco di registri nel passo 5, quindi, non è possibile inviare l'indirizzo di rientro direttamente al banco di registri nel passo 3. Per mantenere il corretto flusso di dati nella struttura a cinque stadi, il processore salva l'indirizzo di rientro in un registro temporaneo, PC-Temp. Da lì, l'indirizzo di rientro viene trasferito nel registro RY nel passo 4, quindi al registro LINK nel passo 5. L'indirizzo di LINK è precodificato nel circuito di controllo.

Le istruzioni di rientro da sottoprogramma trasferiscono il valore salvato nel registro LINK al PC. La codifica dell'istruzione di rientro da sottoprogramma è tale che l'indirizzo del registro LINK appaia nei bit IR_{31-27}. Questo è il campo collegato all'ingresso Indirizzo A del banco di registri. Perciò, quando l'istruzione viene prelevata, il registro LINK viene letto e il suo contenuto posto in RA, da dove può essere trasferito al PC tramite MuxPC in Figura 5.10. Le istruzioni di rientro da interruzione sono gestite in modo simile, eccetto che un registro diverso viene usato per mantenere l'indirizzo di rientro.

Passo	Azione
1	Indirizzo di memoria ← [PC], Leggi memoria, IR ← Dati da memoria, PC ← [PC] + 4
2	Decodifica istruzione, RA ← [R9]
3	PC-Temp ← [PC], PC ← [RA]
4	RY ← [PC-Temp]
5	Registro LINK ← [RY]

Figura 5.17 Sequenza di azioni necessarie per prelevare ed eseguire l'istruzione: Call_Register R9.

5.4.2 In attesa della memoria

Il ruolo del circuito di interfaccia processore-memoria è di controllare il trasferimento di dati fra il processore e la memoria. Si è precedentemente indicato che i processori moderni usano memorie cache veloci disponibili sullo stesso chip. Spesso, le istruzioni o i dati cui si accede in memoria con operazioni Read e Write vengono trovati nella cache e in questo caso l'operazione viene completata in un ciclo di clock. Quando l'informazione richiesta non si trova nella cache e deve essere prelevata dalla memoria principale, possono essere necessari parecchi cicli di clock. Il circuito di interfaccia deve informare di questa situazione il circuito di controllo del processore, per differire i passi di esecuzione successivi finché l'operazione con la memoria non sia stata completata.

Si assuma che il circuito di interfaccia fra processore e memoria generi un segnale chiamato *operazione in memoria completata*, MFC (*memory function completed*). Questo segnale viene asserito quando una richiesta per un'operazione Read o Write viene completata. Il circuito di controllo del processore esamina questo segnale durante ciascun passo di elaborazione in cui viene inviata una richiesta di Read o Write in memoria, per determinare quando può procedere con il passo successivo. Quando i dati richiesti vengono trovati nella cache, il circuito di interfaccia asserisce il segnale MFC prima della fine dello stesso ciclo di clock in cui è stata inviata la richiesta di accesso a memoria. Quindi, l'esecuzione delle istruzioni continua senza interruzioni. Se è necessario accedere alla memoria principale, il circuito di interfaccia differisce l'asserzione di MFC finché l'operazione non sia stata completata. In questo caso, il circuito di controllo del processore deve estendere la durata del passo di esecuzione per il numero di cicli di clock necessario, finché MFC non sia asserito. Si userà il comando Attesa MFC per indicare che un certo passo di esecuzione deve essere esteso, se necessario, finché l'operazione in memoria non sia completata. Quando MFC viene ricevuto, le azioni specificate nel passo vengono completate e il processore procede al passo successivo nella sequenza di esecuzione.

Il passo 1 della sequenza di esecuzione di ogni istruzione coinvolge il prelievo dell'istruzione dalla memoria. Quindi, tale passo deve includere un comando Attesa MFC, come segue:

Indirizzo di memoria ← [PC], Lettura memoria, Attesa MFC,

IR ← Dati da memoria, PC ← [PC] + 4

Il comando Attesa MFC è pure necessario nel passo 4 delle istruzioni Load e Store nelle Figure 5.13 e 5.14. Spesso, l'informazione richiesta viene trovata nella cache, quindi il segnale MFC viene generato velocemente e il passo viene completato in un ciclo di clock. Quando un accesso coinvolge la memoria principale, la risposta MFC viene differita e il passo viene esteso di alcuni cicli di clock.

5.5 Segnali di controllo

Il funzionamento delle componenti hardware del processore è governato da *segnali di controllo*. Questi segnali determinano quale ingresso dei multiplatori viene selezionato, quale operazione viene svolta dall'ALU ecc. In questo paragrafo si discutono i segnali necessari per controllare il funzionamento delle componenti nelle Figure da 5.8 a 5.10.

È utile cominciare rammentando come si spostano i dati attraverso i quattro stadi del percorso dati descritto nel Paragrafo 5.3.3. In ciascun ciclo di clock, i risultati delle azioni svolte in uno stadio vengono immagazzinati nei registri interstadi, per essere disponibili all'uso nel

successivo stadio al prossimo ciclo di clock. Poiché i dati vengono trasferiti da uno stadio al successivo in ciascun ciclo di clock, i registri interstadi sono sempre abilitati. È così per i registri RA, RB, RZ, RY, RM e PC-Temp. I contenuti di altri registri, ovvero il PC, l'IR e il

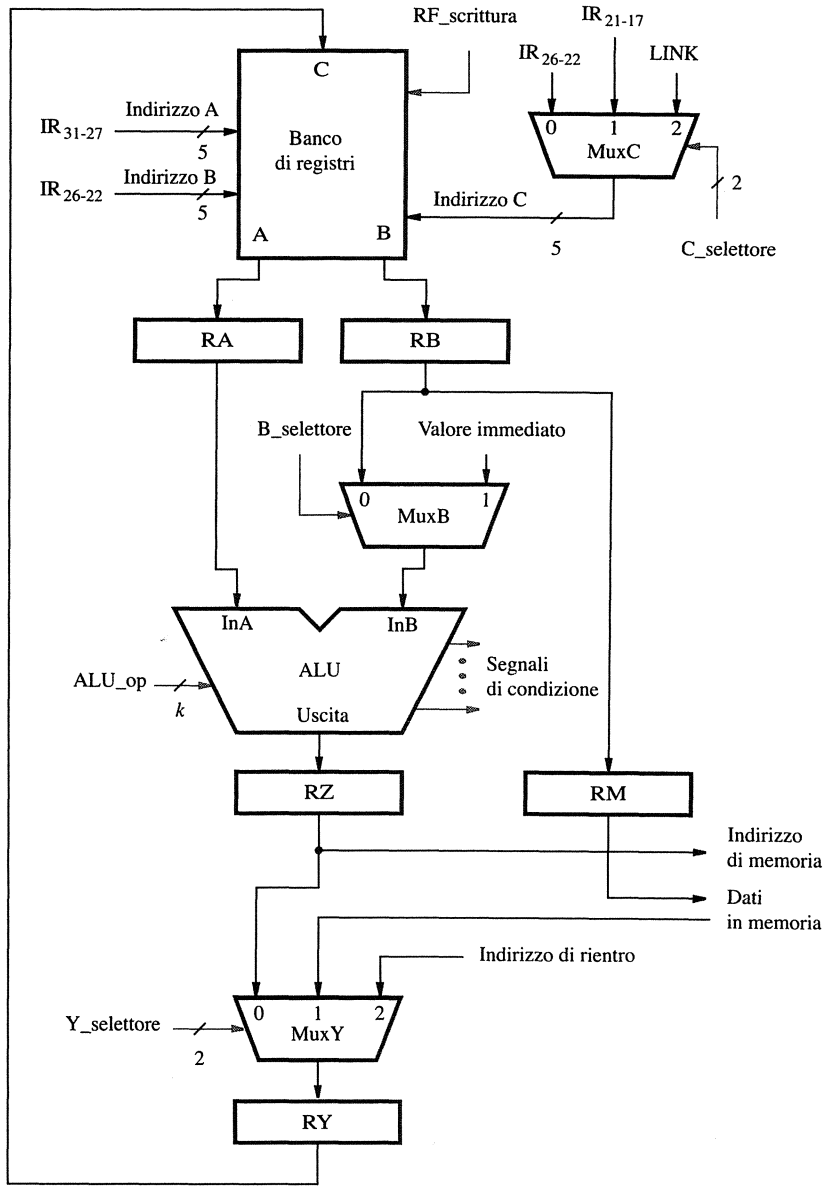


Figura 5.18 Segnali di controllo per il percorso dati.

banco di registri, non devono essere alterati in ogni ciclo di clock. Nuovi dati vengono caricati in questi registri solo quando ciò è richiesto in un particolare passo di elaborazione. Essi devono essere abilitati solo in quei momenti.

Il ruolo dei multiplatori è di selezionare i dati da elaborare in ciascuno stadio. Per esempio, MuxB nello stadio 3 di Figura 5.8 seleziona il campo immediato nell'IR per le istruzioni che usano un operando sorgente immediato. Tale campo viene inoltre selezionato per le istruzioni che usano un dato immediato come spiazzamento quando si calcola l'indirizzo effettivo di un operando in memoria. Altrimenti, si seleziona il registro RB. I dati selezionati dal multiplatore vengono usati dall'ALU. Dalle Figure 5.11, 5.13 e 5.14 è possibile vedere che l'ALU è usata solo nel passo 3 e quindi la selezione effettuata da MuxB è rilevante solo durante tale passo. Per semplificare il circuito di controllo necessario, la stessa selezione può essere mantenuta durante tutti i passi d'esecuzione. Una simile osservazione può essere fatta per MuxY. Tuttavia, MuxMA nella Figura 5.9 deve cambiare la sua selezione in passi diversi di esecuzione. Esso seleziona il PC come sorgente dell'indirizzo di memoria durante il passo 1, quando una nuova istruzione sta per essere prelevata. Durante il passo 4 delle istruzioni Load e Store seleziona il registro RZ, che contiene l'indirizzo effettivo dell'operando in memoria.

Le Figure 5.18, 5.19 e 5.20 mostrano i segnali di controllo richiesti. Il banco di registri ha tre ingressi a 5 bit, per permettere l'accesso a 32 registri di uso generale. Due di questi ingressi, Indirizzo A e Indirizzo B, determinano quali registri devono essere letti. Essi sono collegati ai campi IR_{31-27} e IR_{26-22} nel registro di istruzione. Il terzo ingresso degli indirizzi,

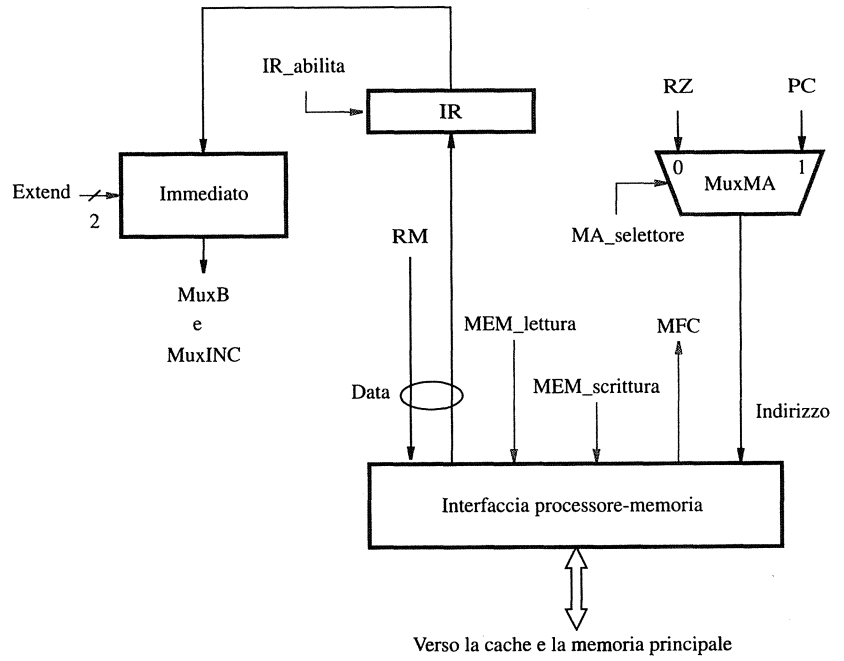


Figura 5.19 Segnali di controllo dell'interfaccia processore-memoria e del registro di istruzione.

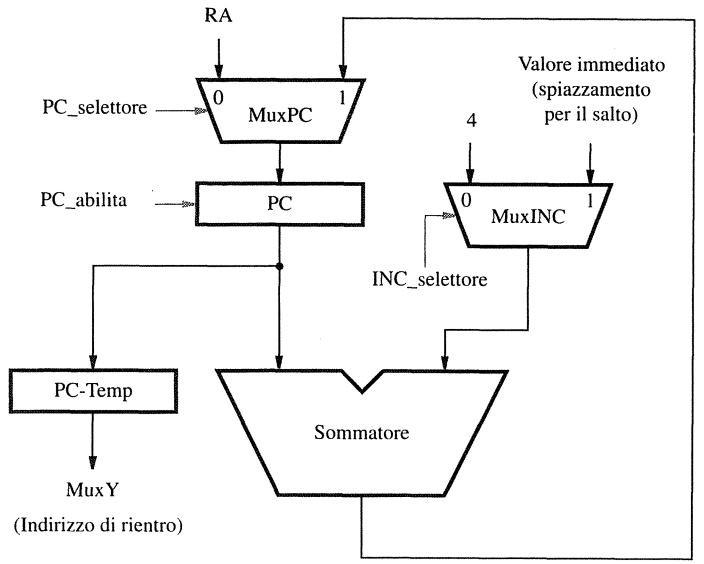


Figura 5.20 Segnali di controllo per il generatore di indirizzi delle istruzioni.

Indirizzo C, seleziona il registro di destinazione, nel quale devono essere scritti i dati in ingresso alla porta C. Il multiplatore MuxC seleziona la sorgente di quell'indirizzo. Si è assunto che istruzioni con tre registri usino i bit IR_{21-17} e altre istruzioni usano IR_{26-22} per specificare il registro di destinazione, come in Figura 5.12. Il terzo ingresso del multiplatore è l'indirizzo del registro di collegamento usato in istruzioni di collegamento a sottoprogrammi. Nuovi dati vengono caricati nel registro selezionato solo quando viene asserito il segnale di controllo RF_scrittura.

I multiplatori sono controllati da segnali che selezionano quali dati in ingresso compariranno nell'uscita del multiplatore. Per esempio, quando B_selettore è uguale a 0, MuxB seleziona il contenuto del registro RB per renderlo disponibile all'ingresso InB dell'ALU. Si noti che sono necessari due bit per controllare MuxC e MuxY, poiché ciascun multiplatore seleziona uno fra tre ingressi.

L'operazione effettuata dall'ALU è determinata da un codice di controllo di k bit, ALU_op, che può specificare fino a 2^k operazioni distinte, come Add, Subtract, AND, OR e XOR. Quando un'istruzione richiede il confronto di due valori, un comparatore effettua il confronto specificato, come precedentemente indicato. Il comparatore genera segnali di condizione che indicano il risultato del confronto. Questi segnali vengono esaminati dal circuito di controllo durante l'esecuzione di istruzioni di salto condizionato per determinare se la condizione per il salto sia vera o falsa.

L'interfaccia processore-memoria e i segnali di controllo associati al registro di istruzione sono presentati in Figura 5.19. Due segnali, MEM_lettura e MEM_scrittura vengono usati per iniziare un'operazione Read o Write in memoria. Quando l'operazione richiesta è stata completata, l'interfaccia asserisce il segnale MFC. Il registro d'istruzione ha un segnale di controllo, IR_abilita, che abilita il caricamento di una nuova istruzione nel registro. Durante un passo di prelievo, IR_abilita deve essere attivato solo dopo che il segnale MFC sia stato asserito.

Si è assunto che il blocco Immediato gestisca tre possibili formati per il valore immediato: un valore a 16 bit esteso con segno, un valore a 16 bit esteso con zeri e un valore a 26 bit che è gestito in modo speciale (si veda il Problema 5.14). Quindi il suo segnale di controllo, Extend, consta di due bit.

I segnali che controllano il funzionamento del generatore di indirizzi delle istruzioni sono mostrati in Figura 5.20. Il segnale INC_selettore seleziona il valore da aggiungere al PC, ovvero la costante 4 o lo spiazzamento di salto specificato nell'istruzione. Il segnale PC_selettore seleziona l'indirizzo aggiornato o il contenuto del registro RA da caricare nel PC quando il segnale PC_abilita viene attivato.

5.6 Controllo di tipo cablato

Per eseguire istruzioni macchina il processore deve generare internamente le sequenze appropriate di segnali di controllo, ovvero le sequenze di ordini che figurano negli esempi discussi sopra. Vi sono svariate soluzioni al problema, ma nell'essenziale si riducono a due sole: la metodologia cablata e quella microprogrammata. Qui verrà spiegata brevemente la metodologia cablata.

Un'istruzione viene eseguita in una sequenza di passi, in cui ogni passo della sequenza occupa esattamente un ciclo di clock. Dunque si può pensare di usare un contatore per enumerare i passi mentre si susseguono. Ogni valore assumibile da parte del contatore corrisponde a un ben preciso passo di controllo, e solo a quello. Si possono allora generare gli ordini contenuti in ciascun passo partendo dalle informazioni seguenti:

- lo stato di conteggio, cioè il valore corrente del contatore;
- il contenuto del registro di istruzione, cioè l'istruzione corrente;
- i bit di esito correnti (o codici di condizione, se si preferisce);
- segnali provenienti dall'esterno del processore, come per esempio le interruzioni.

Il circuito che genera i segnali di controllo può essere organizzato come mostrato in Figura 5.21. Il decodificatore di istruzioni interpreta il codice operativo e l'informazione sul modo di indirizzamento nell'IR e pone a 1 l'uscita corrispondente INSi. Durante ciascun ciclo di clock, una delle uscite da T1 a T5 del contatore di passi è posta a 1 per indicare quale dei cinque passi coinvolti in istruzioni di prelievo ed esecuzione è in corso. Poiché tutte le istruzioni vengono completate in cinque passi, si può usare un contatore modulo 5. Il generatore di segnali di controllo è un circuito combinatorio che produce i segnali di controllo necessari in base a tutti i suoi ingressi. Le impostazioni richieste per i segnali di controllo possono essere determinate dalle sequenze di azioni che realizzano ciascuna delle istruzioni rappresentate dai segnali da INS1 a INSm.

Per esempio, si consideri il passo 1 nel processo di esecuzione di istruzioni. Questo è il passo in cui una nuova istruzione viene prelevata dalla memoria. Esso è identificato dal segnale T1 asserito. Durante il periodo di clock, il segnale MA_selettore in Figura 5.19 è posto a 1 per selezionare il PC come sorgente dell'indirizzo della memoria e MEM_lettura è attivato per iniziare un'operazione di Read dalla memoria. I dati ricevuti dalla memoria vengono caricati nell'IR tramite l'attivazione di IR_abilita quando il segnale di risposta della memoria, MFC, viene asserito. Allo stesso tempo, il PC viene incrementato di 4, ponendo a 0 il segnale INC_selettore in Figura 5.20 e PC_selettore a 1. Il segnale PC_abilita viene attivato per far sì che il nuovo valore sia caricato nel PC sul fronte attivo del clock che segna la fine del passo T1.

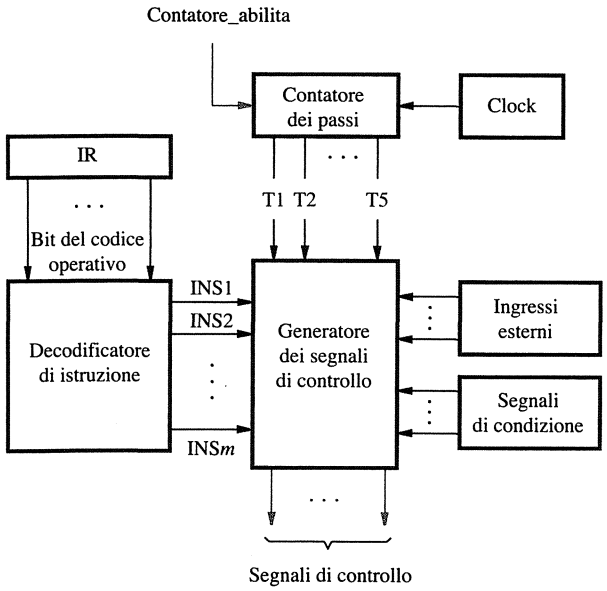


Figura 5.21 Generazione dei segnali di controllo.

5.6.1 Segnali di controllo del percorso dati

Le istruzioni che manipolano dati includono Load, Store e tutte le istruzioni di calcolo. Esse effettuano vari spostamenti di dati e operazioni di manipolazione usando il percorso dati del processore, i cui segnali di controllo sono mostrati nelle Figure 5.18 e 5.19. Quando un'istruzione viene caricata nell'IR, il decodificatore delle istruzioni interpreta il suo contenuto per determinare le azioni necessarie. Allo stesso tempo, i registri sorgenti vengono letti e i loro contenuti diventano disponibili alle uscite A e B del banco di registri. Come detto prima, i registri interstadio RA, RB, RZ, RM e RY sono sempre abilitati. Questo significa che i dati fluiscono automaticamente da uno stadio del percorso dati al successivo su ciascun fronte attivo del segnale di clock.

Le impostazioni utili per i vari segnali di controllo possono essere determinate esaminando le azioni svolte in ciascun passo di esecuzione di ogni istruzione. Per esempio, il segnale RF_scrittura è posto a 1 nel passo T5 durante l'esecuzione di un'istruzione che scrive dati nel banco di registri. Esso può essere generato dall'espressione logica

$$\text{RF_scrittura} = \text{T5} \cdot (\text{ALU} + \text{Load} + \text{Call})$$

dove ALU indica[1] tutte le istruzioni che effettuano operazioni aritmetiche o logiche, Load indica tutte le istruzioni Load, e Call indica tutte le chiamate a sottoprogrammi e le istruzioni

[1] Con riferimento a questa formula logica e alle successive, "indica" va letto come "denota l'appartenenza all'insieme di"; come di solito, i simboli "." e "+" nelle espressioni logiche designano congiunzione e disgiunzione, rispettivamente.

di interruzione software. Il segnale RF_scrittura è una funzione di istruzioni e segnali di temporizzazione. Tuttavia, come detto prima, le impostazioni di alcuni multiplatori non hanno bisogno di cambiare da un passo di temporizzazione a un altro. In questo caso, il segnale di selezione del multiplatore può essere realizzato come una funzione della sola istruzione. Per esempio,

$$\text{B_selettore} = \text{Immediato}$$

in cui Immediato indica tutte le istruzioni che usano un valore immediato nell'IR. Si stimola il lettore a esaminare altri segnali di controllo e derivare le espressioni logiche appropriate per essi, in base ai passi di esecuzione di varie istruzioni.

5.6.2 Gestione dei ritardi della memoria

I segnali di temporizzazione da T1 a T5 vengono asseriti in sequenza mano a mano che il contatore di passi viene incrementato. Perlopiù, il contatore di passi viene incrementato alla fine di ciascun ciclo di clock. Tuttavia, i passi in cui viene emesso un comando MEM_lettura o MEM_scrittura non terminano finché il segnale MFC non sia asserito, indicando che l'operazione di memoria richiesta è stata completata.

Per estendere la durata di un passo di esecuzione a più di un ciclo di clock, bisogna disabilitare il contatore di passi. Si assuma che il contatore sia incrementato quando abilitato da un segnale di controllo designato Contatore_abilita. Si supponga che la necessità di aspettare affinché un'operazione in memoria venga completata sia indicata da un segnale di controllo designato WMFC, che venga attivato durante ciascun passo d'esecuzione in cui si ometta il comando di attesa per MFC. Contatore_abilita dovrebbe essere posto a 1 in ciascun passo in cui WMFC non è asserito. Altrimenti, esso dovrebbe essere posto a 1 quando MFC è asserito. Questo significa che

$$\text{Contatore_abilita} = \overline{\text{WMFC}} + \text{MFC}$$

Un nuovo valore viene caricato nel PC alla fine di ogni ciclo di clock in cui il segnale PC_abilita in Figura 5.20 è attivato. Bisogna assicurarsi che il PC venga incrementato solo una volta quando un passo di esecuzione è esteso per più di un ciclo di clock. Quindi, quando si preleva un'istruzione, il PC dovrebbe essere abilitato solo quando MFC viene ricevuto. È anche abilitato nel passo 3 di istruzioni che causano salti. Si supponga che BR indichi tutte le istruzioni di questo gruppo. Allora PC_abilita può essere realizzato come

$$\text{PC_abilita} = \text{T1} \cdot \text{MFC} + \text{T3} \cdot \text{BR}$$

5.7 Processori in stile CISC

Nei precedenti paragrafi si è visto che l'insieme di istruzioni in stile RISC può facilitare una realizzazione a più stadi del processore. Tutte le istruzioni possono essere eseguite in modo uniforme usando lo stesso hardware a cinque stadi, di conseguenza l'hardware è semplice e adatto a funzionare in pipeline. Inoltre, i segnali di controllo sono facili da generare.

Gli insiemi di istruzioni in stile CISC sono più complessi poiché essi permettono maggiore flessibilità nell'accesso agli operandi delle istruzioni. A differenza degli insiemi di istruzioni in stile RISC, per i quali solo le istruzioni Load e Store accedono a dati in memoria, le istruzioni CISC possono lavorare direttamente su operandi in memoria. Inoltre, esse non sono limitate a una parola di lunghezza. Un'istruzione può usare più parole per specificare gli indirizzi degli operandi e le azioni da effettuare, come spiegato nel Paragrafo 2.10. Quindi, le istruzioni in stile CISC richiedono un'organizzazione differente dell'hardware del processore.

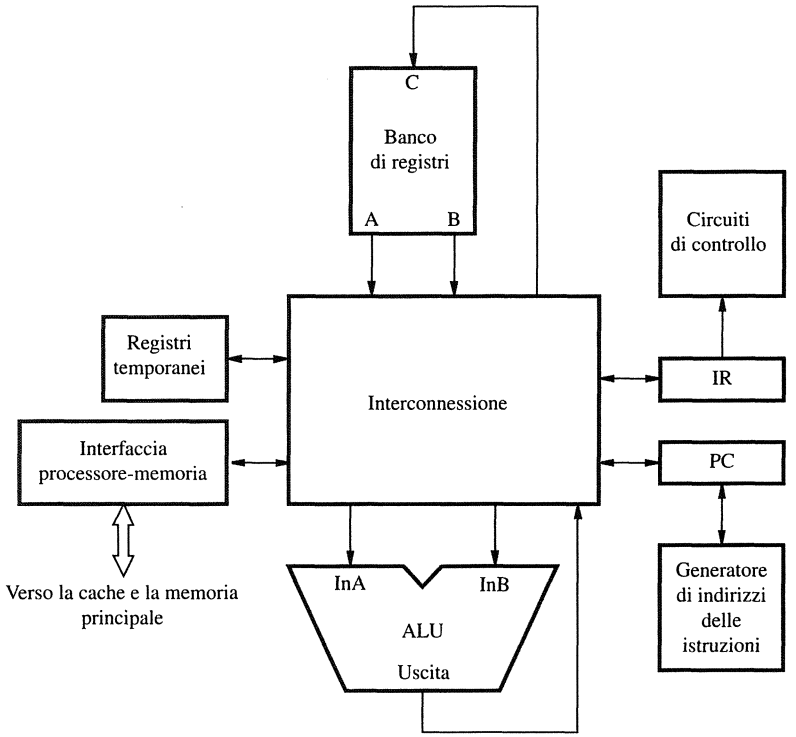


Figura 5.22 Organizzazione di un processore CISC.

La Figura 5.22 mostra una possibile organizzazione del processore. La differenza principale tra questa organizzazione e la struttura a cinque stadi precedentemente discussa è che il blocco Interconnessione, che fornisce le interconnessioni con altri blocchi, non prescrive una particolare struttura o sequenza di flusso di dati. Esso fornisce percorsi che rendono possibile trasferire i dati fra qualunque coppia di componenti, in base alle istruzioni da realizzare. La struttura a più stadi di Figura 5.8 usa registri interstadi, come RZ e RY. Questi non sono necessari nell'organizzazione di Figura 5.22. Servono invece alcuni registri per mantenere i risultati intermedi durante l'esecuzione delle istruzioni. Il banco di registri temporanei presente in figura è fornito per questo scopo. Esso include due registri temporanei, Temp1 e Temp2. La necessità di questi registri sarà rivelata da esempi più avanti.

Un approccio tradizionale alla realizzazione del blocco Interconnessione è l'uso dei bus. Un *bus* consiste di un insieme di linee a cui possono essere connessi vari dispositivi, abilitando così i dati a essere trasferiti da qualunque dispositivo a qualsiasi altro. Una porta logica che invia un segnale su una linea del bus viene chiamata *pilota del bus* (*bus driver*). Poiché tutti i dispositivi connessi al bus possono inviare dati, è necessario assicurarsi che solo uno di essi stia pilotando il bus in ogni dato momento. Per questo motivo, il pilota del bus è un tipo speciale di porta logica chiamata *porta a tre stati* (*tri-state gate* o *tri-state buffer*, si veda l'Appendice A). Essa ha un ingresso di controllo che abilita o disabilita la porta, mentre

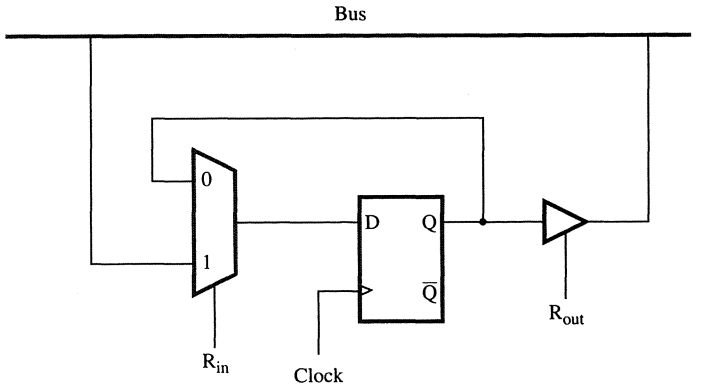


Figura 5.23 Connessioni al bus per la scrittura e lettura di un bit di un registro.

quando è abilitata, la porta invia un segnale logico 0 o 1 sul bus, in base al valore del suo ingresso. Quando è disabilitata, la porta è elettricamente disconnessa dal bus, come spiegato in Appendice A.

La Figura 5.23 mostra come un flip-flop che forma un bit di un registro di dati possa essere connesso a una linea del bus. Vi sono due segnali di controllo, R_{in} e R_{out}. Quando R_{in} è posto a 1 il multiplatore seleziona i dati sulla linea del bus da caricare nel flip-flop. Porre R_{in} a 0 fa sì che il flip-flop mantenga il suo valore corrente. L'uscita del flip-flop è collegata alla linea del bus tramite una porta a tre stati, abilitata quando R_{out} è asserito. In altre occasioni, la porta a tre stati è disabilitata, permettendo ad altre componenti di pilotare la linea del bus.

5.7.1 Interconnessione che usa i bus

L'Interconnessione in Figura 5.22 può essere realizzata usando uno o più bus. La Figura 5.24 mostra una realizzazione con tre bus. Si assume che tutti i registri commutino sul fronte, ovvero, quando un registro è abilitato, i dati vengono caricati su esso sul fronte attivo del clock alla fine del periodo di clock. Gli indirizzi per le tre porte del banco di registri vengono forniti dal blocco di Controllo. Queste connessioni non vengono mostrate per semplicità della figura. Non viene neppure mostrato il blocco Immediato attraverso cui l'IR viene connesso al bus B. Questo è il circuito che estende a 32 bit un operando immediato nell'IR.

Si consideri l'istruzione a due operandi:

Add R5, R6

che effettua l'operazione:

$$[R5] \leftarrow [R5] + [R6]$$

Il prelievo e l'esecuzione di questa istruzione, usando l'hardware in Figura 5.24, possono essere effettuati in tre passi, come mostrato in Figura 5.25. Ciascun passo, eccetto il passo che coinvolge l'accesso alla memoria, viene completato in un ciclo di clock. Nel passo 1, il bus B viene usato per inviare il contenuto del PC all'interfaccia processore-memoria, che lo invia sulle linee dell'indirizzo della memoria e inizia un'operazione Read di memoria. I dati ricevuti

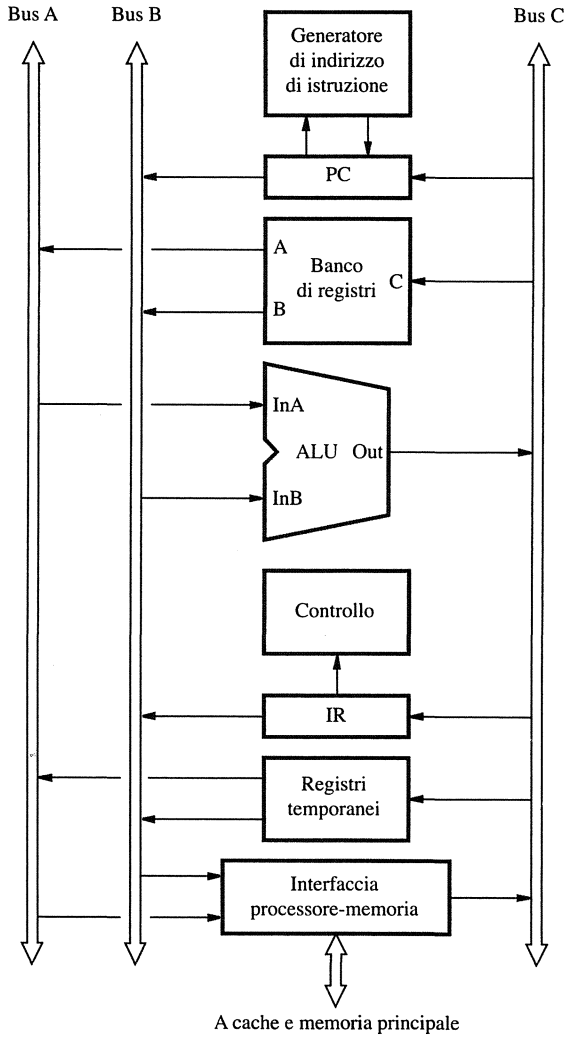


Figura 5.24 Struttura di unità di calcolo di processore CISC con tre bus interni.

dalla memoria, che rappresentano un'istruzione da eseguire, sono inviati all'IR sul bus C. Il comando Attesa MFC viene incluso per consentire la possibilità che l'accesso alla memoria richieda più di un ciclo di clock, come spiegato nel Paragrafo 5.4.2. L'istruzione viene decodificata nel passo 2 e il circuito di controllo inizia la lettura dei registri sorgenti R5 e R6. Tuttavia, i contenuti dei registri non diventano disponibili sulle uscite A e B del banco di registri prima del passo 3. Essi vengono inviati all'ALU usando i bus A e B. L'ALU effettua l'operazione di addizione e la somma viene inviata indietro all'ALU sul bus C, per essere scritta nel registro R5 alla fine del ciclo di clock.

Passo	Azione
1	Indirizzo di memoria ← [PC], Leggi memoria, Attesa MFC, IR ← Dati da memoria, PC ← [PC] + 4
2	Decodifica istruzione
3	R5 ← [R5] + [R6]

Figura 5.25 Sequenza di azioni necessarie per prelevare ed eseguire l'istruzione Add R5, R6.

Si noti che la lettura dei registri sorgenti viene completata nel passo 2 in Figura 5.11. In questo caso, l'azione di lettura dei registri continua in parallelo con l'azione di decodifica dell'istruzione, poiché la locazione dei campi di bit che contengono gli indirizzi dei registri in un'istruzione in stile RISC è conosciuta. Poiché le istruzioni in stile CISC non sempre usano lo stesso campo dell'istruzione per specificare gli indirizzi dei registri, l'azione di lettura dei registri sorgenti non inizia finché l'istruzione non sia stata almeno parzialmente decodificata, quindi, potrebbe non essere possibile completare la lettura dei registri sorgenti nel passo 2.

Ora, si consideri l'istruzione

And X(R7), R9

che effettua un'operazione di AND logico sui contenuti del registro R9 e della locazione di memoria X + [R7] e immagazzina il risultato nella stessa locazione di memoria. Si assuma che lo spiazzamento X sia un valore a 32 bit fornito dalla seconda parola dell'istruzione. Per eseguire questa istruzione, è necessario accedere alla memoria quattro volte. Inizialmente si preleva la parola del codice operativo. Poi, quando il circuito di decodifica dell'istruzione riconosce il modo di indirizzamento con indice e spiazzamento, viene prelevato lo spiazzamento X. Successivamente, si preleva l'operando in memoria e l'operazione AND viene effettuata. Infine, si immagazzina il risultato in memoria.

La Figura 5.26 presenta i passi necessari a eseguire l'istruzione. Dopo aver decodificato l'istruzione nel passo 2, la seconda parola dell'istruzione viene letta nel passo 3. Il dato ricevuto, che rappresenta lo spiazzamento X, viene immagazzinato temporaneamente nel registro Temp1, per essere usato nel prossimo passo per calcolare l'indirizzo effettivo dell'operando in memoria. Nel passo 4, i contenuti dei registri Temp1 e R7 vengono inviati agli ingressi dell'ALU sui bus A e B. L'indirizzo effettivo viene calcolato e posto nel registro Temp2, quindi usato per leggere l'operando nel passo 5. Qui il registro Temp1 viene usato nuovamente, questa volta per mantenere l'operando ricevuto dalla memoria. Il calcolo viene effettuato nel passo 6 e il risultato è posto nel registro Temp1. Nel passo finale, il risultato viene inviato in memoria per essere immagazzinato all'indirizzo dell'operando, che è ancora disponibile nel registro Temp2.

I due esempi nelle Figure 5.25 e 5.26 illustrano la variabilità del numero di passi di esecuzione nelle istruzioni in stile CISC. Non c'è uniformità nella sequenza di azioni che possono essere svolte per tutte le istruzioni, come invece accade nel modo visto per le istruzioni RISC nel Paragrafo 5.2.

5.7.2 Controllo microprogrammato

Nel Paragrafo 5.6 si è visto come internamente al processore si possano produrre i segnali di controllo tramite un controllo cablato costituito da contatore, decodificatore e generatore di or-

Passo	Azione
1	Indirizzo di memoria ← [PC], Leggi memoria, Attesa MFC, IR ← Dati da memoria, PC ← [PC] + 4
2	Decodifica istruzione
3	Indirizzo di memoria ← [PC], Leggi memoria, Attesa MFC, Temp1 ← Dati da memoria, PC ← [PC] + 4
4	Temp2 ← [Temp1] + [R7]
5	Indirizzo di memoria ← [Temp2], Leggi memoria, Attesa MFC, Temp1 ← Dati da memoria
6	Temp1 ← [Temp1] AND [R9]
7	Indirizzo di memoria ← [Temp2], Dati da memoria ← Temp1, Scrivi in memoria, Attesa MFC

Figura 5.26 Sequenza di azioni necessarie per prelevare ed eseguire l'istruzione: And X (R7), R9.

dini. Qui si illustrerà uno schema alternativo, molto diffuso in passato, vale a dire la tecnica di controllo microprogrammato, secondo cui i segnali di controllo del processore vengono prodotti tramite un vero e proprio programma interno, di dimensione ridotta e con funzioni specializzate, ma per il resto non troppo dissimile da un normale programma in linguaggio macchina.

Innanzitutto è opportuno definire la terminologia. Se si immagina di prendere i bit che rappresentano i valori assunti da un gruppo di segnali di ordine in un certo istante di tempo e di disporli ordinatamente, si ottiene una *parola di controllo* (*control word*). Si è visto in precedenza che qualsiasi istruzione macchina viene eseguita tramite una sequenza di passi di controllo. All'inizio di ciascun passo della sequenza vengono prodotti e diramati al processore ordini, dunque a ogni passo si può associare una ben definita parola di controllo, precisamente quella corrispondente agli ordini che attuano il passo. La parola di controllo corrispondente agli ordini impartiti all'inizio di un passo di controllo si chiama *microistruzione* (*microinstruction*), ogni passo ha dunque una ben definita microistruzione. Per esempio, l'azione di lettura di un'istruzione o di un operando dalla memoria richiede l'uso dei segnali MEM_lettura e WMFC introdotti nei Paragrafi 5.5 e 5.6.2, rispettivamente. Questi segnali vengono asseriti ponendo a 1 i bit corrispondenti nella parola di controllo per i passi 1, 3 e 5 nella Figura 5.26. Quando una microistruzione viene letta dalla memoria di controllo, ciascun segnale di controllo acquisisce il valore del suo bit corrispondente.

La sequenza di microistruzioni corrispondenti ai passi di controllo che eseguono l'istruzione macchina costituisce la *microroutine*; ciascuna istruzione ha dunque la rispettiva microroutine. I primi due passi nelle Figure 5.25 e 5.26 specificano le azioni per prelevare e decodificare un'istruzione. Esse sono comuni a tutte le istruzioni. La microroutine che è specifica per una certa istruzione macchina comincia con il passo 3.

La Figura 5.27 illustra una tipica organizzazione dell'hardware necessario per il controllo microprogrammato. Il registro *contatore di μ-programma* o μPC (*μ-program counter*) serve per prelevare sequenzialmente μ-istruzioni dalla μ-memoria di controllo. Ogniqualvolta una nuova istruzione macchina viene prelevata e scritta nel registro di istruzione IR, il circuito generatore di μ-indirizzo iniziale emette il μ-indirizzo della μ-istruzione iniziale della μ-routine che governa l'esecuzione dell'istruzione appena prelevata e lo scrive in μPC. Ogniqualvolta si passa dal ciclo di clock corrente a quello successivo il registro μPC viene incrementato di un'unità e in tale modo si leggono in successione tutte le μ-istruzioni necessarie, una per ciclo. L'effetto complessivo è quello di generare gli ordini per il processore e di diramarli ai componenti del processore cui sono destinati, il tutto nell'ordine temporale appropriato.

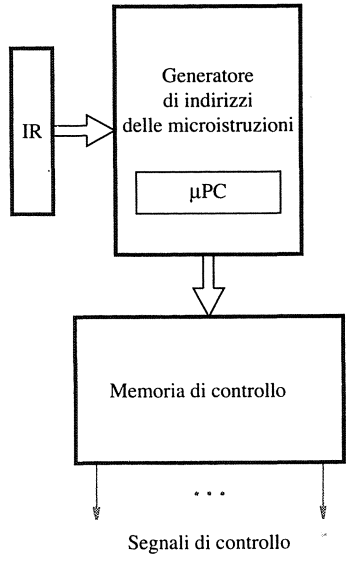


Figura 5.27 Organizzazione del controllo microprogrammato.

Il controllo microprogrammato può essere inteso come un processore per il controllo dentro il processore principale. Le microistruzioni vengono prelevate ed eseguite in modo simile alle istruzioni macchina. Il loro lavoro è di dirigere le azioni delle componenti hardware del processore principale, indicando quali segnali di controllo devono essere attivati durante ciascun passo di esecuzione.

Il controllo microprogrammato è semplice da realizzare e fornisce notevole flessibilità nel controllare l'esecuzione di istruzioni macchina, tuttavia è più lento del controllo cablato. Inoltre, la flessibilità che fornisce non è necessaria nei processori in stile RISC. Come ha illustrato la discussione di questo capitolo, i segnali di controllo necessari a realizzare le istruzioni in stile RISC sono generabili in modo relativamente semplice. Poiché il costo dei circuiti logici non è più un fattore significativo, il controllo cablato è diventato la scelta preferita.

Sommario

Questo capitolo ha spiegato la struttura di base di un processore e come esso esegue le istruzioni. I processori moderni hanno un'organizzazione a più stadi poiché questa struttura è adatta al funzionamento in pipeline. Ogni stadio realizza le azioni necessarie in uno dei passi di esecuzione di un'istruzione. È stata mostrata una sequenza di cinque passi in cui ciascun passo viene completato in un ciclo di clock. Questo approccio è usato comunemente nei processori che hanno un insieme di istruzioni in stile RISC.

La discussione in questo capitolo ha assunto che l'esecuzione di un'istruzione venga completata prima che la prossima istruzione sia prelevata. Solo uno dei cinque stadi viene usato in un certo momento, mano a mano che l'esecuzione si sposta da uno stadio al successivo in ogni ciclo di clock. Si mostrerà nel prossimo capitolo che è possibile sovrapporre i passi di esecuzione di istruzioni in sequenza, determinando prestazioni migliori. Questo porta a un'organizzazione in pipeline.

6 Introduzione al pipelining

In questo capitolo

- Il *pipelining* per migliorare le prestazioni sovrapponendo l'esecuzione di istruzioni macchina
- Possibili situazioni che limitano il guadagno di prestazioni in processori con pipeline e modi per alleviare il loro effetto
- Implicazioni hardware e software del pipelining
- Influenza del pipelining sulla progettazione dell'insieme di istruzioni
- Processori superscalari

Il Capitolo 5 ha introdotto l'organizzazione di un processore per l'esecuzione delle istruzioni una alla volta. In questo capitolo si discuterà il concetto di pipelining, che sovrappone l'esecuzione di istruzioni successive per ottenere alte prestazioni. Si inizierà spiegando i fondamenti del pipelining e come questo possa portare a prestazioni migliori. Quindi si esamineranno possibili situazioni di conflitto che provocano il degrado delle prestazioni e tecniche per alleviare il loro effetto sulle prestazioni. Si discuterà il ruolo dei *compilatori ottimizzanti*, che riadattano la sequenza di istruzioni per massimizzare i benefici dell'esecuzione in pipeline. Per migliorare ulteriormente le prestazioni, si prenderà in considerazione la replica di unità hardware in un processore *superscalare*, così che più pipeline possano funzionare concorrentemente.

6.1 Concetti fondamentali

La velocità di esecuzione del programma è influenzata da numerosi fattori. Un primo modo per aumentare la prestazione consiste nel progettare processore e memoria centrale in tecnologia microelettronica più veloce. L'alternativa consiste nel ristrutturare il processore in modo da renderlo maggiormente parallelo, cioè capace di svolgere più operazioni contemporaneamente. Occorrerebbe che complessivamente il processore potesse svolgere un numero maggiore di operazioni nell'unità di tempo, anche se alcune o al limite tutte le operazioni, prese individualmente, dovessero diventare più lente.

La metodologia di pipelining è un modo molto efficiente per svolgere e coordinare attività concorrenti nel calcolatore. L'idea in se stessa è molto semplice e la si trova applicata anche in molti altri ambiti, come la catena di montaggio comune nell'industria meccanica e manifatturiera. Si pensi alle lunghe catene di montaggio caratteristiche dell'industria automobilistica: più veicoli sono disposti lungo la catena, ciascuno in uno stadio differente di lavorazione. Può ben essere che il veicolo impieghi un giorno o più per percorrere tutta la catena, ma questa ogni pochi minuti può produrre un veicolo finito giacché ne contiene sempre numerosi in preparazione.

Si provi ora ad applicare la stessa idea al calcolatore. L'organizzazione a cinque stadi del processore in Figura 5.7 e il percorso dati corrispondente in Figura 5.8 permettono alle

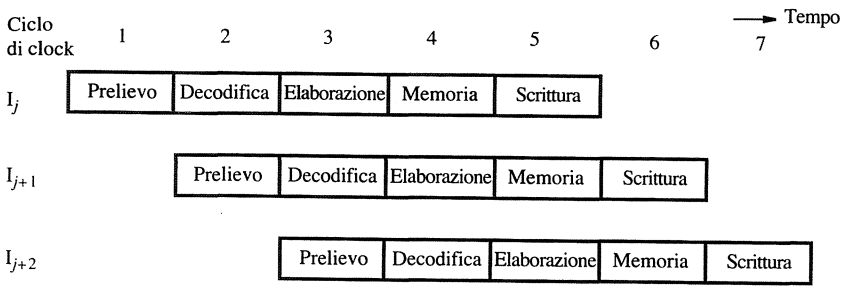


Figura 6.1 Esecuzione in pipeline: il caso ideale

istruzioni di essere prelevate ed eseguite una alla volta. Per completare l'esecuzione di ciascuna istruzione sono necessari cinque cicli di clock. Anziché aspettare finché ciascuna istruzione non sia stata completata, le istruzioni possono essere prelevate ed eseguite in pipeline, come mostrato in Figura 6.1. I cinque stadi corrispondenti a quelli in Figura 5.7 sono designati Prelievo, Decodifica, Elaborazione, Memoria e Scrittura. L'istruzione I_j viene prelevata nel primo ciclo e attraversa gli stadi rimanenti nei cicli successivi. Nel secondo ciclo, l'istruzione I_{j+1} viene prelevata mentre l'istruzione I_j è nello stadio Decodifica, dove i suoi operandi vengono pure letti dal banco di registri. Nel terzo ciclo, l'istruzione I_{j+2} viene prelevata mentre l'istruzione I_{j+1} è nello stadio Decodifica e l'istruzione I_j è nello stadio Elaborazione, dove viene realizzata un'operazione aritmetica o logica sui suoi operandi. Idealmente, questo modello di sovrapposizione dell'esecuzione sarebbe possibile per tutte le istruzioni. Sebbene ciascuna istruzione richieda cinque cicli per completare la sua esecuzione, le istruzioni vengono completate alla frequenza di una per ciclo.

6.2 Organizzazione in pipeline

La Figura 6.2 indica come l'organizzazione a cinque stadi nelle Figure 5.7 e 5.8 possa funzionare in pipeline. Nel primo stadio della pipeline, il contatore del programma (PC) viene usato per prelevare una nuova istruzione. Mentre altre istruzioni vengono prelevate, l'esecuzione procede attraverso stadi successivi. In ogni momento, ciascuno stadio della pipeline sta elaborando un'istruzione diversa. Informazioni quali indirizzi dei registri, dati immediati e operazioni da effettuare devono avanzare lungo la pipeline mentre ciascuna istruzione procede da uno stadio al successivo. Queste informazioni vengono mantenute nei *buffer interstadi*. Questi buffer includono i registri RA, RB, RM, RY e RZ in Figura 5.8, i registri IR e PC-Temp nelle Figure 5.9 e 5.10, e spazio aggiuntivo. I buffer interstadi sono usati come segue.

- Il buffer interstadi B1 alimenta lo stadio Decodifica con un'istruzione appena prelevata.
- Il buffer interstadi B2 alimenta lo stadio Elaborazione con i due operandi letti dal banco di registri, gli identificatori dei registri sorgenti e/o di destinazione, il valore immediato derivante dall'istruzione, il valore incrementato del PC usato come indirizzo di rientro da una chiamata a sottoprogramma, e le impostazioni di segnali di controllo determinate dal decodificatore dell'istruzione. Le impostazioni dei segnali di controllo avanzano lungo la pipeline per determinare l'operazione dell'ALU, l'operazione di memoria e un'eventuale scrittura nel banco di registri.

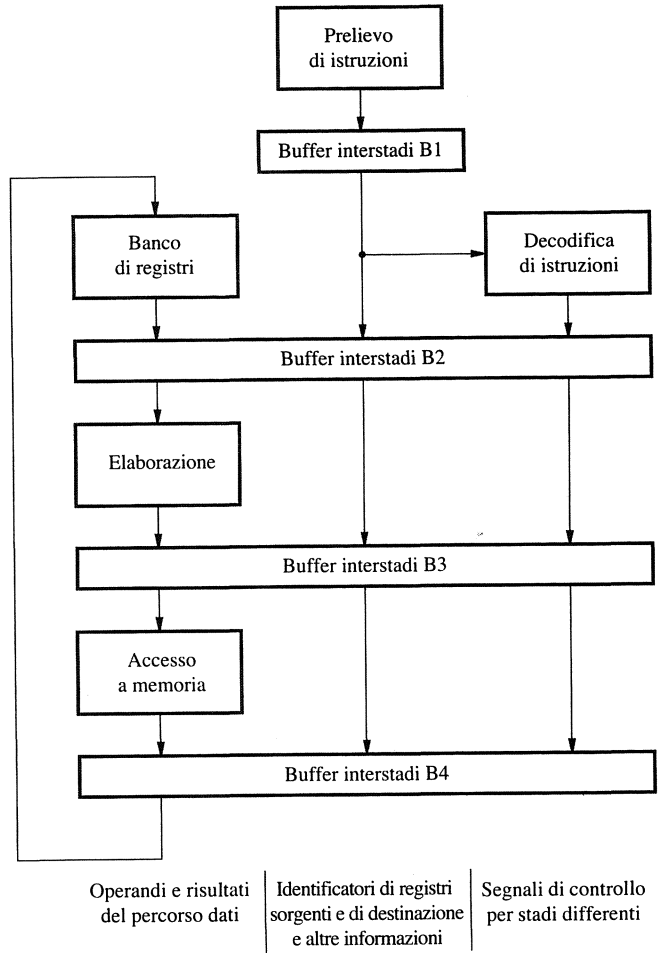


Figura 6.2 Una pipeline a cinque stadi.

- Il buffer interstadi B3 mantiene il risultato del calcolo svolto dall'ALU, che può essere un dato da scrivere nel banco di registri o un indirizzo che alimenta lo stadio Memoria. Nel caso di un accesso in scrittura alla memoria, il buffer B3 mantiene il dato da scrivere. Questo dato era stato letto dal banco di registri nello stadio Decodifica. Il buffer inoltre mantiene il valore del PC incrementato passato dallo stadio precedente, nel caso che serva come indirizzo di rientro per un'istruzione di chiamata a sottoprogramma.
- Il buffer interstadi B4 alimenta lo stadio Scrittura con un valore da scrivere nel banco di registri. Questo valore può essere il risultato dell'ALU dallo stadio Elaborazione, il risultato dello stadio di accesso alla Memoria, o il valore del PC incrementato che viene usato come indirizzo di rientro per un'istruzione di chiamata a sottoprogramma (si veda la Figura 6.2).

6.3 Problematiche del pipelining

La Figura 6.1 illustra la sovrapposizione ideale di tre istruzioni in sequenza. Tuttavia, alcune volte non è possibile avere l'ingresso di una nuova istruzione nella pipeline in ogni ciclo. Consideriamo il caso di due istruzioni, I_j e I_{j+1}, per le quali il registro di destinazione per l'istruzione I_j sia un registro sorgente per l'istruzione I_{j+1}. Il risultato dell'istruzione I_j non sarà scritto nel banco di registri fino al ciclo 5, ma l'istruzione I_{j+1} ne ha bisogno al ciclo 3, quando l'operando sorgente viene letto per l'elaborazione al successivo, terzo stadio dell'esecuzione. Se l'esecuzione procedesse come mostrato in Figura 6.1, il risultato dell'istruzione I_{j+1} non sarebbe corretto poiché l'operazione aritmetica sarebbe effettuata usando il vecchio valore del registro in questione. Per ottenere il risultato corretto è necessario attendere che il nuovo valore sia scritto nel registro dall'istruzione I_j. Quindi, l'istruzione I_{j+1} non può leggere il suo operando prima del ciclo 6, il che significa che deve rimanere in *stallo* nello stadio Decodifica per tre cicli. Mentre l'istruzione I_{j+1} è in stallo, l'istruzione I_{j+2} e tutte le istruzioni successive vengono analogamente differite. Le nuove istruzioni non possono far ingresso nella pipeline, e il tempo totale di esecuzione ne viene aumentato.

Qualsiasi condizione che causi uno stallo della pipeline viene detta *conflitto* (*hazard*). Abbiamo appena descritto un esempio di *conflitto su dati*, in cui il valore di un operando sorgente di un'istruzione non è disponibile quando sarebbe necessario. Altri conflitti provengono da ritardi della memoria, istruzioni di salto e limiti di risorse. I prossimi paragrafi descrivono più in dettaglio questi conflitti, insieme a tecniche per ridurne l'impatto sulle prestazioni.

6.4 Dipendenze di dato

Si considerino le due istruzioni in Figura 6.3:

Add	R2, R3, #100
Subtract	R9, R2, #30

Il registro di destinazione R2 per l'istruzione Add è un registro sorgente per l'istruzione Subtract. C'è una *dipendenza di dato* tra queste due istruzioni, poiché il registro R2 porta il dato dalla prima istruzione alla seconda. L'esecuzione in pipeline di queste due istruzioni è illustrata in Figura 6.3. L'istruzione Subtract rimane in stallo per tre cicli per ritardare la lettura del registro R2 fino al ciclo 6 quando un nuovo valore diverrà disponibile.

Si spiega adesso in maggiore dettaglio lo stallo. Il circuito di controllo deve anzitutto riconoscere la dipendenza di dato quando decodifica l'istruzione Subtract nel ciclo 3 confrontando l'identificatore del suo registro sorgente dal buffer interstadi B1 con l'identificatore

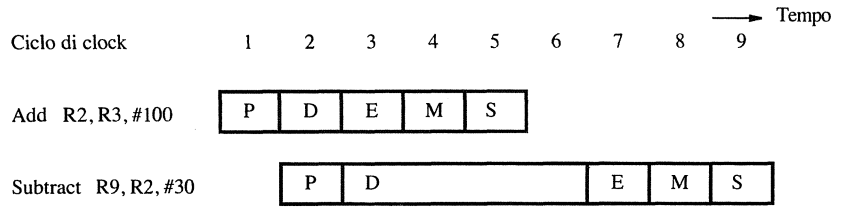


Figura 6.3 Stallo di una pipeline dovuto a dipendenza di dato.

del registro di destinazione dell'istruzione Add che si trova nel buffer interstadi B2. Quindi, l'istruzione Subtract deve essere trattenuta nel buffer interstadi B1 durante i cicli da 3 a 5. Nel frattempo, l'istruzione Add procede attraverso gli stadi rimanenti della pipeline. Nei cicli da 3 a 5, mentre l'istruzione Add va avanti, si possono impostare segnali di controllo nel buffer interstadi B2 per un'*istruzione nulla* o NOP (*No-operation*) implicita, che non modifica la memoria o il banco di registri. Ciascuna NOP crea un ciclo di clock di inattività, chiamato *bolla* (*bubble*), che attraversa gli stadi Elaborazione, Memoria e Scrittura fino alla fine della pipeline.

6.4.1 Inoltro di operandi

Gli stalli della pipeline dovuti a dipendenze di dato possono essere alleviati attraverso l'uso dell'*inoltro di operandi* (*operand forwarding*). Si consideri la coppia di istruzioni discussa sopra, per la quale la pipeline rimane in stallo per tre cicli per abilitare l'istruzione Subtract a usare il nuovo valore nel registro R2. Il valore desiderato è in realtà disponibile alla fine del ciclo 3, quando l'ALU completa l'operazione per l'istruzione Add. Questo valore viene caricato nel registro RZ in Figura 5.8, che è parte del buffer interstadi B3. Anziché far rimanere in stallo l'istruzione Subtract, l'hardware può *inoltrare* il valore dal registro RZ a dove è necessario nel ciclo 4, ovvero l'ingresso dell'ALU. La Figura 6.4 mostra l'esecuzione in pipeline quando si realizza l'inoltro. Le frecce mostrano che il risultato dell'ALU dal ciclo 3 viene usato come un ingresso per l'ALU nel ciclo 4.

La Figura 6.5 mostra la modifica necessaria al percorso dati di Figura 5.8 per rendere possibile l'inoltro. Un nuovo multiplatore, MuxA, viene anteposto all'ingresso InA dell'ALU, e il multiplatore esistente MuxB viene esteso con un altro ingresso. I multiplatori selezionano un valore letto dal banco di registri nel solito modo, o il valore disponibile nel registro RZ.

L'inoltro può anche essere esteso a un risultato nel registro RY in Figura 5.8. Questo risolverebbe una dipendenza di dato come quella che coinvolge il registro R2 nella sequenza di istruzioni sottostante:

Add	R2, R3, #100
Or	R4, R5, R6
Subtract	R9, R2, #30

Quando l'istruzione Subtract è nello stadio Elaborazione della pipeline, l'istruzione Or è nello stadio Memoria (nel quale nessuna operazione viene effettuata), mentre l'istruzione Add è nello stadio Scrittura. Il nuovo valore del registro R2 generato dall'istruzione Add è ora nel

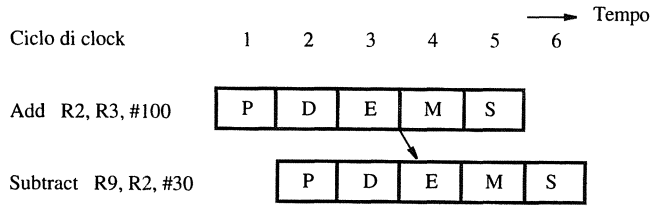


Figura 6.4 Eliminazione di uno stallo usando l'inoltro di operandi.

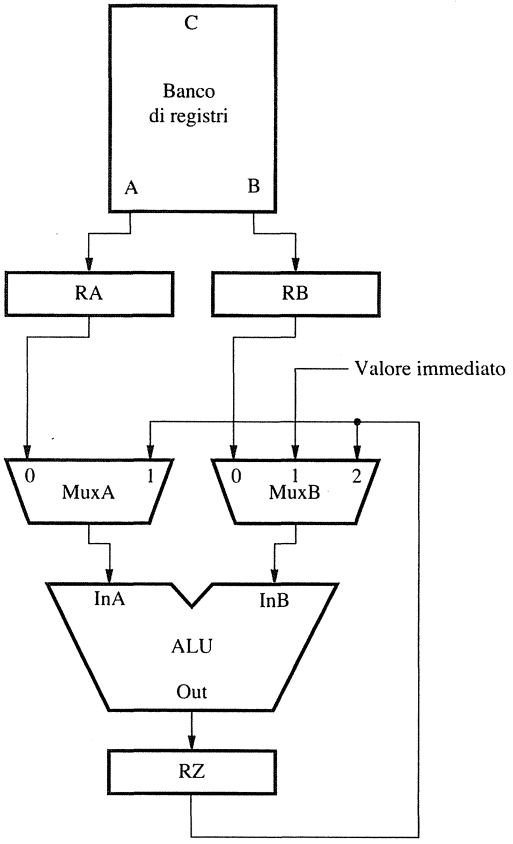


Figura 6.5 Modifica del percorso dati di Figura 5.8 per l'inoltro di dati dal registro RZ agli ingressi dell'ALU.

registro RY. Inoltrare questo valore dal registro RY all'ingresso InA dell'ALU evita lo stallo della pipeline.

MuxA richiede un altro ingresso per il valore di RY. Analogamente MuxB è esteso con un ulteriore ingresso.

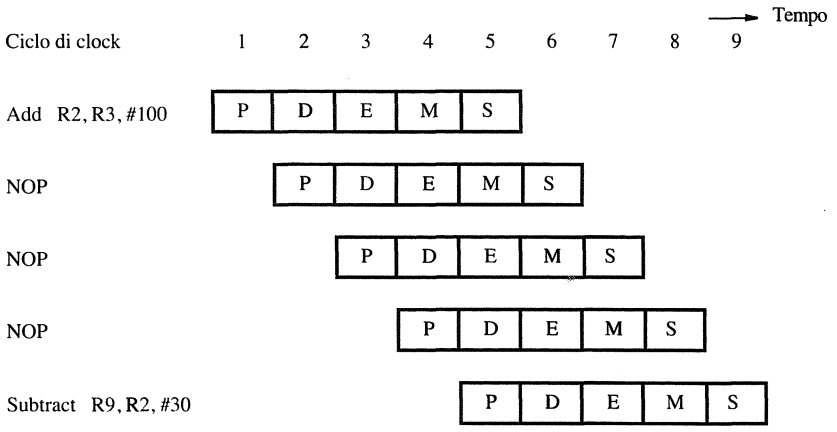
6.4.2 Gestione in software di dipendenze di dato

Le Figure 6.3 e 6.4 mostrano come le dipendenze di dato possano essere gestite dall'hardware del processore, mediante lo stallo della pipeline o l'inoltro dei dati. Un approccio alternativo è di lasciare al compilatore la rilevazione delle dipendenze di dato e la loro gestione.

Quando il compilatore identifica una dipendenza di dato tra due istruzioni successive I_j e I_{j+1}, può inserire tre istruzioni NOP esplicite tra esse. Le NOP introducono il ritardo necessario per abilitare l'istruzione I_{j+1} a leggere il nuovo valore dal banco di registri dopo che sia stato scritto. Per le istruzioni in Figura 6.4, il compilatore genererebbe la sequenza di


```
Add       R2, R3, #100
NOP
NOP
NOP
Subtract  R9, R2, #30
```

(a) Inserimento di istruzioni NOP per una dipendenza di dato



(b) Esecuzione in pipeline delle istruzioni

Figura 6.6 Uso di istruzioni NOP per la gestione in software di una dipendenza di dato.

istruzioni in Figura 6.6a. La Figura 6.6b mostra che le tre istruzioni NOP hanno lo stesso effetto, sul tempo di esecuzione, dello stallo in Figura 6.3.

Richiedere al compilatore di identificare dipendenze e inserire istruzioni NOP semplifica la realizzazione hardware della pipeline. Tuttavia, la dimensione del codice aumenta e il tempo di esecuzione non viene ridotto come invece avviene grazie all'inoltro di operandi. Il compilatore può cercare di *ottimizzare* il codice per migliorare le prestazioni e ridurre la dimensione del codice riordinando le istruzioni per spostare istruzioni utili nei posti delle NOP. Così facendo, il compilatore deve considerare le dipendenze di dato tra istruzioni, che vincolano la quantità di posti delle NOP che possono essere utilmente riempiti.

6.5 Ritardi della memoria

I ritardi dovuti ad accessi alla memoria sono un'altra causa di stallo della pipeline. Per esempio, un'istruzione Load può richiedere più di un ciclo di clock per ottenere il suo operando dalla memoria. Questo può avvenire poiché l'istruzione o i dati richiesti non vengono trovati

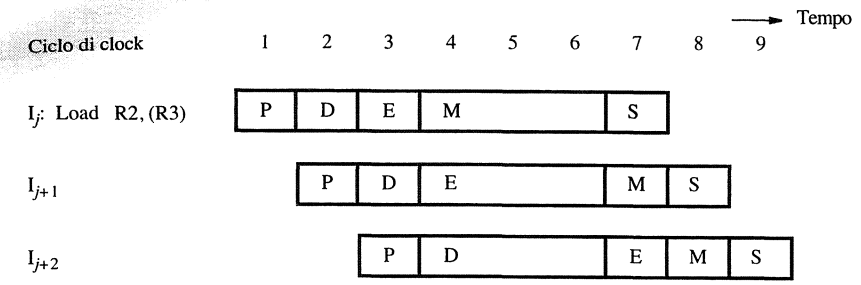


Figura 6.7 Stallo causato da un ritardo di accesso alla memoria per un'istruzione Load.

nella cache, dando come risultato una *cache miss*. La Figura 6.7 mostra l'effetto che un ritardo nell'accesso ai dati in memoria ha su un'esecuzione in pipeline. Un accesso alla memoria può richiedere dieci o più cicli. Per semplicità la figura mostra solo tre cicli. Una cache miss fa sì che tutte le istruzioni successive vengano ritardate. Un ritardo simile può essere provocato da una cache miss quando si preleva un'istruzione.

Un ulteriore tipo di stallo dovuto alla memoria avviene quando vi è una dipendenza di dato che coinvolge un'istruzione Load. Si considerino le istruzioni:

Load	R2, (R3)
Subtract	R9, R2, #30

Si assuma che il dato per l'istruzione Load venga trovato nella cache, quindi l'accesso all'operando richiede solo un ciclo. Il registro di destinazione R2 per l'istruzione Load è un registro sorgente per l'istruzione Subtract. L'inoltro degli operandi non può essere effettuato allo stesso modo come in Figura 6.4, poiché i dati letti dalla memoria (la cache, in questo caso) non saranno disponibili finché non saranno caricati nel registro RY all'inizio del ciclo 5. Quindi l'istruzione Subtract deve restare in stallo per un ciclo, come mostrato in Figura 6.8, per differire il lavoro dell'ALU. L'operando in memoria, che è adesso nel registro RY, può essere inoltrato all'ingresso dell'ALU nel ciclo 5.

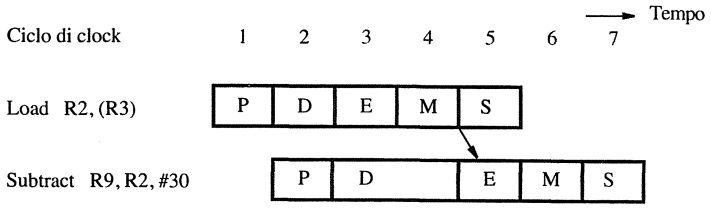


Figura 6.8 Stallo necessario a permettere l'inoltro a un'istruzione che segue un'istruzione Load.

Il compilatore può eliminare lo stallo di un ciclo per questo tipo di dipendenza di dato riordinando le istruzioni per inserire un'istruzione utile tra l'istruzione Load e l'istruzione che dipende dai dati letti dalla memoria. L'istruzione inserita riempie la bolla che sarebbe altrimenti creata. Se un'istruzione utile non può essere trovata dal compilatore allora l'hardware introduce lo stallo di un ciclo automaticamente. Se l'hardware del processore non gestisce le dipendenze allora il compilatore deve inserire un'istruzione NOP esplicita.

6.6 Ritardi nei salti

In un'esecuzione in pipeline ideale una nuova istruzione viene prelevata a ogni ciclo, mentre la precedente istruzione viene ancora decodificata. Le istruzioni di salto possono alterare la sequenza di esecuzione, ma devono prima essere eseguite per determinare se e dove saltare. Si esaminano adesso l'effetto delle istruzioni di salto e le tecniche che possono essere usate per ridurre il loro effetto sull'esecuzione in pipeline.

6.6.1 Salti incondizionati

La Figura 6.9 mostra l'esecuzione in pipeline di una sequenza di istruzioni, che inizia con un'istruzione di salto incondizionato, I_j. Le successive due istruzioni, I_{j+1} e I_{j+2}, sono immagazzinate negli indirizzi di memoria consecutivi che seguono I_j. La destinazione del salto è l'istruzione I_k. Come mostrato in Figura 5.15, l'istruzione di salto viene prelevata nel ciclo 1 e decodificata nel ciclo 2, e l'indirizzo di destinazione è calcolato nel ciclo 3. Quindi, l'istruzione I_k è prelevata nel ciclo 4, dopo che il contatore del programma è stato aggiornato con l'indirizzo di destinazione. Nell'esecuzione in pipeline, le istruzioni I_{j+1} e I_{j+2} vengono prelevate nei cicli 2 e 3, rispettivamente, prima che l'istruzione di salto sia decodificata e l'indirizzo di destinazione sia conosciuto. Esse devono essere scartate. Il ritardo di due cicli che ne deriva costituisce una *penalità di salto*.

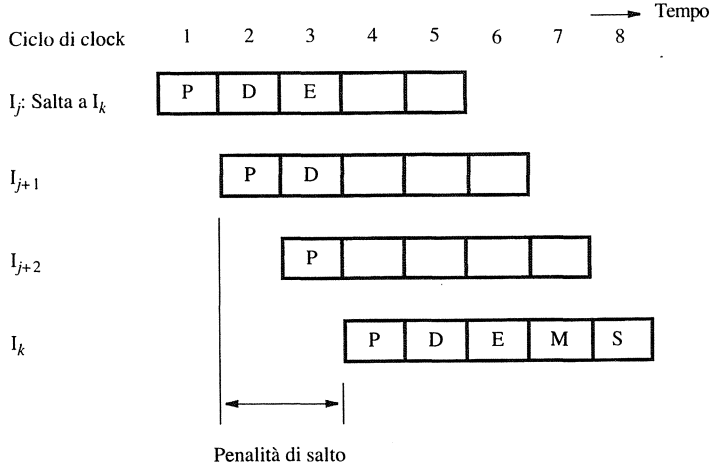


Figura 6.9 Penalità di salto quando l'indirizzo di destinazione è determinato nello stadio Elaborazione della pipeline.

Le istruzioni di salto si verificano frequentemente. Di fatto esse rappresentano circa il 20% del conteggio dinamico di istruzioni della maggior parte dei programmi (il conteggio dinamico è il numero di istruzioni eseguite, considerando il fatto che alcune istruzioni in un programma vengono eseguite più volte, a causa dei cicli). Con una penalità di salto di due cicli, la frequenza relativamente alta di istruzioni di salto potrebbe far aumentare del 40% il tempo di esecuzione per un programma. Perciò è importante trovare il modo di alleviare questo impatto sulle prestazioni.

Per ridurre la penalità di salto occorre anticipare, nella pipeline, il calcolo dell'indirizzo di destinazione. Anziché aspettare lo stadio Elaborazione, è possibile determinare l'indirizzo di destinazione e aggiornare il contatore del programma nello stadio Decodifica. Quindi, l'istruzione I_k può essere prelevata con un anticipo di un ciclo di clock, riducendo la penalità di salto a un ciclo, come mostrato in Figura 6.10. Questa volta, solo un'istruzione, I_{j+1}, viene prelevata inutilmente, poiché l'indirizzo di destinazione viene determinato nello stadio Decodifica.

L'hardware in Figura 5.10 deve essere modificato per realizzare questo cambiamento. Il sommatore nella figura è necessario per incrementare il PC in ciascun ciclo. Un secondo sommatore è necessario nello stadio Decodifica per calcolare l'indirizzo di destinazione per ciascuna istruzione. Quando il decodificatore di istruzioni determina che l'istruzione è in effetti un'istruzione di salto, l'indirizzo di destinazione calcolato sarà disponibile prima della fine del ciclo. Lo si può usare per prelevare l'istruzione di destinazione nel ciclo successivo.

6.6.2 Salti condizionati

Si consideri un'istruzione di salto condizionato come:

Branch_if_[R5]=[R6] CICLO

I passi di esecuzione per quest'istruzione sono mostrati in Figura 5.16. Il risultato del confronto nel terzo passo determina se effettuare il salto.

Per l'esecuzione in pipeline, la condizione di salto deve essere valutata quanto prima possibile per limitare la penalità di salto. È stato appena descritto come l'indirizzo di destinazione per un'istruzione di salto incondizionato possa essere determinato nello stadio Decodifica. Analogamente, il comparatore che valuta la condizione di salto può essere pure spostato nello stadio Decodifica, abilitando la decisione di salto condizionato a essere presa nello stesso momento in cui si determina l'indirizzo di destinazione. In questo caso, il comparatore usa i valori delle uscite A e B del banco di registri direttamente.

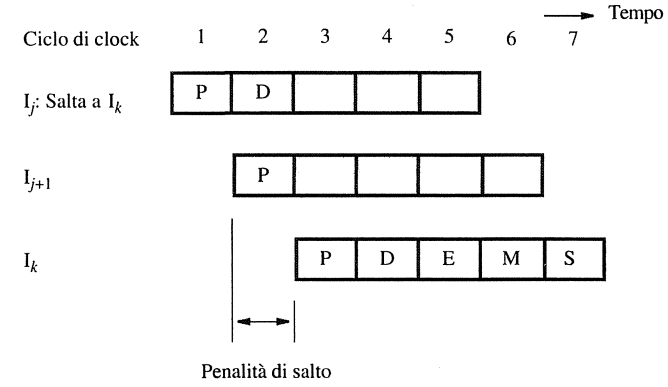


Figura 6.10 Penalità di salto quando l'indirizzo di destinazione è determinato nello stadio Decodifica della pipeline.

Anticipare la decisione di salto allo stadio Decodifica assicura una penalità di salto di un solo ciclo per tutte le istruzioni di salto. Nei prossimi due paragrafi si discuteranno tecniche aggiuntive che possono essere usate per alleviare ulteriormente l'effetto di salti sul tempo di esecuzione.

6.6.3 Il posto del ritardo di salto

Si consideri il frammento di programma mostrato in Figura 6.11a. Si assuma che l'indirizzo di destinazione del salto e la decisione per il salto siano determinati nello stadio Decodifica, nello stesso momento in cui l'istruzione I_{j+1} viene prelevata. L'istruzione di salto può far sì che l'istruzione I_{j+1} venga scartata, dopo che la condizione di salto sia stata valutata. Se la condizione è vera, allora si ha una penalità di salto di un ciclo prima che la corretta istruzione di destinazione I_k sia prelevata. Se la condizione è falsa, allora l'istruzione I_{j+1} viene eseguita, e non si ha penalità. In entrambi i casi, l'istruzione che segue immediatamente l'istruzione di salto viene sempre prelevata. In base a quest'osservazione, descriviamo una tecnica per ridurre la penalità delle istruzioni di salto.

La posizione che segue un'istruzione di salto è detta *posto del ritardo di salto*. Anziché eliminare in modo condizionale l'istruzione nel posto del ritardo, si può organizzare la pipeline così che la esegua comunque, sia che il salto venga effettuato o meno. L'istruzione nel posto del ritardo non può essere I_{j+1}, quella che può essere scartata in base alla condizione di salto.

```
                Add                 R7, R8, R9
                Branch_if_[R3]=0    DESTINAZIONE
                I j+1
                :
DESTINAZIONE:   I k
```

(a) Sequenza originaria di istruzioni contenente un'istruzione di salto condizionato

```
                Branch_if_[R3]=0    DESTINAZIONE
                Add                 R7, R8, R9
                I j+1
                :
DESTINAZIONE:   I k
```

(b) Ricollocazione dell'istruzione Add nel posto del ritardo dove viene sempre eseguita

Figura 6.11 Occupazione del posto del ritardo con un'istruzione utile.

Invece, il compilatore prova a reperire un'istruzione adatta per occupare il posto del ritardo, una che debba essere eseguita persino quando il salto viene effettuato. Può farlo ricollocando nel posto del ritardo una delle istruzioni che precedono l'istruzione di salto. Naturalmente, questo può essere fatto solo se si rispetta ogni dipendenza di dato che coinvolga l'istruzione da spostare. Se si trova un'istruzione utile, allora non ci sarà nessuna penalità di salto. Se a causa dei vincoli derivanti dalle dipendenze di dato nessuna istruzione utile può essere innestata nel posto del ritardo, allora si dovrà usare una NOP. In questo caso, vi sarà una penalità di un ciclo sia che il salto venga effettuato o meno.

Per le istruzioni in Figura 6.11a, l'istruzione Add può essere trasferita senza rischi nel posto del ritardo di salto, come mostrato in Figura 6.11b. L'istruzione Add è sempre prelevata ed eseguita, anche se il salto viene effettuato. L'istruzione I_{j+1} è prelevata solo se il salto non viene effettuato. Logicamente, l'esecuzione procede come se l'istruzione di salto fosse inserita dopo l'istruzione Add.

In altre parole, il salto è effettuato un'istruzione dopo rispetto a dove l'istruzione di salto appare nella sequenza di istruzioni. Questa tecnica è detta *salto differito*.

L'efficacia del salto differito dipende da quanto spesso il compilatore può riordinare istruzioni per riempire utilmente il posto del ritardo. I dati sperimentali raccolti da molti programmi indicano che il compilatore può riempire un posto di ritardo di salto nel 70% o più di casi.

6.6.4 Predizione di salti

La discussione precedente mostra che prendere la decisione di salto nel ciclo 2 dell'esecuzione di un'istruzione di salto riduce la penalità di salto. Tuttavia, anche così, l'istruzione che segue immediatamente l'istruzione di salto è ancora prelevata nel ciclo 2 e potrebbe essere scartata. La decisione di prelevare quest'istruzione è in effetti presa nel ciclo 1, quando il PC viene incrementato mentre l'istruzione di salto sta per essere prelevata. Quindi, per ridurre ulteriormente la penalità di salto, il processore ha bisogno di determinare in anticipo che un'istruzione che sta per essere prelevata è un'istruzione di salto e *predire* il suo risultato per determinare quale istruzione debba essere prelevata nel ciclo 2. In questo paragrafo, si descriveranno dapprima diversi metodi di predizione di salto, quindi si discuterà di come la predizione venga effettuata nel ciclo 1 mentre un'istruzione di salto viene prelevata.

Predizione statica di salto

La forma più semplice di predizione di salto è assumere che il salto non sia effettuato e prelevare la successiva istruzione in ordine sequenziale di indirizzo. Se la predizione è corretta, si completa l'esecuzione dell'istruzione prelevata e non c'è penalità. Tuttavia, se si stabilisce che deve essere effettuato il salto, l'istruzione che è stata prelevata viene scartata e si preleva quella che si trova alla corretta destinazione del salto. Predizioni non corrette vanno incontro alla piena penalità di salto. Questo semplice approccio è una forma di *predizione statica di salto*. La stessa scelta (assumere di non effettuare il salto) è fatta ogni volta che si incontra un salto condizionato.

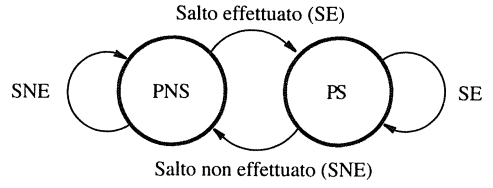
Se i risultati dei salti fossero casuali, allora metà di tutti i salti condizionati sarebbero indovinati. In questo caso, assumere costantemente che i salti non siano effettuati significa avere un'accuratezza della predizione del 50%. Tuttavia un salto all'indietro alla fine di un ciclo viene effettuato la maggior parte delle volte. Per tale salto, una migliore accuratezza può essere ottenuta predicendo che è probabile che il salto venga fatto, quindi le istruzioni vengono prelevate usando l'indirizzo di destinazione appena esso diventa noto. Analogamente per un salto in avanti all'inizio di un ciclo, la predizione di non saltare porta a una buona accuratezza di predizione. Il processore può determinare la predizione statica per saltare o meno valutando il segno dello spiazzamento di salto. In alternativa, la codifica macchina di un'istruzione di salto può includere un bit che indica se il salto dovrebbe essere predetto come da fare o non fare. L'impostazione di questo bit può essere specificata dal compilatore.

Predizione dinamica di salto

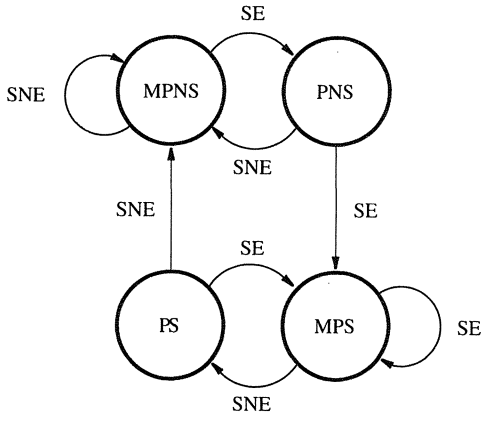
Per migliorare ulteriormente l'accuratezza della predizione è possibile usare l'attuale comportamento di salto per influenzare la predizione, determinando una *predizione dinamica di salto*. L'hardware del processore valuta la probabilità di effettuare un certo salto tenendo traccia delle decisioni di salto ogni volta che un'istruzione di salto sia eseguita.

Nella sua forma più semplice, un algoritmo di predizione dinamica può usare il risultato dell'esecuzione più recente di un'istruzione di salto. Il processore assume che la prossima volta che l'istruzione sarà eseguita, la decisione sul salto sia probabilmente la stessa dell'ultima volta. Quindi l'algoritmo può essere descritto dalla macchina a due stati in Figura 6.12a. I due stati sono:

PS	–	Probabilmente salta
PNS	–	Probabilmente non salta



(a) Un algoritmo a due stati



(b) Un algoritmo a quattro stati

Figura 6.12 Rappresentazione di algoritmi di predizione di salti con macchine a stati finiti.

Si supponga che l'algoritmo parta nello stato PNS. Quando l'istruzione di salto viene eseguita e il salto viene effettuato, la macchina si sposta nello stato PS. Altrimenti rimane nello stato PNS. La prossima volta che si incontra la stessa istruzione, la predizione di salto è affermativa se la macchina è nello stato PS, altrimenti è negativa.

Questo semplice schema, che richiede solo un singolo bit per rappresentare la storia dell'esecuzione per un'istruzione di salto, funziona bene all'interno di un ciclo. Una volta che si entra nel ciclo, la decisione per l'istruzione di salto che controlla il ciclo sarà sempre la stessa eccetto che per l'ultima passata del ciclo. Quindi, ciascuna predizione per l'istruzione di salto sarà corretta eccetto che nell'ultima passata, per cui la macchina a stati della storia del salto verrà mutata allo stato opposto. Purtroppo, questo significa che la prossima volta che si entra in questo stesso ciclo – assumendo che ci sia ancora una passata attraverso il ciclo – la macchina a stati condurrà alla predizione sbagliata per la prima passata.

Quindi esecuzioni ripetute dello stesso ciclo portano a predizioni sbagliate nella prima e nell'ultima passata.

Migliore accuratezza di predizione può essere ottenuta tenendo più informazioni sulla storia dell'esecuzione. In Figura 6.12b è mostrato un algoritmo che usa quattro stati. I quattro stati sono:

MPS	–	Molto probabilmente salta
PS	–	Probabilmente salta
PNS	–	Probabilmente non salta
MPNS	–	Molto probabilmente non salta

Si assuma nuovamente che lo stato dell'algoritmo sia inizialmente impostato a PNS. Dopo che l'istruzione di salto viene eseguita, e se il salto è effettuato, lo stato viene cambiato a MPS; altrimenti, viene cambiato a MPNS. Con l'avanzamento dell'esecuzione del programma e mano a mano che la stessa istruzione di salto viene incontrata più volte, lo stato dell'algoritmo di predizione cambia come mostrato. La predizione di salto è affermativa se lo stato è MPS o PS, altrimenti è negativa.

Si consideri cosa accade quando si esegue un ciclo. Si assuma che l'istruzione di salto sia alla fine del ciclo e che il processore imposti lo stato iniziale dell'algoritmo a PNS. Nella prima passata, la predizione (negativa) sarà sbagliata, quindi lo stato sarà cambiato a MPS. In tutte le passate successive, la predizione sarà corretta, eccetto che per l'ultima passata. In quel momento, lo stato cambierà a PS. Quando si entra nel ciclo una seconda volta, la predizione nella prima passata sarà di fare il salto, che sarà corretta se c'è più di un'iterazione. Quindi, l'esecuzione ripetuta dello stesso ciclo porta a solo una predizione sbagliata nell'ultimo passo, quella della prima esecuzione.

Buffer di destinazione di salto per predizione dinamica

Nella discussione precedente, si è notato che l'indirizzo di destinazione del salto e la decisione di salto possono entrambi essere determinati nello stadio Decodifica della pipeline, che è il ciclo 2 dell'esecuzione dell'istruzione. L'istruzione che viene prelevata nello stesso ciclo potrebbe o meno essere quella che deve essere eseguita dopo l'istruzione di salto. Essa potrebbe essere scartata, nel qual caso l'istruzione corretta sarà prelevata nel ciclo 3. Come usare la predizione di salto per ottenere prestazioni migliori?

L'idea per migliorare le prestazioni è di fare aumentare la probabilità che l'istruzione prelevata nel ciclo 2 sia quella corretta. Questo può essere ottenuto solo se la predizione di salto avviene nel ciclo 1, nello stesso momento in cui l'istruzione di salto viene prelevata. Per renderlo possibile, il processore ha bisogno di mantenere più informazioni sulla storia di esecuzione. Le informazioni richieste sono di solito immagazzinate in una memoria piccola e veloce chiamata *buffer di destinazione di salto*.

Il buffer di destinazione di salto identifica le istruzioni di salto dai loro indirizzi. Per ogni istruzione di salto eseguita, il processore salva nel buffer l'indirizzo dell'istruzione e il risultato della decisione di salto. Le informazioni sono organizzate nella forma di una tabella di ricerca, in cui ciascun elemento include:

- l'indirizzo dell'istruzione di salto;
- uno o due bit di stato per l'algoritmo di predizione di salto;
- l'indirizzo di destinazione del salto.

Con queste informazioni, il processore può identificare le istruzioni di salto e ottenere i bit di predizione corrispondenti in base all'indirizzo dell'istruzione che sta prelevando.

Ogni volta che il processore preleva una nuova istruzione, controlla se il buffer di destinazione di salto contiene un elemento con lo stesso indirizzo dell'istruzione. Se un elemento con tale indirizzo viene trovato, significa che l'istruzione che si sta prelevando è un'istruzione di salto. Il processore può quindi usare i bit di stato per predire se il salto sarà fatto. Allo stesso tempo ottiene l'indirizzo di destinazione, ovvero quando preleva l'istruzione di salto nel ciclo 1. Nel ciclo 2, il processore usa il risultato della predizione di salto per prelevare la prossima istruzione. Certamente, deve anche determinare la decisione corrente di salto e l'indirizzo di destinazione per valutare se i valori predetti erano corretti. Se lo sono, l'esecuzione continua senza penalità. Altrimenti l'istruzione che è stata appena prelevata viene scartata e l'istruzione corretta viene prelevata nel ciclo 3. Il principale valore del buffer di destinazione di salto è che le informazioni sullo stato necessarie per la predizione del salto e l'indirizzo di destinazione di un'istruzione di salto sono ottenute nello stesso momento in cui l'istruzione di salto viene prelevata.

Programmi di grandi dimensioni hanno molte istruzioni di salto. Un buffer di destinazione di salto che avesse spazio sufficiente a contenere informazioni per tutte sarebbe grande e la ricerca veloce sarebbe difficile. Per questo motivo, la tabella ha una dimensione limitata, contenente informazioni solo per le istruzioni di salto eseguite più di recente. Gli elementi nella tabella vengono sostituiti mano a mano che si eseguono altre istruzioni. Tipicamente la dimensione della tabella è dell'ordine di 1024 elementi.

6.7 Limiti di risorse

L'uso della pipeline permette l'esecuzione sovrapposta di istruzioni, ma la pipeline va in stallo quando non ci sono risorse hardware sufficienti per permettere a tutte le attività di procedere concorrentemente. Se due istruzioni necessitano di accedere alla stessa risorsa nello stesso ciclo di clock, un'istruzione deve andare in stallo per permettere all'altra istruzione di usare la risorsa. Questo può essere evitato provvedendo a un hardware aggiuntivo.

Tali stalli possono verificarsi in un calcolatore che abbia una singola cache che supporti solo un accesso per ciclo. Se entrambi gli stadi Prelievo e Memoria della pipeline sono collegati alla cache, allora non è possibile che le attività di entrambi gli stadi procedano simultaneamente. Normalmente, lo stadio Prelievo accede alla cache in ogni ciclo. Tuttavia, quest'attività deve andare in stallo per un ciclo quando c'è anche un'istruzione Load o Store nello stadio Memoria che ha bisogno di accedere alla cache. Se il 25% di tutte le istruzioni eseguite sono

istruzioni Load o Store, questi stalli aumentano il tempo di esecuzione del 25%. L'uso di cache separate per istruzioni e dati permette agli stadi Prelievo e Memoria di procedere simultaneamente senza stalli.

6.8 Valutazione della prestazione

Per un processore senza pipeline, il tempo di esecuzione, T, di un programma che abbia un conteggio dinamico di N istruzioni macchina è dato dalla formula:

$$T = \frac{N \times S}{R}$$

nota anche come *equazione di base della prestazione*, dove S indica il numero medio di cicli di clock necessari per completare l'istruzione macchina (comprensivo di prelievo ed esecuzione) e R è la frequenza di clock del processore. Per quantificare la prestazione del processore, più che al tempo di esecuzione di base è preferibile ricorrere alla frequenza di operazione o *throughput* P, che dà il numero (medio) di istruzioni eseguite dal processore nell'unità di tempo, di solito espresso in istruzioni al secondo. Per l'esecuzione senza pipeline, il throughput P_{np} è dato da:

$$P_{np} = \frac{R}{S}$$

Qui si vuole esaminare come la metodologia di pipelining agisca sul throughput permettendo di aumentarlo. Il processore presentato nel Capitolo 5 usa cinque cicli per eseguire tutte le istruzioni. Quindi S è pari a 5 se non vi sono cache miss.

Il pipelining migliora le prestazioni sovrapponendo l'esecuzione di istruzioni in successione, incrementando così il throughput di istruzioni, sebbene una singola istruzione venga eseguita nello stesso numero di cicli. Per la pipeline a cinque stadi descritta in questo capitolo, ciascuna istruzione viene eseguita in cinque cicli, ma una nuova istruzione può idealmente entrare nella pipeline a ogni ciclo. Quindi, in assenza di stalli, S è pari a 1 e il throughput usando la pipeline è:

$$P_p = R$$

Una pipeline con cinque stadi può potenzialmente far aumentare il throughput di un fattore 5. Generalizzando, il guadagno potenziale di una pipeline a $n > 1$ stadi vale n. Sembrerebbe dunque che aumentando a piacere il numero di stadi n il guadagno potrebbe aumentare illimitatamente. L'intuizione fisica dice però che ciò è impossibile e pertanto sorgono due domande.

- Fino a quanto può essere effettivamente realizzata la crescita potenziale del guadagno?
- Qual è un buon valore per il numero di stadi n, e come lo si può determinare?

Ogni volta che una pipeline ha uno stallo o le istruzioni vengono scartate, il throughput è ridotto al di sotto del suo valore ideale. Quindi, le prestazioni di una pipeline sono fortemente influenzate da fattori come stalli dovuti a dipendenze di dato tra istruzioni e penalità dovute a salti. Le cache miss aumentano ulteriormente il tempo di esecuzione. Pertanto prima si esamineranno gli effetti dei conflitti sulla prestazione e poi si ritornerà sulla questione di come determinare il numero di stadi.

6.8.1 Effetti di stalli, penalità e cache miss

Nei paragrafi precedenti si sono esaminati in modo qualitativo gli effetti di varie categorie di conflitti; qui si determineranno in modo quantitativo. La pipeline a cinque stadi coinvolge operazioni di accesso alla memoria negli stadi Prelievo e Memoria, e operazioni con l'ALU nello stadio Elaborazione. Le operazioni con il ritardo più lungo determinano la durata del ciclo e quindi la frequenza di clock R. Per un processore che ha la cache sul chip, le operazioni di accesso alla memoria hanno un ritardo piccolo quando le istruzioni o i dati desiderati vengono trovati nella cache. Il ritardo attraverso l'ALU è probabilmente il parametro critico. Se questo ritardo è di 2 ns, allora R = 500 MHz e il throughput ideale di istruzioni attraverso la pipeline è P_p = 500 MIPS (milioni di istruzioni al secondo).

Si consideri un processore con l'inoltro di operandi in hardware, come spiegato nel Paragrafo 6.4.1. Ciò significa che non vi sono penalità dovute a dipendenze di dato, eccetto nel caso di istruzioni Load. Per valutare l'effetto di stalli non dovuti a cache miss, si può considerare quanto spesso un'istruzione Load sia seguita immediatamente da un'altra istruzione che usi il risultato dell'accesso alla memoria. Il Paragrafo 6.5 spiega che in tali casi è necessario uno stallo di un ciclo di clock. Mentre l'esecuzione in pipeline ideale ha $S = 1$, gli stalli dovuti a tali istruzioni Load hanno l'effetto di incrementare S di una quantità δ_{stallo}. Per esempio, si assuma che le istruzioni Load costituiscano il 25% del conteggio dinamico delle istruzioni, e si assuma che il 40% di queste istruzioni Load sia seguito da un'istruzione dipendente. In questi casi è necessario uno stallo di un ciclo. Quindi l'incremento rispetto al caso ideale $S = 1$ è:

$$\delta_{stallo} = 0{,}25 \times 0{,}40 \times 1 = 0{,}10$$

ovvero, il tempo di esecuzione T viene incrementato del 10% e il throughput viene ridotto a:

$$P_p = \frac{R}{1+\delta_{stallo}} = \frac{R}{1{,}1} = 0{,}91R$$

Il compilatore può migliorare le prestazioni riducendo il numero di volte in cui un'istruzione Load sia seguita immediatamente da un'istruzione dipendente. Uno stallo viene eliminato ogni volta che il compilatore può trasferire senza rischi un'istruzione vicina nella posizione tra l'istruzione Load e l'istruzione dipendente.

Si considerino adesso le penalità dovute a predizioni errate di salti durante l'esecuzione del programma. Quando sia la decisione di salto sia l'indirizzo di destinazione di salto vengono determinati nello stadio Decodifica della pipeline, la penalità di salto è di un ciclo. Si assuma che i salti costituiscano il 20% del conteggio dinamico di istruzioni in un programma e che l'accuratezza media di predizione per istruzioni di salto sia del 90%. In altre parole, il 10% di tutte le istruzioni di salto eseguite è soggetto a una penalità di un ciclo dovuta a predizione errata. L'incremento nel numero medio di cicli per istruzione dovuto a penalità di salto è:

$$\delta_{penalità_salto} \times 0{,}20 \times 0{,}10 \times 1 = 0{,}02$$

Un'elevata accuratezza della predizione è vantaggiosa al fine di limitare l'effetto avverso di questa penalità sulle prestazioni.

Gli stalli dovuti a istruzioni Load e le penalità dovute a errate predizioni di salto sono indipendenti. Quindi il loro effetto sulle prestazioni è additivo. La somma di δ_{stallo} e $\delta_{penalità_salto}$ determina l'incremento nel numero di cicli, S, l'incremento del tempo di esecuzione, T, e la riduzione del throughput, P_p.

L'effetto di cache miss sulle prestazioni può essere valutato considerando la frequenza della loro occorrenza. Il tempo di accesso alla memoria principale più lenta è una penalità

che manda in stallo la pipeline per p_m cicli ogni volta che si verifica una cache miss. Una frazione m_i di tutte le istruzioni prelevate è soggetta a cache miss. Una frazione d di tutte le istruzioni sono istruzioni Load o Store, e una frazione m_d di queste istruzioni è soggetta a cache miss. L'incremento rispetto al caso ideale di $S = 1$ dovuto a cache miss è:

$$\delta_{\text{miss}} = (m_i + d \times m_d) \times p_m$$

Si supponga che il 5% di tutte le istruzioni prelevate sia soggetto a cache miss, che il 30% di tutte le istruzioni eseguite siano istruzioni Load o Store e che il 10% degli accessi ai loro operandi dati sia soggetto a cache miss. Si assuma che la penalità di accesso alla memoria principale per una cache miss sia 10 cicli. L'incremento rispetto al caso ideale $S = 1$ dovuto a cache miss in questo caso è fornito da:

$$\delta_{\text{miss}} = (0{,}05 + 0{,}30 \times 0{,}10) \times 10 = 0{,}8$$

Confrontato con δ_{stallo} per le dipendenze di dato e con $\delta_{\text{penalità_salto}}$ per predizioni errate di salto, l'effetto di una memoria principale lenta per cache miss è più significativo in questo esempio. Quando tutti i fattori vengono messi insieme, S è incrementato dal valore ideale di 1 a $1 + \delta_{\text{stallo}} + \delta_{\text{penalità_salto}} + \delta_{\text{miss}}$. Il contributo di cache miss è spesso quello dominante.

6.8.2 Numero di stadi della pipeline

Il fatto che la pipeline a n stadi riesca a far aumentare il throughput di un fattore pari a n nel caso migliore sembra suggerire che il numero di stadi debba essere elevato. Ma così si trascura il fatto che aumentando la lunghezza o, come anche si dice, la *profondità della pipeline* (*pipeline depth*), si eleva anche la probabilità di stallo, perché la pipeline conterrebbe più istruzioni macchina in trattamento contemporaneo. Ne seguirebbe che istruzioni altrimenti piuttosto distanti, pertanto non problematiche, si troverebbero insieme nella pipeline e le rispettive eventuali dipendenze darebbero luogo a conflitti. Inoltre, la penalità di salto può essere maggiore di un ciclo se una pipeline più lunga trasferisce la decisione sul salto a uno stadio successivo. Tutto ciò attenua la crescita del guadagno al crescere di n, mentre il costo associato diventa sempre meno giustificabile.

Un altro fattore limitativo importante è dato dal ritardo intrinseco delle operazioni di base del processore. Di solito l'unità funzionale con ritardo maggiore è l'ALU, che in molti processori determina la durata del ciclo di clock. Le altre operazioni di base del processore sono abbastanza facilmente scomponibili in fasi di durata non superiore a un'operazione di ALU. È comunque possibile scomporre anche l'unità ALU suddividendola in stadi. Alcune recenti realizzazioni di processori hanno usato venti o più stadi di pipeline per ridurre notevolmente la durata del ciclo. La realizzazione di pipeline lunghe usando la moderna tecnologia permette di raggiungere frequenze di clock di vari GHz.

6.9 Funzionamento superscalare

Il massimo throughput di un processore con pipeline è di un'istruzione per ciclo di clock. In un approccio più aggressivo, si può equipaggiare il processore con svariate unità di esecuzione, ciascuna delle quali può essere organizzata in pipeline, per accrescere l'abilità del processore di gestire più istruzioni in parallelo. Con questa organizzazione, più istruzioni cominciano l'esecuzione nello stesso ciclo di clock, ma in unità di esecuzione diverse, e si dice che il processore usa l'*emissione multipla* (*multiple-issue*). Tali processori possono ottenere un throughput di esecuzione di istruzioni maggiore di un'istruzione per ciclo. Sono conosciuti come processori *superscalari*. Molti processori moderni ad alte prestazioni usano questo approccio.

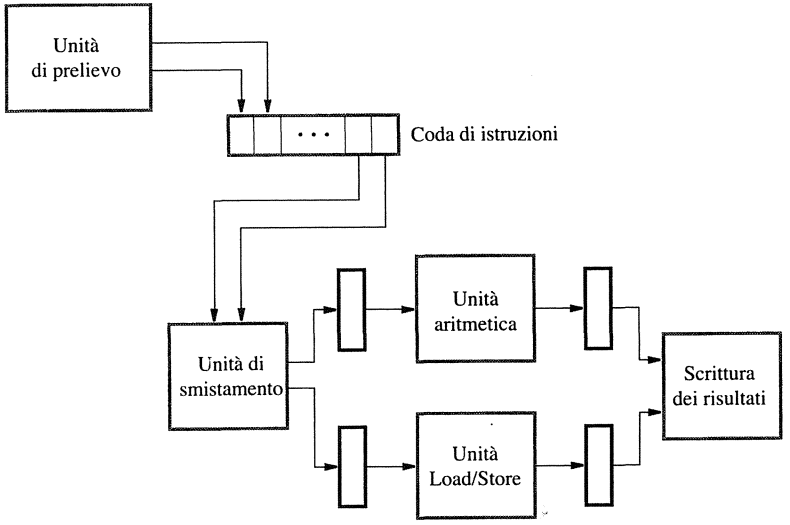


Figura 6.13 Un processore superscalare con due unità di esecuzione.

Per abilitare l'esecuzione a emissione multipla, un processore superscalare ha un'*unità di prelievo* (*fetch unit*) più elaborata, che preleva due o più istruzioni per ciclo prima che siano necessarie e le pone in una coda di istruzioni. Un'unità separata, chiamata *unità di smistamento* (*dispatch unit*), prende due o più istruzioni dalla testa della coda, le decodifica e le invia alle unità di esecuzione appropriate. Alla fine della pipeline, un'altra unità è responsabile per la scrittura dei risultati nel banco di registri. La Figura 6.13 mostra un processore superscalare con questa organizzazione. Esso incorpora due unità di esecuzione, una per istruzioni aritmetiche e un'altra per istruzioni Load e Store. Le operazioni aritmetiche richiedono normalmente solo un ciclo, quindi la prima unità di esecuzione è semplice. Poiché le istruzioni Load e Store coinvolgono il calcolo di un indirizzo per il modo con indice prima di ciascun accesso alla memoria, l'unità Load/Store ha una pipeline a due stadi.

L'organizzazione in Figura 6.13 porta a considerare alcune importanti implicazioni per il banco di registri. Un'istruzione aritmetica e un'istruzione Load o Store devono ottenere tutti i loro operandi dal banco di registri quando vengono smistate nello stesso ciclo alle due unità di esecuzione. Il banco di registri deve ora avere quattro porte di uscita invece delle due porte necessarie nella semplice pipeline.

Similmente, un'istruzione aritmetica e un'istruzione Load devono scrivere i loro risultati nel banco di registri quando vengono completate nello stesso ciclo. Quindi ora il banco di registri deve avere due porte di ingresso invece di una singola porta di ingresso, sufficiente nella pipeline semplice. Vi è anche la potenziale complicazione di due istruzioni che vengono completate nello stesso momento con lo stesso registro di destinazione per i loro risultati. Questa complicazione è evitata, se possibile, inoltrando le istruzioni in modo da prevenire che accada. Altrimenti, un'istruzione va in stallo per assicurare che i risultati siano scritti nei registri di destinazione nello stesso ordine della sequenza di istruzioni originale del programma.

Per illustrare l'esecuzione superscalare nel processore in Figura 6.13, si consideri la seguente sequenza di istruzioni:

Add	R2, R3, #100
Load	R5, 16(R6)
Subtract	R7, R8, R9
Store	R10, 24(R11)

La Figura 6.14 mostra come verrebbero eseguite queste istruzioni. L'unità di prelievo prende due istruzioni a ogni ciclo. Le istruzioni vengono decodificate e i loro registri sorgenti vengono letti nel ciclo successivo. Quindi, esse sono inviate alle unità aritmetiche e Load/Store. Le operazioni aritmetiche possono cominciare a ogni ciclo. Un'istruzione Load o Store può anche iniziare a ogni ciclo, poiché la pipeline a due stadi sovrappone il calcolo dell'indirizzo per un'istruzione Load o Store con l'accesso alla memoria per la precedente istruzione Load o Store.

Al completamento delle istruzioni in ciascuna unità, il banco di registri permette la scrittura di due risultati nello stesso ciclo poiché i registri di destinazione sono differenti.

6.9.1 Salti e dipendenze di dato

In assenza di qualsiasi istruzione di salto e di dipendenze di dato tra istruzioni, il throughput viene massimizzato alternando istruzioni che possono essere inviate simultaneamente a diverse unità di esecuzione. Tuttavia, i programmi contengono istruzioni di salto che cambiano il flusso di esecuzione, e dipendenze di dato tra istruzioni che impongono vincoli di esecuzione sequenziale. Un processore superscalare deve assicurare che le istruzioni siano eseguite nella sequenza appropriata. Inoltre ritardi della memoria dovuti a cache miss possono occasionalmente mandare in stallo il prelievo e l'invio di istruzioni. Di conseguenza il throughput effettivo è tipicamente più basso del massimo possibile. Le difficoltà presentate da istruzioni di salto e dipendenze di dato possono essere affrontate tramite hardware aggiuntivo. Si considerino inizialmente le istruzioni di salto e successivamente le questioni derivanti dalle dipendenze di dato.

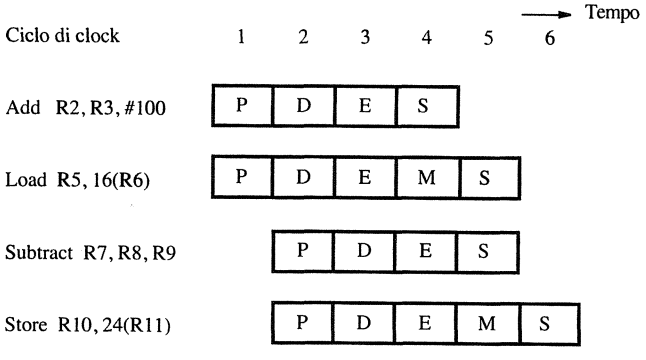


Figura 6.14 Un esempio di flusso di istruzioni nel processore di Figura 6.13.

L'unità di prelievo gestisce istruzioni di salto mano a mano che determina quali istruzioni inserire nella coda di smistamento. Essa deve determinare sia la decisione di salto sia la destinazione per ciascuna istruzione di salto. La decisione di salto può dipendere dal risultato di una precedente istruzione che è ancora in coda o smistata di recente. Mandare in stallo l'unità di prelievo finché il risultato non sia disponibile può ridurre in modo significativo il throughput e quindi non è un approccio desiderabile. Invece, è meglio impiegare la predizione di salto. Poiché lo scopo è di ottenere un alto throughput, la predizione è usata insieme a una tecnica detta *esecuzione speculativa*. In questa tecnica, istruzioni successive basate su una predizione non confermata vengono prelevate, smistate e possibilmente eseguite, ma etichettate come speculative cosicché esse stesse e i loro risultati possano essere scartati se la predizione non risultasse corretta. Hardware aggiuntivo è richiesto per mantenere le informazioni su istruzioni eseguite in modo speculativo e per assicurare che i registri o le locazioni di memoria non siano modificati finché la validità della predizione non venga confermata. Hardware aggiuntivo è pure necessario per assicurare che vengano prelevate e smistate le corrette istruzioni nel caso di predizione errata.

Le dipendenze di dato tra istruzioni impongono vincoli di ordine di esecuzione. Un approccio semplice è di inviare istruzioni dipendenti in sequenza alla stessa unità di esecuzione, che ne preserverà l'ordine. Tuttavia, può accadere che istruzioni dipendenti siano da inviare a differenti unità di esecuzione. Per esempio, il risultato di un'istruzione Load inviato all'unità Load/Store in Figura 6.13 può essere necessario a un'istruzione Add inviata all'unità aritmetica. Poiché le unità operano indipendentemente e poiché altre istruzioni possono essere già state inoltrate verso esse, non vi è garanzia di quando il risultato necessario all'istruzione Add sia generato dall'istruzione Load. È necessario un meccanismo per assicurare che un'istruzione dipendente attenda che i suoi operandi diventino disponibili. Quando un'istruzione è inoltrata a un'unità di esecuzione, viene fatta attendere in un buffer finché tutti i risultati necessari da altre istruzioni non siano stati generati. Tali buffer sono detti *stazioni di prenotazione* e sono usati per mantenere informazioni e operandi rilevanti a ciascuna istruzione smistata. I risultati da ciascuna unità di esecuzione vengono trasmessi a tutte le stazioni di prenotazione, nelle quali ciascun risultato è etichettato con un identificatore di registro. Questo abilita le stazioni di prenotazione a riconoscere un risultato dal quale dipende un'istruzione nel buffer. Quando vi è un'etichetta corrispondente, l'hardware copia il risultato nella stazione di prenotazione contenente l'istruzione. Il circuito di controllo inizia l'esecuzione di un'istruzione nel buffer solo quando questa ha tutti i suoi operandi.

In un processore superscalare a emissione multipla, l'effetto svantaggioso degli stalli diventa ancora più pronunciato che in un processore con pipeline a emissione singola. Il compilatore può evitare molti stalli attraverso attenta selezione e ordinamento delle istruzioni. Per esempio, per il processore in Figura 6.13, il compilatore dovrebbe operare per ottenere l'alternanza tra istruzioni aritmetiche e di memoria. Questo permette all'unità di invio di tenere entrambe le unità impegnate per la maggior parte del tempo.

6.9.2 Esecuzione fuori ordine

Le istruzioni in Figura 6.14 sono inviate nello stesso ordine in cui compaiono nel programma. Tuttavia la loro esecuzione può essere completata secondo una sequenza diversa. Per esempio, l'istruzione Subtract scrive sul registro R7 nello stesso ciclo in cui l'istruzione Load precedentemente prelevata scrive sul registro R5. Se l'accesso alla memoria per l'istruzione Load richiedesse più di un ciclo per essere completato, l'esecuzione dell'istruzione Subtract sarebbe completata prima dell'istruzione Load. Questo genere di situazione comporta problemi?

Si è già discusso delle questioni sollevate da dipendenze fra istruzioni. Per esempio, se un'istruzione I_{j+1} dipende dal risultato dell'istruzione I_j, l'esecuzione di I_{j+1} sarà differita se il

risultato non è disponibile quando è necessario. Finché tali dipendenze sono gestite correttamente, non vi è motivo per rallentare l'esecuzione di un'istruzione non correlata. Se non vi è dipendenza tra una coppia di istruzioni, l'ordine in cui l'esecuzione viene completata non è importante.

Tuttavia, una nuova complicazione emerge quando si considera la possibilità che un'istruzione causi un'eccezione. Per esempio, l'istruzione Load in Figura 6.14 può tentare un accesso proibito alla memoria per un operando dato non allineato. Al momento in cui questa operazione proibita verrà riconosciuta, l'istruzione Subtract che viene prelevata dopo l'istruzione Load può avere già modificato il suo registro di destinazione. L'esecuzione del programma è adesso in uno stato inconsistente. L'istruzione che ha causato l'eccezione nella sequenza originale viene identificata, ma un'istruzione successiva in quella sequenza è stata eseguita fino al completamento. Se questa situazione viene permessa, si dice che il processore ha *eccezioni imprecise*.

L'alternativa di *eccezioni precise* richiede hardware aggiuntivo. Per garantire uno stato consistente quando avvengono le eccezioni, i risultati dell'esecuzione delle istruzioni devono essere scritti nelle locazioni di destinazione esclusivamente secondo l'ordine nel programma. Questo significa che bisogna differire la scrittura nel registro R7 per l'istruzione Subtract in Figura 6.14 finché il registro R5 per l'istruzione Load non sia stato aggiornato. L'unità aritmetica in Figura 6.13 deve trattenere il risultato dell'istruzione Subtract, o il risultato deve essere posto in un registro temporaneo finché le istruzioni precedenti non abbiano scritto i loro risultati. Se avviene un'eccezione durante l'esecuzione di un'istruzione, tutte le successive istruzioni e i loro risultati nei buffer saranno scartati.

È più facile fornire eccezioni precise nel caso di interruzioni esterne. Quando un'interruzione esterna viene ricevuta, l'unità di smistamento blocca la lettura di nuove istruzioni dalla coda delle istruzioni, e le istruzioni che rimangono in coda vengono scartate. Tutte le istruzioni la cui esecuzione è in corso continuano fino al completamento. A questo punto, il processore e tutti i suoi registri sono in uno stato consistente e l'elaborazione dell'interruzione può cominciare.

6.9.3 Completamento dell'esecuzione

Per migliorare le prestazioni, a un'unità di esecuzione dovrebbe essere permesso di eseguire qualunque istruzione i cui operandi siano pronti nella sua stazione di prenotazione. Questo può portare all'esecuzione fuori ordine di istruzioni. Tuttavia, le istruzioni devono essere completate nell'ordine del programma per permettere eccezioni precise. Questi che sembrano requisiti in conflitto possono essere risolti se si permette all'esecuzione di procedere fuori ordine, ma scrivendo i risultati in registri temporanei. I contenuti di questi registri sono successivamente trasferiti ai registri permanenti nel corretto ordine del programma. Quest'ultimo passo è spesso chiamato il passo dell'*impegno* o *commitment*, poiché l'effetto di un'istruzione è irreversibile dopo quel punto. Se un'istruzione causasse un'eccezione, i risultati di qualsiasi istruzione successiva che sia stata però già eseguita sarebbero ancora nei registri temporanei e potrebbero essere scartati senza problemi. I risultati che sarebbero normalmente scritti sulla memoria sarebbero anch'essi scritti temporaneamente su buffer, e potrebbero essere pure scartati senza problemi.

Un registro temporaneo che viene assegnato per il risultato di un'istruzione assume il ruolo del registro permanente di cui sta mantenendo il dato. Il suo contenuto viene inoltrato a qualsiasi istruzione successiva che utilizzi il registro permanente originario durante quel periodo. Questa tecnica è detta *ridenominazione dei registri*. Ci possono essere tanti registri temporanei quanti sono i registri permanenti, o ci possono essere meno registri temporanei, allocati al bisogno per l'associazione a diversi registri permanenti.

Quando viene permessa l'esecuzione fuori ordine, un'unità di controllo speciale è necessaria per garantire il commitment in ordine. Questa è detta *unità di commitment*. Essa usa una coda separata chiamata *buffer di riordino* per determinare quali istruzioni siano le prossime nella sequenza dei commitment. Le istruzioni vengono poste in coda esattamente nell'ordine del programma, che è lo stesso in cui vengono smistate per l'esecuzione. Quando un'istruzione raggiunge la testa di questa coda e l'esecuzione di tale istruzione è stata completata, i risultati corrispondenti vengono trasferiti dai registri temporanei ai registri permanenti e l'istruzione viene rimossa dalla coda. Tutte le risorse che erano state assegnate all'istruzione, inclusi i registri temporanei, vengono rilasciate. Si dice che l'istruzione è stata *ritirata* a questo punto. Poiché un'istruzione viene ritirata solo quando è in testa alla coda, tutte le istruzioni che erano state smistate prima devono anche essere state ritirate. Quindi le istruzioni possono completare l'esecuzione fuori ordine, ma sono ritirate nell'ordine del programma.

6.9.4 Funzionamento dello smistamento

Si ritorni adesso al modo in cui si opera lo smistamento. Quando prende decisioni sull'invio, l'unità di smistamento deve assicurarsi che tutte le risorse necessarie all'esecuzione di un'istruzione siano disponibili. Per esempio, poiché il risultato di un'istruzione può dover essere scritto in un registro temporaneo, dovrebbe essercene uno disponibile, e viene prenotato per essere usato da questa istruzione come parte della funzione di smistamento. Deve inoltre esserci spazio disponibile nella stazione di prenotazione di un'appropriata unità di esecuzione. Infine, deve anche essere disponibile una locazione nel buffer di riordino per il successivo commitment dei risultati dell'istruzione. Quando tutte le risorse necessarie sono state prenotate, l'istruzione viene inviata.

Si potrebbero smistare fuori ordine le istruzioni? Per esempio, l'invio dell'istruzione Load in Figura 6.14 può essere differito poiché non vi è spazio nella stazione di prenotazione dell'unità Load/Store per effetto di una cache miss in un'istruzione precedentemente inviata. Sarebbe da inviare in alternativa l'istruzione Subtract? In principio questo è possibile, purché tutte le risorse necessarie all'istruzione Load, incluso un posto nel buffer di riordino, siano prenotate per essa. Questo è essenziale per assicurare che tutte le istruzioni alla fine siano ritirate nell'ordine corretto e che non si cada in una situazione di blocco.

Un *deadlock* o *blocco* è una situazione che può avvenire quando due unità, A e B, usano una risorsa condivisa. Si supponga che l'unità B non possa completare la sua operazione finché l'unità A non abbia completato la sua. Allo stesso tempo, sia stata assegnata all'unità B una risorsa di cui l'unità A abbia necessità. Se accede questo, nessuna delle due unità può completare la sua operazione. L'unità A è in attesa della risorsa di cui ha necessità, ma che è trattenuta dall'unità B. Al contempo, l'unità B attende che l'unità A finisca prima di poter completare la sua operazione e rilasciare quella risorsa.

Come esempio di deadlock quando si smistano istruzioni fuori ordine, si consideri un processore superscalare che abbia un solo registro temporaneo. Quando l'istruzione Subtract in Figura 6.14 viene inviata prima dell'istruzione Load, il registro temporaneo viene prenotato per essa. L'istruzione Load non può essere inviata poiché le occorre lo stesso registro temporaneo, che a sua volta non diventerà disponibile finché l'istruzione Subtract non sia stata ritirata. Poiché l'istruzione Subtract non può essere ritirata prima dell'istruzione Load, si ha un deadlock.

Per prevenire i deadlock l'unità di smistamento deve considerare molti fattori. Quindi, emettere istruzioni fuori ordine probabilmente incrementa in modo significativo la complessità dell'unità di smistamento, e può anche comportare un aumento del tempo necessario a prendere decisioni di smistamento. Inviare le istruzioni in ordine evita questa complessità. In questo caso, l'ordine di programnma delle istruzioni è imposto al momento in cui le istruzioni vengono smistate e anche al momento in cui vengono ritirate. Tra questi due eventi, l'esecuzione di varie istruzioni attraverso più unità di esecuzione può procedere fuori ordine, soggetta solo alle dipendenze tra esse.

Un commento finale sui processori superscalari riguarda il numero di unità di esecuzione. Il processore in Figura 6.13 ha un'unità aritmetica e un'unità Load/Store. Per prestazioni più alte, i processori superscalari moderni spesso hanno due unità aritmetiche per operazioni con interi e un'unità aritmetica separata per operazioni con numeri in virgola mobile. L'unità in virgola mobile ha il suo banco di registri. Molti processori includono anche un'unità vettoriale per aritmetica su interi o su numeri in virgola mobile, che tipicamente svolge da due a otto operazioni in parallelo. Tale unità può avere anche un banco di registri dedicato. Una singola unità Load/Store supporta tipicamente tutti gli accessi alla memoria verso o dai banchi di registri per interi, numeri in virgola mobile e unità vettoriale. Per tenere molte unità di esecuzione occupate, i processori moderni possono prelevare quattro o più istruzioni nello stesso momento e porle nella coda di istruzioni; parimenti quattro o più istruzioni possono essere smistate alle unità di esecuzione dalla testa della coda di istruzioni.

6.10 Pipeline nei processori CISC

L'insieme di istruzioni di un processore RISC rende relativamente facile la realizzazione della pipeline. Tutte le istruzioni hanno la lunghezza di una parola e le informazioni sugli operandi sono tipicamente nelle stesse posizioni della parola per istruzioni diverse. Nessuna istruzione richiede più di un operando in memoria. Solo le istruzioni Load e Store accedono a operandi in memoria, tipicamente usando solo l'indirizzamento con indice. Tutte le altre istruzioni operano su operandi nei registri. La pipeline a cinque stadi descritta in questo capitolo è adatta a queste caratteristiche di istruzioni in stile RISC.

Per avere la pipeline nei processori CISC, emergono alcune complicazioni dovute a istruzioni che hanno dimensione variabile, più operandi in memoria e modi di indirizzamento complessi, e usano codici di condizione. Il prelievo di istruzioni che occupano più di una parola può richiedere vari cicli. Inoltre, la variabilità nella dimensione e nel formato delle istruzioni complica sia la decodifica sia l'accesso agli operandi, come pure la gestione della coda di smistamento, in un processore superscalare.

La disponibilità di modi di indirizzamento più complessi come l'autoincremento o l'autodecremento introduce *effetti collaterali* quando si eseguono le istruzioni. Si ha un effetto collaterale quando, oltre alla locazione dell'operando di destinazione, anche un'altra locazione è affetta dall'esecuzione. Per esempio, l'istruzione:

```
Move   R5, (R8)+
```

ha un effetto collaterale, poiché la sua esecuzione interviene non solo sul registro di destinazione R5 ma anche sul registro sorgente R8, con l'operazione di autoincremento. Se un'istruzione successiva dipendesse dal valore nel registro R8, questa dipendenza dovrebbe essere gestita con hardware aggiuntivo allo stesso modo di una dipendenza che coinvolga il registro di destinazione R5. Può essere necessario mettere in stallo la pipeline o inoltrare il nuovo valore. In un processore superscalare tale dipendenza richiede l'uso di registri temporanei e la ridenominazione di registri come discusso nel Paragrafo 6.9.3.

Anche i codici di condizione introducono effetti collaterali. Per esempio, nella sequenza di istruzioni:

```
Compare     R7, R8
Branch>0    DESTINAZIONE
```

il risultato dell'istruzione Compare ha un effetto collaterale sui codici di condizione. L'istruzione Branch, a sua volta, dipende implicitamente da questo effetto collaterale. Un registro

per i codici di condizione può essere incluso facilmente in una pipeline semplice, come quella di Figura 6.2, poiché a ogni ciclo si effettua solo un'operazione con l'ALU. Tuttavia, in un processore superscalare con più unità di esecuzione, molte istruzioni possono essere in vari stadi di esecuzione, e due o più operazioni di ALU possono essere svolte in ciascun ciclo. Le dipendenze indotte da effetti collaterali in relazione ai codici di condizione richiedono l'uso di ulteriori registri temporanei e della ridenominazione di registri.

Infine, si consideri la seguente sequenza di istruzioni in stile CISC:

```
Move    (R2), (R3)
Move    (R4), R5
```

La prima istruzione Move richiede due accessi per gli operandi in memoria, mentre la seconda istruzione Move ne richiede solo uno. L'esecuzione di queste istruzioni in una pipeline come quella in Figura 6.2 richiede che l'hardware aggiuntivo porti in stallo la seconda istruzione Move, cosicché la prima istruzione Move possa completare l'accesso in memoria per i suoi due operandi. Analogamente, in un processore superscalare come quello in Figura 6.13, l'unità Load/Store deve portare in stallo la sua pipeline interna.

Le istruzioni in stile CISC complicano la pipeline. Questo è stato uno dei principali motivi per lo sviluppo dell'approccio RISC. Tuttavia, processori con pipeline sono stati realizzati per insiemi di istruzioni in stile CISC, che erano stati inizialmente introdotti prima della diffusione del pipelining. Alcuni esempi includono processori basati sugli insiemi di istruzioni ColdFire e Intel, discussi negli approfondimenti applicativi dei Capitoli 2 e 3. I processori ColdFire sono principalmente intesi per applicazioni embedded, mentre i processori Intel sono di uso generale. Di conseguenza, l'uso di pipeline nei processori ColdFire è meno esteso che nei processori Intel.

6.10.1 Pipeline nei processori ColdFire

Le realizzazioni dei processori ColdFire denominate V1 e V2 hanno due pipeline in serie con un buffer FIFO (*first-in first-out*) tra esse. Una pipeline a due stadi per il prelievo pone le istruzioni nel buffer. Questo quindi alimenta una successiva pipeline a due stadi che esegue le istruzioni. Le istruzioni che coinvolgono operazioni solo sui registri o da registri a memoria attraversano solo una volta i due stadi di esecuzione. Le istruzioni che coinvolgono operazioni da memoria a registro o da memoria a memoria devono fare due passaggi attraverso gli stadi di esecuzione.

Versioni successive di realizzazioni del processore ColdFire usano una sistemazione similare del buffer tra le due pipeline, ma incorporano varie migliorie per ottenere prestazioni più alte. Per esempio, la pipeline per il prelievo di istruzioni nella versione V4 viene estesa a quattro stadi e include la predizione dei salti. La pipeline per l'esecuzione è estesa a cinque stadi. Gli stadi iniziali sono usati per il calcolo di indirizzi, e gli stadi successivi sono usati per le operazioni aritmetiche e logiche. Questa separazione di funzioni permette una forma limitata di elaborazione superscalare. In alcuni casi, un'istruzione Move e un'altra istruzione possono essere emesse verso la pipeline di esecuzione nello stesso ciclo. Le realizzazioni della versione V5 hanno due pipeline di esecuzione distinte basate sull'organizzazione della V4. Esse operano una vera elaborazione superscalare.

6.10.2 Pipeline nei processori Intel

I processori Intel ottengono alte prestazioni con l'esecuzione superscalare e pipeline profonde. Per esempio, le architetture Core 2 e Core i7 hanno una larghezza di emissione multipla di

quattro istruzioni e una pipeline a 14 stadi. Si usano la predizione di salti, la ridenominazione dei registri, l'esecuzione fuori ordine e altre tecniche.

Per ridurre la complessità interna, le istruzioni in stile CISC vengono dinamicamente convertite dall'hardware in micro-operazioni in stile RISC più semplici. Queste micro-operazioni sono quindi emesse verso le unità di esecuzione per completare i compiti specificati dalle istruzioni in stile CISC originarie. Questo approccio preserva la compatibilità del codice e allo stesso tempo rende possibile l'uso delle sofisticate tecniche di miglioramento delle prestazioni che sono state sviluppate per gli insiemi di istruzioni in stile RISC. In alcuni casi, micro-operazioni si rifondono insieme in una macro-operazione per una gestione più efficiente. Per esempio, in un programma che contiene istruzioni originarie in stile CISC, un'istruzione di confronto che ha effetti sui codici di condizione è spesso seguita da un'istruzione di salto. L'hardware potrebbe inizialmente convertire le istruzioni di confronto e di salto in micro-operazioni separate, ma successivamente le fonderebbe in un'operazione aggregata di confronto e salto, il cui funzionamento riflette ciò che tipicamente si incontra in un insieme di istruzioni in stile RISC.

Sommario

In questo capitolo sono state introdotte due importanti caratteristiche per il miglioramento delle prestazioni, il pipelining e l'emissione multipla. Il pipelining permette ai processori di ottenere un throughput di istruzioni che si avvicina a un'istruzione per ciclo di clock. La combinazione di emissione multipla e pipelining permette di avere il funzionamento superscalare, con un throughput di svariate istruzioni per ciclo di clock.

Comunque per ottenere il guadagno potenzialmente garantito dalla metodologia di pipelining si devono considerare attentamente i tre aspetti seguenti:

- il progetto dell'insieme di istruzioni macchina del processore;
- il progetto della struttura della pipeline;
- il progetto di compilatori mirati per il processore.

Tra tutti e tre gli aspetti sussistono forti interdipendenze. Il successo nell'aumentare la prestazione dell'esecuzione dipende in modo critico da quanto bene si siano comprese tali interdipendenze e da quanta parte di tale analisi sia confluita nel progetto del processore. Nei processori moderni dotati di pipeline, il fattore chiave per un progetto di successo passa per una buona ed equilibrata composizione dell'insieme di istruzioni.

Vi sono molte fonti che forniscono dettagli aggiuntivi sugli argomenti presentati in questo capitolo. Il testo [1] tratta il pipelining e il testo [2] tratta i processori superscalari.

Note bibliografiche

1. Patterson D.A., Hennessy J.L., *Struttura e progetto dei calcolatori*, 3ª edizione, Zanichelli, Bologna 2010.
2. Shen J.P., Lipasti M.H., *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill, 2005.

7 Sistema di ingresso e uscita

In questo capitolo

- L'hardware necessario per l'accesso ai dispositivi di I/O
- Il funzionamento sincrono e asincrono del bus
- I circuiti di interfaccia
- Gli standard commerciali, come USB, SAS, e PCI Express

Una delle caratteristiche fondamentali di un calcolatore è la sua abilità di trasferire dati su dispositivi di I/O. Questa abilità di comunicazione permette a un operatore umano, per esempio, di usare una tastiera e uno schermo per elaborare testo e grafica. Si fa un notevole uso di calcolatori per comunicare con altri calcolatori tramite Internet e per accedere a informazioni sparse in tutto il mondo. Nelle applicazioni *embedded* (incorporate), i calcolatori sono meno visibili ma ugualmente importanti. Essi sono parte integrante di apparecchiature domestiche, attrezzature per la produzione, veicoli, telefoni cellulari e terminali di banche e punti di vendita. In tali applicazioni, l'ingresso a un calcolatore può provenire dal tocco di un pannello, un commutatore di un sensore, una macchina fotografica digitale, un microfono, o un allarme antiincendio. L'uscita può essere costituita da caratteri o numeri che devono essere visualizzati, un segnale sonoro da inviare a un altoparlante, o un comando in codifica digitale per cambiare la velocità di un motore, aprire una valvola, o indurre un certo movimento di un robot.

Un calcolatore dovrebbe avere l'abilità di scambiare informazioni con una vasta varietà di dispositivi. In molti casi, il processore è pienamente coinvolto in questi scambi. Tuttavia, i trasferimenti di dati possono anche avvenire direttamente tra dispositivi di I/O, come i dischi fissi magnetici, e la memoria principale, con solo il minimo coinvolgimento del processore. Questa possibilità sarà investigata nel prossimo capitolo dedicato al sistema di memoria.

Il Capitolo 3 presenta i trasferimenti ingresso/uscita di dati che avvengono tra il processore e i registri nelle interfacce dei dispositivi di I/O dal punto di vista del programmatore. In questo capitolo si discutono i dettagli dell'hardware necessario per rendere possibili tali trasferimenti.

Per trasferire dati fra processore, memoria e dispositivi di I/O si usa una rete di interconnessione. Nel seguito se ne descrive una di uso comune, detta *bus*.

7.1 Struttura a bus

Il bus mostrato in Figura 7.1 è una semplice struttura che realizza la rete di interconnessione di Figura 3.1. Solo una coppia di unità sorgente/destinazione può usare questo bus per trasferire dati in un certo momento.

Il bus consta di tre gruppi di linee, usati per il trasporto di indirizzi, dati e segnali di controllo. Le interfacce dei dispositivi di I/O sono collegate a queste linee, come mostrato in Figura 7.2 per un dispositivo di ingresso. A ciascun dispositivo di I/O è assegnato un gruppo

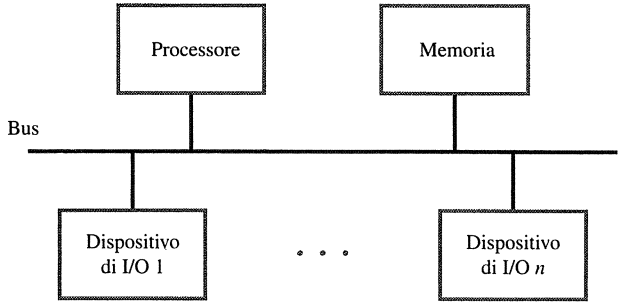


Figura 7.1 Una struttura a bus singolo.

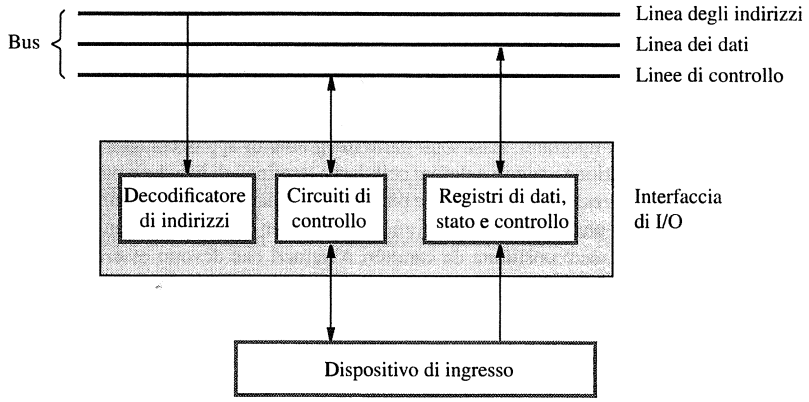


Figura 7.2 Interfaccia di I/O per un dispositivo di ingresso.

di indirizzi univoco per i registri nella sua interfaccia. Quando il processore pone un particolare indirizzo sulle linee degli indirizzi, tale indirizzo viene esaminato dai decodificatori di indirizzo di tutti i dispositivi collegati al bus. Il dispositivo che riconosce come proprio questo indirizzo risponde ai comandi emessi sulle linee di controllo. Il processore usa le linee di controllo per richiedere un'operazione di lettura o scrittura, e i dati richiesti vengono trasferiti sulle linee dei dati.

Quando i dispositivi di I/O e la memoria condividono lo stesso spazio di indirizzamento, tale organizzazione è detta *unificazione degli spazi di indirizzamento di memoria e I/O* (*memory-mapped I/O*), come descritto nel Paragrafo 3.1. Qualsiasi istruzione macchina che può accedere alla memoria può essere usata per trasferire dati da e verso un dispositivo di I/O. Per esempio, se il dispositivo di ingresso in Figura 7.2 è una tastiera e se DATAIN è il suo registro dati, l'istruzione:

Load R2, DATAIN

legge i dati da DATAIN e li pone nel registro del processore R2. Analogamente, l'istruzione:

Store R2, DATAOUT

invia il contenuto del registro R2 alla locazione DATAOUT, che può essere il registro dati dell'interfaccia di uno schermo. I registri di stato e di controllo contengono informazioni rilevanti per il funzionamento del dispositivo di I/O. Il decodificatore dell'indirizzo, i registri di dati e di stato e il circuito di controllo necessario per coordinare trasferimenti di I/O costituiscono il circuito di interfaccia del dispositivo.

7.2 Funzionamento del bus

Processore, memoria e periferiche sono tutti collegati tramite un bus comune, il bus del calcolatore o di sistema, la cui funzione principale è di mettere a disposizione un percorso per trasferire dati tra le varie unità menzionate. Il bus contiene anche le linee di controllo per gestire i meccanismi di interruzione e arbitraggio. Nel paragrafo vengono illustrati i protocolli di bus per trasferimento di dati. Il protocollo di bus è l'insieme di regole che governano il comportamento dei vari dispositivi collegati al bus; per esempio esse stabiliscono quando mandare informazioni sul bus (indirizzo o dato), attivare linee di controllo e così via.

Il bus è un fascio di linee da un bit ciascuna e si divide in tre gruppi principali: linee di indirizzo, di dato e di controllo (chiamati anche bus di indirizzo, di dato e di controllo). Le linee di controllo comandano il tipo di operazione da eseguire, per esempio lettura o scrittura di memoria oppure di periferica. Per quanto attiene le operazioni di I/O, di solito tra le linee di controllo ce n'è una chiamata genericamente R/$\overline{W}$ (*read / write*), dove *lettura* e *scrittura* sono attive alta e bassa, rispettivamente: se la linea vale 1, il processore, o comunque l'unità che in quel momento ha ruolo di master, sta chiedendo l'operazione di lettura da memoria o da periferica, se vale 0 sta chiedendo scrittura in memoria o su periferica. Se il bus è capace di trasferire dati di diverse dimensioni, per esempio byte, parola o parola doppia, tra le linee di controllo ce ne sono alcune che specificano la dimensione del dato corrente. Se gli spazi di indirizzamento di memoria e I/O sono unificati, come è spesso il caso, nel bus non si vede alcuna reale distinzione tra le operazioni di lettura da memoria piuttosto che da periferica, e tra quelle di scrittura in memoria piuttosto che su periferica.

Le linee di controllo servono anche per scandire il tempo e sincronizzare le varie unità. Essenzialmente ciò che occorre è segnalare l'istante di tempo quando processore o periferica devono emettere il dato sul bus o acquisirlo dal bus. Sono stati inventati parecchi protocolli diversi atti a risolvere il problema, che si dividono nelle due famiglie seguenti: *bus sincrono* (*synchronous bus*) e *bus asincrono* (*asynchronous bus*).

Quando c'è trasferimento di dati tra due unità una di esse deve avere *ruolo di master* (*master unit*). L'unità master o *principale* è quella che dà avvio all'operazione di I/O mandando sul bus il comando di lettura o scrittura: essa è dunque l'*iniziatrice* dell'operazione. Di solito l'unità master è il processore, ma altri dispositivi possono assumerne di tanto in tanto il ruolo, come si vedrà nel Paragrafo 7.3. L'altra unità che partecipa al trasferimento viene chiamata *unità slave* (*slave unit*) o *unità secondaria* e rappresenta l'*obiettivo* dell'operazione.

7.2.1 Bus sincrono

Il concetto di bus sincrono è molto semplice: il bus contiene una linea che distribuisce a tutte le unità collegate il segnale di scansione del tempo, o *segnale di clock* (*clock signal*). Di solito

il segnale di clock è una forma d'onda quadra e si conviene che uno dei due fronti definisca l'istante di tempo di passaggio dal ciclo di clock corrente a quello successivo; per esempio potrebbe trattarsi del fronte di salita. L'operazione di trasferimento tramite bus è divisa in *cicli di bus* (*bus cycle*); il ciclo di bus può durare uno o più cicli di clock, secondo il tipo di protocollo. I protocolli di bus più semplici stabiliscono che il ciclo di bus duri un solo ciclo di clock e che l'operazione duri un solo ciclo di bus, ovvero un solo ciclo di clock; protocolli sofisticati ammettono cicli di bus estesi su più cicli di clock e perfino operazioni estese su più cicli di bus. Tipicamente la frequenza di clock di bus va da poche unità di MHz fino ad alcune centinaia di MHz.

La Figura 7.3 dà il diagramma temporale dell'operazione di lettura da memoria o periferica, nella forma più semplice. I fasci di linee di indirizzo e di dato sono rappresentati tramite linea doppia, per indicare che alcune di esse sono a 0 e altre a 1; si incrociano le linee negli istanti di tempo in cui alcune (o tutte) transitano di valore. Se occorre specificare lo stato di ciascuna linea del fascio si scrive la sequenza di bit corrispondente all'interno della linea doppia. Quando si vuole rappresentare il fatto che il fascio non ha valore ben definito, o che lo ha ma non è significativo, o che le linee si trovano in stato di alta impedenza cioè sono elettricamente scollegate, invece della linea doppia si traccia una sola linea intermedia, oppure si ombreggia lo spazio tra le due linee. Siffatta convenzione per rappresentare lo stato di fasci di linee è molto comune e utilissima.

Ecco come si svolge l'operazione di lettura. All'istante di tempo t_0 l'unità master, iniziatrice dell'operazione, emette sulle linee di indirizzo la stringa di bit che rappresenta l'indirizzo della parola di memoria o della periferica e attiva nelle linee di controllo il comando di lettura (cioè pone $R/\overline{W} = 1$); se occorre attiva anche le linee di controllo che specificano la dimensione del dato da leggere. Tutte queste informazioni viaggiano lungo il bus a una certa velocità, elevata ma pur sempre finita, e definita dalle caratteristiche fisiche ed elettriche

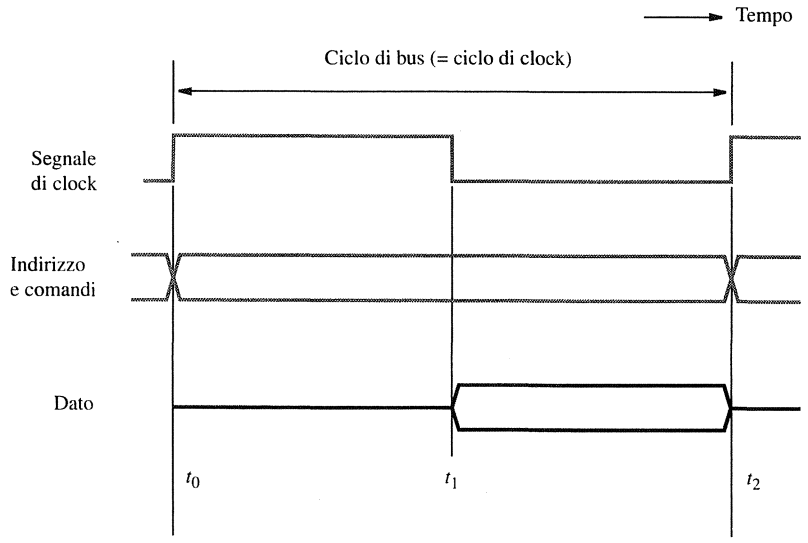


Figura 7.3 Andamento temporale dei segnali di bus sincrono nel ciclo di lettura.

del bus.[1] La durata $t_1 - t_2$ del semiciclo $[t_0, t_1]$ non deve essere inferiore al massimo ritardo di propagazione tra due unità collegate al bus, e deve anche lasciare alle unità il tempo bastante per riconoscere (decodificare) l'indirizzo e interpretare i comandi, cosicché l'unità slave obiettivo dell'operazione sia pronta a rispondere entro l'istante di tempo t_1; è perciò essenziale che le unità slave non agiscano prima dell'istante t_1. Durante l'intervallo di tempo $[t_0, t_1]$ il contenuto del bus dei dati è privo di significato perché indirizzo e comandi sono ancora in viaggio. Subito dopo l'istante t_1 l'unità slave può emettere sulle linee di dato la stringa di bit che rappresenta il dato.

Alla fine del ciclo di clock, vale a dire nell'istante di tempo t_2, l'unità master iniziatrice *campiona* (*strobe*) il valore presente sulle linee di dato e lo scrive dove deve, generalmente in un suo registro interno. Qui il termine campionare significa rilevare lo stato delle linee di dato all'istante t_2, ottenendo pertanto una stringa di bit, e scrivere tale stringa in un elemento di memoria capace di contenerla, di solito un registro. Il valore si deve propagare lungo le linee di dato fino a raggiungere l'unità master, il che richiede un certo *tempo di propagazione* (*propagation time*). Se il dato va messo in un registro è necessario che rimanga stabile agli ingressi del registro per un intervallo di tempo di durata non inferiore al *tempo di setup* o *di impostazione* dei bistabili (*setup time*), parametro caratteristico della tecnologia microelettronica in uso. Dunque la durata $t_2 - t_1$ dell'intervallo di tempo $[t_1, t_2]$ non deve essere inferiore alla somma del *massimo tempo di propagazione* dei bit del dato lungo il bus con il *tempo di setup* del registro.

L'operazione di scrittura procede in modo molto simile a quella di lettura. L'unità master, iniziatrice dell'operazione, emette indirizzo, dato e comando di scrittura (cioè pone $R/\overline{W} = 0$) sulle linee di indirizzo, di dato e di controllo del bus, rispettivamente, nello stesso istante di tempo all'inizio del ciclo, cioè all'istante t_0. Alla fine del ciclo, cioè all'istante t_2, l'unità slave obiettivo dell'operazione *campiona* le linee di dato, ottiene il valore e lo scrive dove deve, di solito in un suo registro interno. Il lettore può esercitarsi in modo autonomo a tracciare il diagramma temporale di scrittura.

La Figura 7.3 rappresenta l'andamento temporale dei segnali di bus in forma ideale, ma l'andamento reale è un po' diverso a causa dei ritardi inevitabili cui sono soggetti i vari eventi di bus, dovuti alla velocità finita dei segnali e ai tempi di azione e reazione delle unità coinvolte. La Figura 7.4 mostra l'andamento del ciclo di lettura in modo più realistico. Essa riporta due volte ciascun segnale, tranne il segnale di clock che è unico: sopra, i segnali come sono visti dall'unità master, sotto, come sono visti dall'unità slave. Siccome la velocità dei segnali è finita, le due unità vedono lo stesso segnale con un certo sfasamento temporale. Sola eccezione è data dal segnale di clock, i cui fronti di transizione si suppone vengano davvero visti simultaneamente da tutte le unità coinvolte; del resto se così non fosse il presupposto di bus sincrono cadrebbe. Spesso il progettista di calcolatori si deve affaticare non poco proprio per garantire che il segnale di clock sia effettivamente sincrono per tutte le unità funzionali del calcolatore, comportamento mai facile da ottenere.

L'unità master iniziatrice emette sul bus indirizzo e comandi in corrispondenza del fronte di salita delimitante l'inizio del ciclo di clock, cioè all'istante t_0. Tali segnali escono sul bus all'istante di tempo t_{IP} (tempo di Indirizzo unità Principale) perché l'unità master ritarda per produrli e imporli sul bus. Essi raggiungono l'unità slave obiettivo all'istante di tempo successivo t_{IS} (tempo di Indirizzo unità Secondaria) perché ritardano nel viaggio lungo il bus. L'unità slave riconosce (decodifica) l'indirizzo e interpreta i comandi all'istante di tempo t_1, ed emette sul bus il dato richiesto. Di nuovo, il dato esce sul bus all'istante di tempo successivo t_{DS} (tempo di Dato unità Secondaria) perché l'unità slave ritarda per imporlo sul bus. Ancora,

[1] Comunque la velocità non è superiore a quella della luce nel vuoto, che in 1 ns percorre 30 cm.

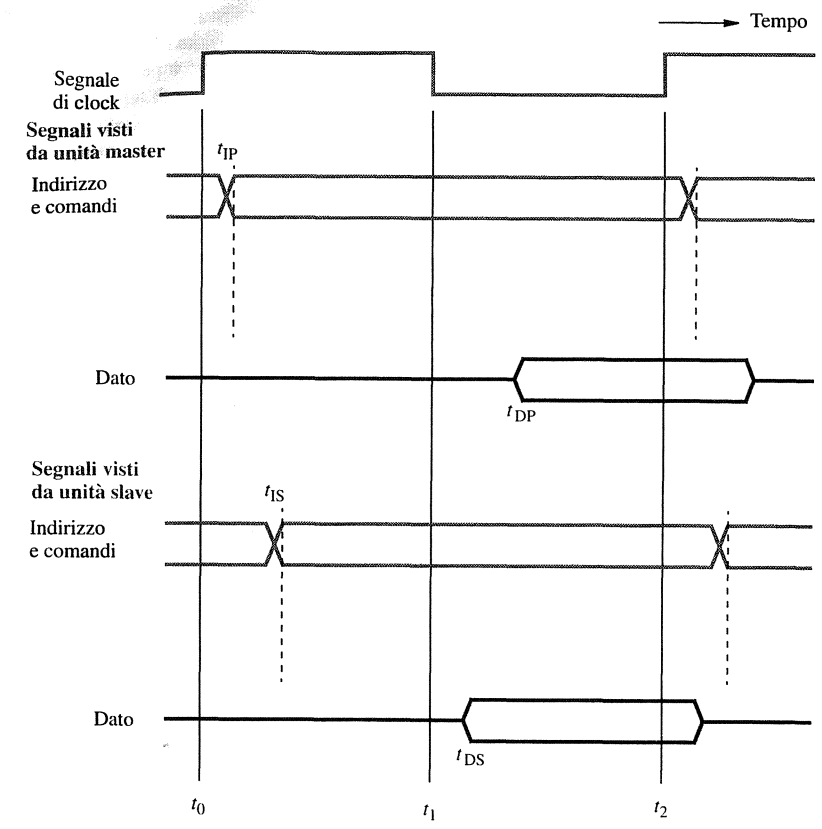


Figura 7.4 Andamento dettagliato dei segnali di bus sincrono nel ciclo di lettura.

esso raggiunge l'unità master all'istante di tempo successivo t_{DP} (tempo di Dato unità Principale) perché ritarda nel viaggio lungo il bus. L'unità master campiona il dato all'istante di tempo t_2 e lo scrive dove deve, per esempio in un suo registro interno. La durata $t_2 - t_{DP}$ dell'intervallo di tempo $[t_{DP}, t_2]$ non deve essere inferiore al tempo di setup del registro. Inoltre il dato deve restare stabile sul bus ancora per un breve intervallo di tempo dopo l'istante t_2; tale intervallo non deve essere inferiore al *tempo di hold* o *di mantenimento* (*hold time*) del registro.

In genere nella letteratura scientifica dedicata al progetto digitale si trovano solo diagrammi temporali semplificati come quello in Figura 7.3, soprattutto quando l'autore intende illustrare i concetti omettendo i dettagli pratici. Tuttavia non bisogna mai dimenticare che i ritardi sono inevitabili.

Naturalmente il bus si può trovare anche in stato di riposo, quando non ci sono in corso né letture né scritture (di I/O o di memoria). Ciò si vede sul diagramma temporale perché l'indirizzo non è significativo, ovvero non corrisponde a nessuna unità slave (e neppure alla memoria). Svariati protocolli di bus prevedono una linea di controllo apposita (compresa nel

bus di controllo) per indicare se l'indirizzo correntemente presente sul bus sia valido e dunque un'operazione di bus sia in corso, oppure se sia da ignorare e dunque il bus sia in riposo. Tale segnale viene attivato dall'unità master all'inizio dell'operazione e disattivato non appena è terminata.

Trasferimento multiciclo

Lo schema di operazione di bus tratteggiato sopra è semplice e pertanto dà luogo a interfacce di bus circuitalmente semplici, ma presenta limitazioni. Siccome tutta l'operazione si svolge in un solo ciclo di clock, il periodo di clock $t_2 - t_0$ non deve essere inferiore alla somma dei ritardi derivanti dal massimo tempo di propagazione e dall'unità più lenta ad agire o reagire. Ciò costringe tutte le unità a lavorare alla stessa velocità di quella più lenta.

Inoltre il processore, o in generale l'unità master, non ha modo di stabilire se l'unità slave abbia effettivamente risposto al comando. Il processore si limita a supporre che all'istante di tempo t_2, cioè alla fine del ciclo di clock: in caso di scrittura, che il dato abbia raggiunto l'unità slave e che essa lo abbia campionato; in caso di lettura, che l'unità slave gli abbia mandato il dato. Ma se a seguito di malfunzionamento l'unità slave non si fosse comportata come avrebbe dovuto l'errore non verrebbe rilevato.

Onde risolvere il problema la maggior parte dei bus contiene uno o più segnali di controllo appositi che rappresentano la risposta dell'unità slave. Essi informano l'unità master che l'unità slave ha ottenuto e riconosciuto l'indirizzo, ha interpretato il comando ed è dunque pronta a partecipare al trasferimento di dato. Inoltre essi rendono possibile aggiustare la durata effettiva del trasferimento per venire incontro a esigenze particolari delle unità coinvolte; di solito si tratta di mettere d'accordo unità master veloce con unità slave alquanto più lenta (di rado si dà la situazione opposta). La soluzione al problema consiste nell'usare un segnale di clock a frequenza elevata, cosicché l'operazione di bus si possa estendere su due o più cicli di clock; unità slave lente consumano più cicli di quelle veloci.

La Figura 7.5 mostra un esempio di lettura multiciclo. Durante il ciclo di clock numero 1 l'unità master iniziatrice emette sul bus indirizzo e comandi per l'operazione di lettura. L'unità slave obiettivo li riceve, riconosce (decodifica) e interpreta, consumando il resto del ciclo numero 1. Sul fronte di salita successivo del segnale di clock, cioè all'inizio del ciclo numero 2, essa, avendo compreso che cosa le si stia chiedendo, prende la decisione di rispondere al comando impartitole e comincia a recuperare il dato da mandare (presumibilmente si trova già in un suo registro interno). Come si è detto prima tale operazione di recupero non può essere istantanea, dunque l'unità slave non è in grado di rispondere immediatamente. Il dato è effettivamente pronto e l'unità slave lo emette sul bus all'inizio del ciclo numero 3, e nel medesimo istante di tempo essa attiva un segnale di controllo apposito (che compare qui per la prima volta) per rendere noto di averlo emesso; in Figura 7.5 esso è chiamato genericamente "Slave-pronto" ed è supposto attivo alto. L'unità master, che stava appunto aspettando di vedere l'attivazione di tale segnale, campiona il dato e lo scrive dove deve (presumibilmente in un suo registro interno) alla fine del ciclo durante cui ha visto l'attivazione del segnale, vale a dire alla fine del ciclo numero 3. Ora l'operazione di lettura è terminata, avendo impiegato 3 cicli in totale, e nel ciclo numero 4 l'unità master potrebbe, se lo volesse, iniziarne un'altra, di lettura o scrittura.

Il segnale Slave-pronto rappresenta una sorta di *riscontro* (*acknowledgement*) mandato all'unità master per informarla che sulle linee di dato del bus è stato mandato un dato valido, il quale ormai è in viaggio. Nell'esempio di Figura 7.5 l'unità slave attiva il segnale di riscontro nel ciclo numero 3, ma un'unità differente avrebbe potuto essere più veloce o lenta e dunque avrebbe potuto riscontrare prima o dopo il ciclo numero 3. Pertanto il segnale di riscontro permette di adattare la durata effettiva dell'operazione di trasferimento secondo le esigenze delle unità coinvolte. Se l'unità slave non riscontrasse mai (per esempio se fosse guasta oppure se l'indirizzo mandato sul bus non corrispondesse a nessuna unità), l'unità

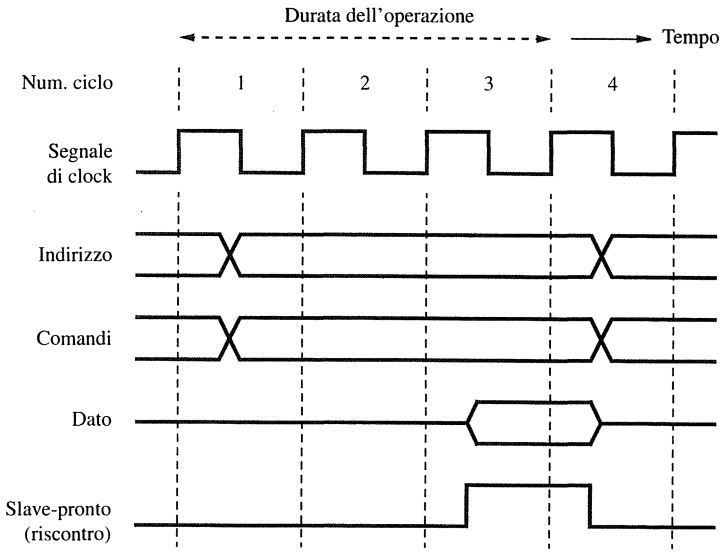


Figura 7.5 Andamento temporale dell'operazione di lettura multiciclo.

master dovrebbe attendere per un numero prestabilito di cicli, abbastanza elevato da soddisfare anche l'unità slave più lenta, e poi, se il riscontro ancora non fosse arrivato, mettere fine all'operazione interrompendola.

Il segnale di riscontro ha una funzione positiva: il processore attende finché l'unità slave non ha confermato di avere completato l'operazione. Taluni protocolli di bus prevedono invece un meccanismo di tipo negativo: di norma il processore non attende, salvo che l'unità slave non gli mandi in tempo utile una richiesta di proroga attivando una linea di *attesa* (*wait*); fintantoché tale linea resta attivata il processore attende, e non appena viene disattivata conclude l'operazione.

Di seguito si mostra il bus asincrono, dove il segnale di clock manca del tutto.

7.2.2 Bus asincrono

Si può organizzare il trasferimento di dato tramite bus facendo in modo che le unità master e slave si accordino intorno a che cosa fare e a quando farlo. Il concetto di bus asincrono generalizza l'idea di ricorrere al segnale di riscontro, che già si è vista funzionare bene nel bus sincrono in Figura 7.5. Nel bus asincrono non c'è alcun segnale di clock; al posto di questo si trovano invece due linee di controllo con funzione di segnalazione e riscontro, rispettivamente. La prima linea è pilotata dall'unità master per segnalare di essere pronta all'operazione e in seguito di averla espletata, la seconda è pilotata dall'unità slave che la usa per riscontrare all'unità master. In sostanza le due linee hanno funzione di sincronismo; si suppone che entrambe siano attive alte.

In linea di principio il trasferimento di dato tramite bus asincrono, secondo la procedura di *handshake* (letteralmente "stretta di mano") o se si preferisce *di mutuo accordo*, si sviluppa nel modo seguente. L'unità master iniziatrice manda indirizzo e comandi sul bus, poi segnala a tutte le altre unità di averlo fatto attivando la linea di controllo Master-pronto. Ciò stimola

tutte le unità a decodificare l'indirizzo e interpretare il comando. L'unità slave obiettivo reagisce mandando il dato sul bus in caso di lettura, oppure campionando il dato presente sul bus in caso di scrittura, poi riscontra all'unità master di averlo fatto attivando la linea di controllo Slave-pronto. L'unità master, che stava appunto aspettando tale segnale di riscontro, poco dopo averne visto l'attivazione disattiva la linea Master-pronto ovvero segnala di avere dal canto suo espletato l'operazione, e insieme toglie dal bus indirizzo e comandi; inoltre, se l'operazione è di lettura essa campiona il dato presente sul bus. Infine l'unità slave disattiva la linea Slave-pronto, ovvero riscontra nuovamente all'unità master; con ciò si conclude l'operazione di trasferimento.

La Figura 7.6 mostra l'andamento dei segnali di bus per l'operazione di lettura. La successione di eventi si sviluppa nel modo seguente.

- Al tempo t_0, l'unità master manda sul bus indirizzo e comandi, che iniziano a viaggiare verso le unità slave.
- Al tempo t_1, l'unità master attiva la linea Master-pronto per segnalare alle unità slave che indirizzo e comandi sono presenti sul bus. L'intervallo di tempo $[t_0, t_1]$ compensa l'eventuale *sfasamento temporale* (*skew*) tra le unità slave, dovuto al fatto di trovarsi a distanze diverse da quella master e al fatto che i segnali si propagano lungo il bus con ritardi variabili in funzione della distanza. Pertanto la durata dell'intervallo, $t_1 - t_0$, non deve essere inferiore al massimo sfasamento possibile tra due unità slave. Anche nel bus sincrono si verifica sfasamento, ma se ne tiene contro calcolando il ritardo massimo di propagazione. Quando indirizzo e comandi arrivano alle unità slave, esse li esaminano. Naturalmente bisogna dare ai circuiti di interfaccia delle unità tempo sufficiente per fare l'esame, e anche tale intervallo di tempo va conteggiato in $t_1 - t_0$.
- Al tempo t_2, l'unità slave obiettivo, avendo completato l'esame e dunque compreso di essere oggetto di un'operazione e che cosa precisamente le si stia chiedendo di fare, ese-

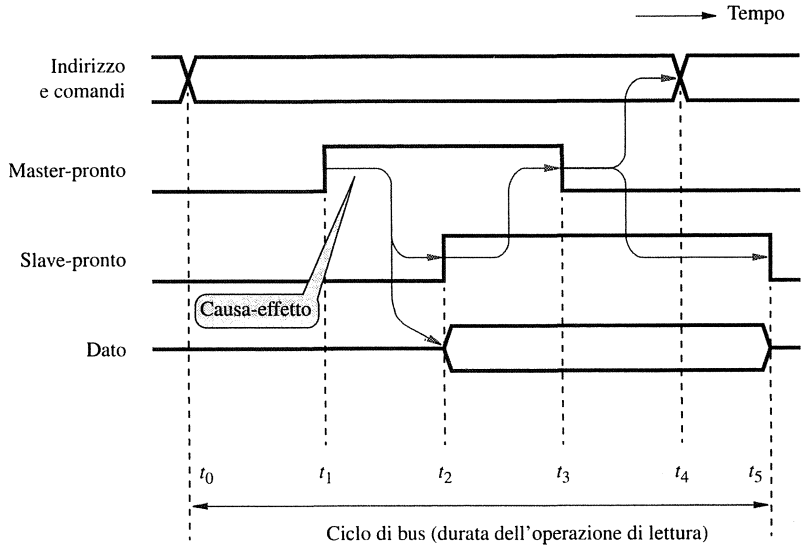


Figura 7.6 Andamento temporale dell'operazione di lettura tramite bus asincrono.

gue l'operazione di lettura mandando il dato sul bus e nello stesso istante di tempo attiva la linea Slave-pronto, riscontrando così all'unità master. Presumibilmente il dato proviene da un registro interno all'unità e dunque le occorre un po' di tempo per imporlo sul bus; anche tale intervallo di tempo va conteggiato nel calcolare l'istante di tempo t_2. Insomma la durata $t_2 - t_1$ dell'intervallo di tempo $[t_1, t_2]$ va definita in modo da compensare il ritardo cui sono soggette le unità slave nell'emettere sul bus il dato e il ritardo di propagazione del dato fino a raggiungere l'unità master. Si noti come unità diverse possano presentare ritardi diversi e dunque gli intervalli $[t_0, t_1]$ e $[t_1, t_2]$ non sono fissati, ma variano secondo le unità coinvolte.

- Al tempo t_3, il segnale di Slave-pronto raggiunge l'unità master e così rende noto che il dato è presente sul bus. L'unità master deve aspettare ancora un po' per compensare l'eventuale sfasamento e, come se non bastasse, deve anche aggiungere al tempo di attesa l'intervallo di tempo di setup del suo registro interno, dove presumibilmente andrà a scrivere il dato. Dunque, dopo avere atteso per un intervallo di tempo non inferiore alla somma dei massimi di tutti i ritardi appena elencati, essa può finalmente campionare il dato presente sul bus e scriverlo dove deve. È questo l'istante di tempo per disattivare la linea Master-pronto, segnalando così all'unità slave di avere ottenuto il dato.
- Al tempo t_4, l'unità master toglie indirizzo e comandi dal bus. Di nuovo l'intervallo di tempo $[t_3, t_4]$ è necessario per compensare gli sfasamenti temporali tra segnali in viaggio lungo il bus. Infatti, se succedesse che un'unità slave vedesse mutare l'indirizzo sul bus mentre il segnale Master-pronto fosse ancora attivo, e se per caso l'indirizzo nuovo fosse uno di quelli attribuitile, potrebbe ritenere erroneamente di essere obiettivo di una nuova operazione di trasferimento che stesse appena cominciando.
- Al tempo t_5, non appena l'unità slave obiettivo ha rilevato la disattivazione del segnale di Master-pronto, essa toglie il dato dal bus e disattiva la linea Slave-pronto, riscontrando così nuovamente all'unità master. Con ciò l'operazione di bus si conclude.

Nell'operazione di scrittura l'andamento temporale dei segnali è essenzialmente lo stesso della lettura, come mostra la Figura 7.7. La sola differenza rilevante è che l'unità master iniziatrice manda sul bus il dato insieme a indirizzo e comandi, ovvero al principio dell'operazione. L'unità slave obiettivo campiona il dato presente sul bus e lo scrive dove deve (presumibilmente in un suo registro interno) non appena ha rilevato l'attivazione del segnale di Master-pronto, e riscontra di averlo fatto attivando la linea Slave-pronto. Il resto dell'operazione si sviluppa come in lettura.

La procedura di handshake illustrata nelle Figure 7.6 e 7.7 è di tipo *completamente interallacciato* (*fully interlocked*). Ogniqualvolta uno dei due segnali di sincronismo varia, anche l'altro varia in risposta. Siffatto protocollo di bus garantisce affidabilità e flessibilità massime. Nelle due figure sono messe in evidenza, tramite frecce grige, le relazioni causa-effetto che legano i vari eventi di bus. Nel protocollo di bus asincrono tali relazioni sono essenziali giacché, mancando il segnale di clock, ogni evento si verifica solo come effetto di un evento precedente, che ne è la causa. L'intera operazione dura un ciclo di bus (non di clock) e non avrebbe alcun senso tentare di suddividerlo in cicli di clock giacché il segnale di clock manca del tutto.

Discussione sul bus

Nei calcolatori commerciali si trovano numerose varianti ai protocolli di bus schematici illustrati sopra. Per scegliere quale sia meglio usare bisogna soppesare i fattori seguenti.

- Semplicità del circuito di interfaccia al bus.
- Flessibilità nell'aggiungere nuove unità dotate di interfacce funzionanti a velocità differenti.

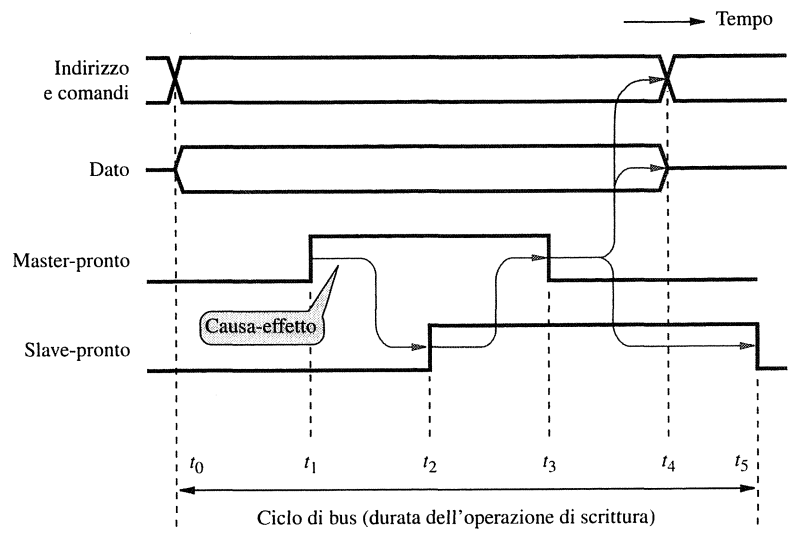


Figura 7.7 Andamento temporale dell'operazione di scrittura tramite bus asincrono.

- Tempo totale necessario per trasferire il dato.
- Capacità di rilevare errori conseguenti al tentativo di indirizzare unità slave obiettivo inesistenti o errori derivanti da malfunzionamento dell'unità.

Il vantaggio principale offerto dal protocollo di bus asincrono è che la procedura di handshake fa a meno del segnale di clock, e così semplifica il progetto delle unità (o meglio delle rispettive interfacce al bus). Con bus asincrono i ritardi, non importa se conseguenti agli sfasamenti temporali o derivanti dai circuiti di interfaccia, vanno automaticamente a posto. Se i ritardi mutano, per esempio perché il carico elettrico del bus varia quando si collega o scollega un'interfaccia, lo schema di sincronizzazione tra unità si adatta spontaneamente alla nuova situazione. Invece con bus sincrono bisogna progettare il circuito di distribuzione del segnale di clock in modo molto accurato per garantire che la scansione del tempo sia la stessa per tutte le unità, senza sfasamento, e i ritardi vanno tenuti sotto controllo affinché non eccedano limiti molto stretti.

Tuttavia con bus asincrono e procedura di handshake la frequenza di trasferimento è limitata dal fatto che per ogni dato trasferito i fronti dei segnali di sincronismo percorrono il bus avanti e indietro per due volte. Dunque ci sono quattro ritardi che si assommano: andata-ritorno-andata-ritorno. Ogni ritardo è causato dal tempo che i fronti dei segnali impiegano per viaggiare da un'unità all'altra.[2] Ciò si vede subito nelle Figure 7.6 e 7.7, giacché il segnale di Slave-pronto transita in risposta al segnale di Master-pronto, e viceversa. Invece con bus

[2] In inglese il tempo di viaggio di sola andata si chiama *trip time*, e quello di andata e ritorno *round trip time*; pertanto con bus asincrono si cumulano due *round trip time*.

sincrono basta tenere conto di due soli ritardi di viaggio lungo il bus, in avanti dall'unità iniziatrice a quella obiettivo (indirizzo, comandi e, se scrittura, dato) e poi indietro (riscontro e, se lettura, dato).[3]

Pertanto con bus sincrono si riescono a raggiungere frequenze di trasferimento più elevate; inoltre per collegare al bus unità lente basta prolungare l'operazione di trasferimento aggiungendole qualche ciclo di clock. Per tali ragioni la maggior parte dei bus di calcolatori attuali è di tipo sincrono.

7.2.3 Considerazioni elettriche

Un bus è un mezzo di interconnessione a cui possono essere collegati vari dispositivi. È essenziale assicurarsi che solo un dispositivo possa inviare dati sul bus in qualsiasi momento dato. Una porta logica che invia dati sul bus è chiamata *pilota del bus* (*bus driver*). Tutti i dispositivi connessi al bus, eccetto quello che al momento sta inviando dati, devono avere i loro piloti del bus disabilitati. Si usa a tal uopo un tipo speciale di porta logica, detta *porta a tre stati* (*tri-state gate*). Questa ha un ingresso di controllo che viene usato per abilitare o disabilitare la porta. Quando è abilitata essa pilota il bus con 1 o 0, in corrispondenza al valore del suo segnale di ingresso. Quando è disabilitata è in effetti disconnessa dal bus. Da un punto di vista elettrico, la sua uscita va in uno stato di alta impedenza che non influenza il segnale sul bus.

7.3 Arbitraggio

Ci sono occasioni in cui due o più entità si contendono l'uso di una singola risorsa in un calcolatore. Per esempio, due dispositivi possono aver bisogno di accedere a un certo slave allo stesso momento. In questi casi, è necessario decidere quale dispositivo accederà allo slave per primo. La decisione è di solito presa per mezzo di un processo di arbitraggio effettuato da un circuito *arbitro* (*arbiter*). Il processo di arbitraggio inizia quando ciascun dispositivo invia una *richiesta* di uso della risorsa condivisa. L'arbitro associa una priorità alle singole richieste. Se riceve due richieste allo stesso tempo, *concede* (*grant*) l'uso dello slave per primo al dispositivo che ha priorità più alta.

Per illustrare il processo di arbitraggio, si consideri il caso in cui un singolo bus sia la risorsa condivisa. Il dispositivo che fa partire le richieste di trasferimento dati è il master del bus. Nel Paragrafo 7.2, la discussione coinvolge solo un master del bus, il processore. È possibile che vari dispositivi in un calcolatore abbiano necessità di diventare master del bus per trasferire dati. Per esempio, un dispositivo di I/O necessita di essere master del bus per trasferire dati direttamente alla o dalla memoria del calcolatore. Poiché il bus è una singola risorsa condivisa, è essenziale regolarne l'accesso dai bus master.

Un dispositivo che desideri usare il bus manda una richiesta all'arbitro. Quando più richieste arrivano contemporaneamente, l'arbitro ne seleziona una e concede il bus al dispositivo corrispondente. Per alcuni dispositivi, un ritardo nell'accesso al bus può portare a un errore. A tali dispositivi va data alta priorità. Se non c'è particolare urgenza tra le richieste, l'arbitro può concedere il bus usando un semplice schema a rotazione (*round-robin*).

La Figura 7.8 illustra una disposizione per l'arbitraggio del bus che coinvolge due master. Ci sono due linee per la richiesta del bus, BR1 e BR2, e due linee per concedere il bus, BG1 e BG2, che collegano l'arbitro ai master.

[3] Dunque con bus sincrono il ritardo cumulativo ammonta a un solo *round trip time*.

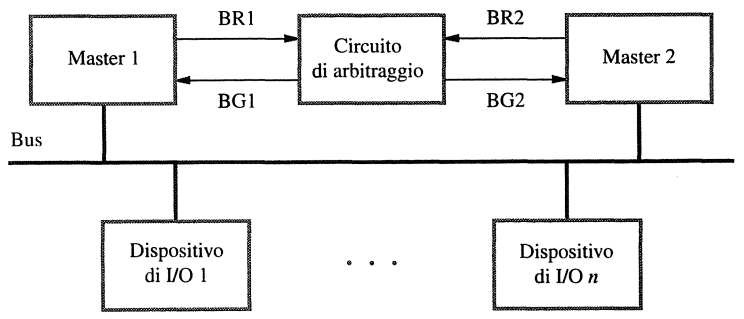


Figura 7.8 Arbitraggio del bus.

Un master chiede l'uso del bus attivando la sua linea di richiesta del bus. Se una sola linea di richiesta è attiva, l'arbitro attiva la corrispondente linea di concessione. Questo indica al master selezionato che adesso può usare il bus per trasferire dati. Quando il trasferimento è concluso, il master disattiva la sua richiesta del bus e l'arbitro disattiva la concessione del bus.

La Figura 7.9 mostra una possibile sequenza di eventi per il caso di tre master. Si assuma che il master 1 abbia la priorità più alta, seguito dagli altri in ordine numerico crescente. Il master 2 invia una richiesta d'uso del bus per primo. Poiché non ci sono altre richieste, l'arbitro concede il bus a questo master asserendo BG2. Quando il master 2 completa la sua operazione di trasferimento dati, rilascia il bus disattivando BR2. Per quel momento, entrambi i master 1 e 3 hanno attivato le loro linee di richiesta. Poiché il dispositivo 1 ha priorità più alta,

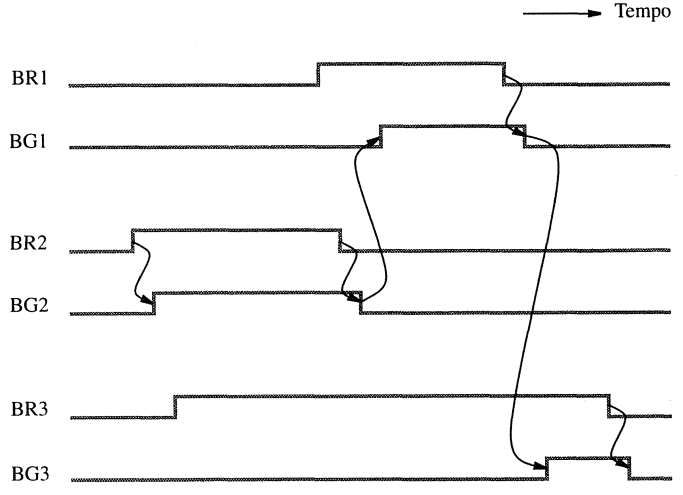


Figura 7.9 Concessione dell'uso del bus in base a delle priorità.

l'arbitro attiva BG1 dopo aver disattivato BG2, quindi concede il bus al master 1. Successivamente, quando il master 1 rilascia il bus disattivando BR1, l'arbitro disattiva BG1 e attiva BG3 per concedere il bus al master 3. Si noti che il bus è concesso al master 1 prima del master 3 sebbene il master 3 abbia attivato la sua linea di richiesta prima del master 1.

7.4 Interfacce di I/O

L'interfaccia di ingresso e uscita, o di I/O, è il circuito digitale che collega la periferica al bus del calcolatore. L'interfaccia di I/O ha due lati: quello rivolto verso il bus è collegato alle linee di indirizzo, di dato e di controllo; quello rivolto verso la periferica non ha struttura uniforme, bensì dipende dalla natura della periferica stessa, ma generalmente contiene le linee di dato e di controllo che servono a quest'ultima per scambiare dati con l'interfaccia. Il lato di periferica dell'interfaccia di I/O viene spesso chiamato *porta* (*port*), ma per abuso di linguaggio il termine porta è spesso inteso come sinonimo di interfaccia; anche qui si seguirà tale uso, del resto comunissimo. Benché le tipologie di porta siano poco uniformi se ne possono comunque distinguere due grandi famiglie: *porta parallela* (*parallel port*) e *porta seriale* (*serial port*); alcune di tali porte sono anche state standardizzate. Talvolta l'interfaccia di I/O è integrata nella periferica ed è inseparabile da essa, ma più spesso la periferica va collegata a una porta già esistente nel calcolatore, più o meno standard. Le porte standard più comuni sono spesso visibili sui pannelli del calcolatore, sotto forma di connettori di varia conformazione che rappresentano il lato di periferica della porta. Il lato di bus non è di norma visibile se non ispezionando l'interno del calcolatore.

Di solito dal lato di periferica la porta parallela presenta un connettore con un certo numero di morsetti, dove si inserisce il cavo di collegamento contenente altrettante linee; spesso si usa il *cavo piatto* (*flat cable*). La porta parallela scambia dati con la periferica a gruppi di bit, mentre la porta seriale scambia un solo bit per volta. Dal lato di bus porta parallela e seriale non differiscono; internamente però la porta seriale deve essere in grado di effettuare la conversione da formato seriale a parallelo e viceversa.

Prima di illustrare in dettaglio esempi di circuito di interfaccia è opportuno richiamarne le funzioni. Secondo quanto detto nel Paragrafo 3.1, un'interfaccia di I/O ha i compiti seguenti.

- Mettere a disposizione il buffer temporaneo per il dato.
- Disporre di un registro di stato, i cui bit siano leggibili da parte del processore per stabilire quale sia lo stato dell'interfaccia.
- Disporre di un registro di controllo che detiene le informazioni che governano il comportamento dell'interfaccia.
- Contenere il circuito riconoscitore (decodificatore) di indirizzo per stabilire se l'indirizzo corrente sul bus sia quello assegnatole (o uno di quelli assegnatile).
- Generare e rilevare i segnali di controllo coinvolti nello schema di sincronismo.
- Effettuare la conversione di formato necessaria per trasferire il dato dal lato di bus a quello di periferica o viceversa; per esempio nella porta seriale sia la conversione seriale-parallelo sia quella parallelo-seriale.

Nelle tecnologie più sofisticate, non di rado tali funzioni sono programmabili e variamente riconfigurabili.

7.4.1 Interfaccia parallela

Gli aspetti salienti del circuito di interfaccia di I/O verranno spiegati nel seguito mediante esempi. Inizialmente, si descrive un circuito d'interfaccia per una porta di ingresso a 8 bit

che può essere usata per collegare un semplice dispositivo d'ingresso, come una tastiera. Quindi, si descrive un circuito d'interfaccia per una porta di uscita a 8 bit, che può essere usata per collegare un dispositivo di uscita, quale uno schermo. Si fa l'ipotesi che l'interfaccia sia collegata a un processore a 32 bit (dunque il bus di indirizzo è da 32 bit), che lo spazio di indirizzamento di I/O sia unificato con quello di memoria e che il bus funzioni secondo il protocollo asincrono illustrato nelle Figure 7.6 e 7.7.

Interfaccia di ingresso

La Figura 7.10 mostra i componenti necessari per collegare la tastiera al calcolatore. I registri in questo circuito corrispondono a quelli di Figura 3.3. Si assuma che le interruzioni non siano usate, quindi non è necessario alcun registro di controllo. Ci sono solo due registri: un registro dati, KBD_DATA, e un registro di stato, KBD_STATUS. L'ultimo contiene la condizione di stato della tastiera, KIN.

Comunemente la tastiera consiste in una matrice di interruttori meccanici normalmente in stato aperto. Quando si preme un tasto l'interruttore corrispondente si chiude e così si crea un circuito dove passa un segnale elettrico (per esempio di tensione). Tale segnale viene rilevato da un circuito codificatore apposito che genera il codice ASCII corrispondente al carattere associato al tasto. L'inconveniente dell'interruttore meccanico è che il contatto non si stabilisce istantaneamente bensì dopo una serie di microrimbalzi molto brevi e ravvicinati (impercettibili al tatto ma ben visibili con l'oscilloscopio), che possono durare uno o due millisecondi in totale. È abbastanza perché il calcolatore possa equivocare circa il numero di pressioni del tasto. I microrimbalzi che precedono la stabilizzazione definitiva del segnale vanno dunque in qualche modo o eliminati o ignorati. Per eliminarli basta interporre un circuito antirimbalzo, che taglia variazioni di segnale troppo brevi per essere significative. In alternativa, per ignorarli si ricorre a una tecnica di tipo programmativo: la routine che effettua l'operazione di lettura del carattere da tastiera attende abbastanza a lungo da essere sicura che il transitorio di rimbalzo si sia esaurito. La Figura 7.10 si basa sulla prima soluzione e il circuito antirimbalzo è integrato nel codificatore ASCII.

Il codificatore emette i bit rappresentanti il codice ASCII del carattere introdotto da tastiera accompagnati dal segnale Valido (attivo alto), che indica appunto l'avvenuta pressione del tasto. Tali informazioni vengono mandate al circuito di interfaccia (la porta di ingresso) contenente il registro di dato KBD_DATA e il bit di stato KIN. Quando si preme un tasto, il

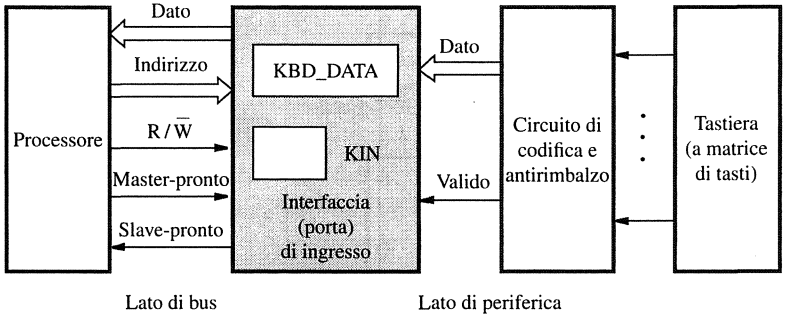


Figura 7.10 Schema a blocchi di porta di ingresso collegata a tastiera.

segnale Valido si attiva e con ciò introduce il codice ASCII del carattere nel registro KBD_DATA e porta a 1 il bit di stato KIN. Il circuito di interfaccia è collegato al bus asincrono, in cui il trasferimento di dato è controllato dai due segnali di unità master e slave pronte, rispettivamente, come mostrato in Figura 7.6; la linea di controllo R/$\overline{W}$ discrimina tra operazione di lettura (quella che interessa qui) e scrittura.

La Figura 7.11 mostra un possibile schema circuitale della porta di ingresso. Ci sono due locazioni indirizzabili in questa interfaccia, KBD_DATA e KBD_STATUS. Esse occupano locazioni di memoria adiacenti nello spazio degli indirizzi, come in Figura 3.3. Solo un bit, b_1, nel registro di stato contiene in realtà informazione utile. Questo è l'indicatore di stato

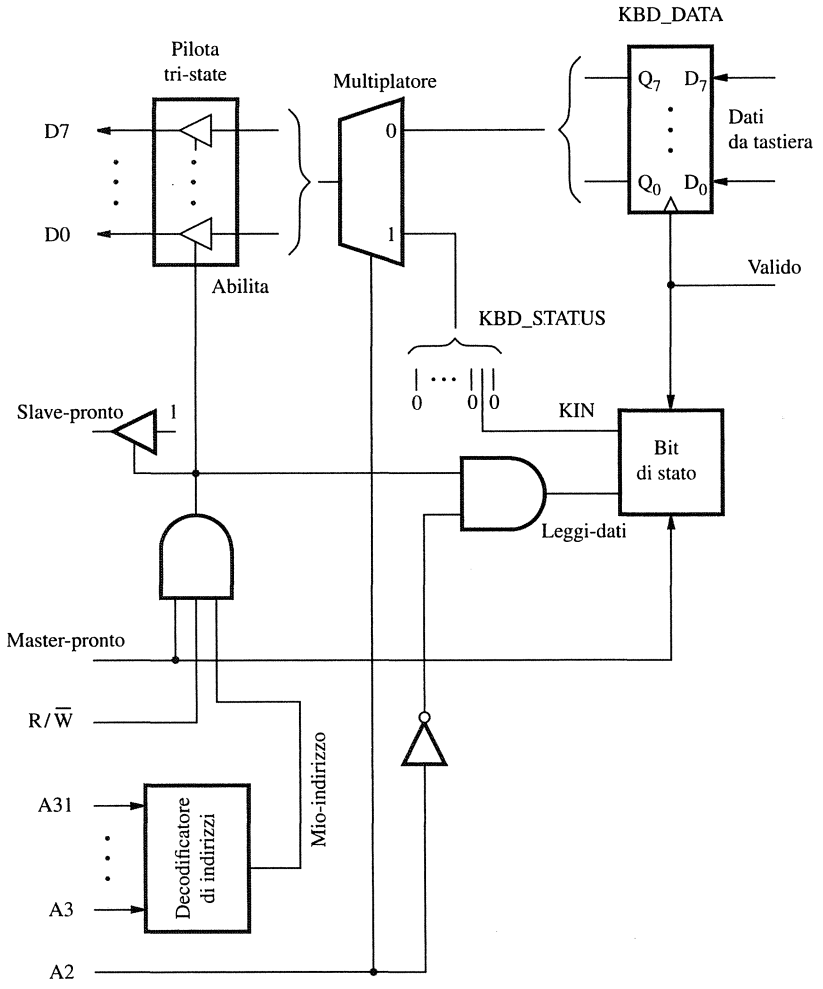


Figura 7.11 Un circuito di interfaccia d'ingresso.

della tastiera, KIN. Quando il registro di stato viene letto dal processore, le locazioni di tutti gli altri bit appaiono contenere zeri.

Quando il processore richiede un'operazione Read, pone l'indirizzo del registro appropriato sulle linee di indirizzo del bus. Il riconoscitore di indirizzo nel circuito d'interfaccia esamina i bit A_{31-3}, e asserisce la sua uscita Mio-indirizzo quando viene indirizzato uno dei due registri KBD_DATA o KBD_STATUS. Il bit A_2 determina quale dei due registri è coinvolto. Quindi si usa un multiplatore per selezionare il registro da collegare al bus in base al bit di indirizzo A_2. I due bit di indirizzo meno significativi, A_1 e A_0, non sono usati, poiché si è assunto che tutti gli indirizzi siano allineati alla parola.

L'uscita del multiplatore è collegata alle linee del bus dei dati attraverso una serie di porte a tre stati. Il circuito d'interfaccia abilita la porta a tre stati solo quando i tre segnali Master-pronto, Mio-indirizzo e R/$\overline{W}$ sono tutti uguali a 1, indicando un'operazione Read.

Il segnale Slave-pronto è asserito allo stesso momento, per informare il processore che il dato o l'informazione di stato richiesti sono stati posti sulle linee dei dati. Quando il bit di indirizzo A_2 è pari a 0, viene pure asserito Leggi-dati. Si usa questo segnale per azzerare l'indicatore di stato KIN.

Una possibile realizzazione del circuito per l'indicatore di stato è presentata in Figura 7.12. L'indicatore di stato KIN è l'uscita di un bistabile NOR collegato come mostrato. Un flip-flop è impostato a 1 dal fronte di salita sulla linea del segnale Valido. Questo evento cambia lo stato del bistabile NOR per impostare KIN a 1, ma solo quando Master-pronto è basso. La ragione per questa condizione aggiuntiva è assicurare che KIN non cambi stato mentre viene letto dal processore. Sia il flip-flop sia il bistabile vengono azzerati quando Leggi-dati diventa uguale a 1, che indica la lettura di KBD_DATA.

I circuiti mostrati nelle Figure 7.11 e 7.12 sono intesi illustrare le varie funzioni che un circuito di interfaccia deve effettuare. Un progettista che usi i moderni strumenti di progettazione assistita dal calcolatore specificherebbe tali funzioni usando un linguaggio di descrizione dell'hardware come VHDL o Verilog. I circuiti risultanti dipenderebbero dalla tecnologia usata e possono o meno essere gli stessi circuiti mostrati in queste figure.

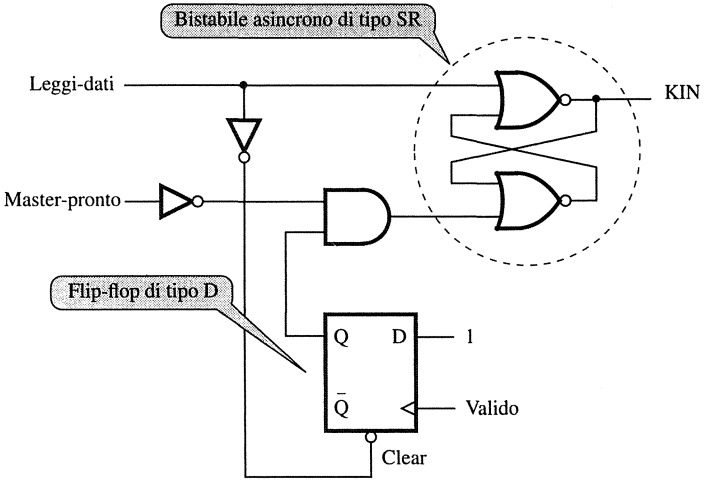


Figura 7.12 Schema circuitale del bit di stato della porta di ingresso.

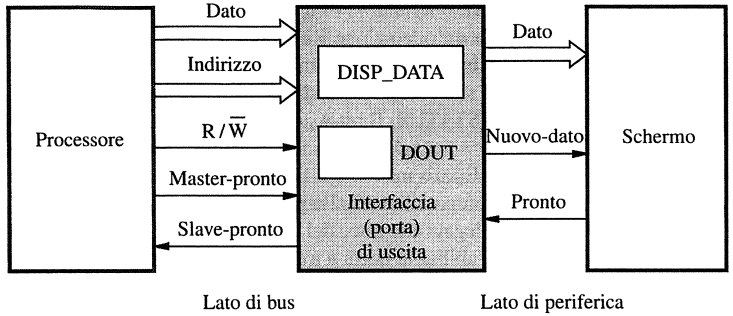


Figura 7.13 Schema a blocchi di porta di uscita collegata a uno schermo.

Interfaccia di uscita

Si consideri adesso l'interfaccia di uscita mostrata in Figura 7.13, che può essere usata per collegare un dispositivo di uscita come uno schermo.

Si è assunto che lo schermo usi due segnali di handshake, Nuovo-dato e Pronto, in modo simile all'handshake tra i segnali del bus Master-pronto e Slave-pronto. Quando lo schermo è pronto ad accettare un carattere, viene asserito il segnale Pronto; questo fa sì che il bit di stato DOUT nel registro DISP_STATUS sia posto a 1. Quando la routine di I/O valuta DOUT e trova che è uguale a 1, invia un carattere a DISP_DATA. Questo azzera il bit di stato DOUT e pone il segnale Nuovo-dato a 1. In risposta, lo schermo riazzera Pronto e mostra il carattere presente in DISP_DATA. Quando è pronto a ricevere un altro carattere, asserisce nuovamente Pronto, e il ciclo si ripete.

La Figura 7.14 mostra una realizzazione di questa interfaccia. Il suo funzionamento è simile a quello dell'interfaccia d'ingresso di Figura 7.11, eccetto che risponde a entrambe le operazioni Read e Write. Un'operazione Write in cui $A_2 = 0$ carica un byte di dati nel registro DISP_DATA. Un'operazione Read in cui $A_2 = 1$ legge il contenuto del registro di stato DISP_STATUS. In questo caso solo il bit di condizione DOUT, che è il bit b_2 del registro di stato, viene inviato dall'interfaccia. I rimanenti bit di DISP_STATUS non vengono usati. Il valore del bit di stato viene determinato dal circuito di controllo dell'handshake.

Un diagramma di stato che descrive il comportamento di questo circuito è fornito nel Problema risolto 7.4 nell'Eserciziario.

7.4.2 Interfaccia seriale

La porta seriale serve per collegare al calcolatore la periferica che deve scambiare dati in formato seriale, cioè un bit alla volta, e pertanto deve essere capace di trasferire il dato in formato parallelo dal lato di bus, in formato seriale dal lato di periferica. La conversione seriale-parallelo del dato viene svolta tramite *registri a scorrimento* (*shift register*), dotati anche di funzione di scrittura o lettura in parallelo.

La Figura 7.15 mostra lo schema a componenti della porta seriale, contenente i registri DATO_ING e DATO_USC, analoghi ai registri dei dati già visti nella porta parallela, e due registri a scorrimento, uno di ingresso e uno di uscita, che sono una novità. Il registro a scorrimento di ingresso acquisisce la sequenza di bit in arrivo dalla periferica e quando 8 bit sono entrati vengono copiati in parallelo nel registro DATO_ING. Similmente gli 8 bit contenuti nel registro DATO_USC vengono emessi in parallelo e copiati nel registro a scorrimento di

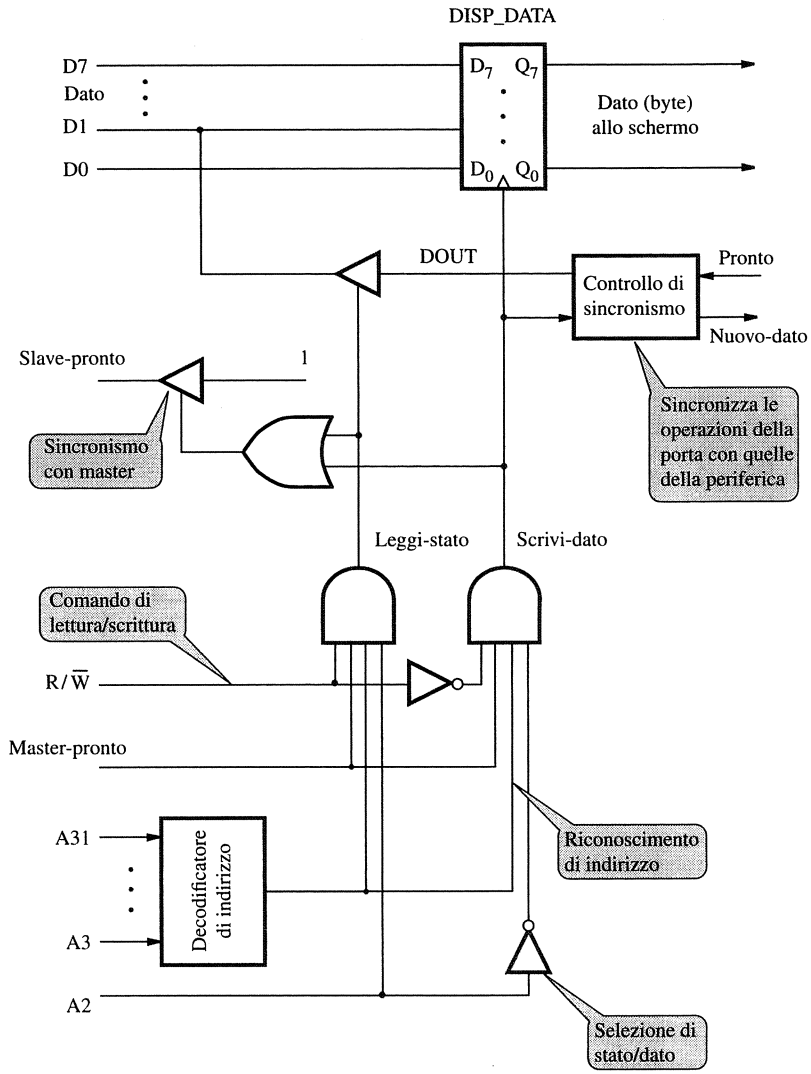


Figura 7.14 Schema circuitale di porta parallela di uscita.

uscita (tramite la funzione di scrittura in parallelo); i bit vengono poi emessi serialmente dal registro e mandati alla periferica.

Il lato di bus della porta seriale è identico a quello della porta parallela e anche i bit di stato S_ING e S_USC vengono mantenuti dal blocco Stato e controllo, e hanno la stessa funzione. Se il bit S_ING vale 1, c'è un dato nuovo in ingresso nel registro DATO_ING; il bit viene portato automaticamente a 0 non appena il processore ha effettuato l'operazione di lettura del dato, cioè lo ha letto dalla porta. Se il bit S_USC vale 1, il registro DATO_USC è

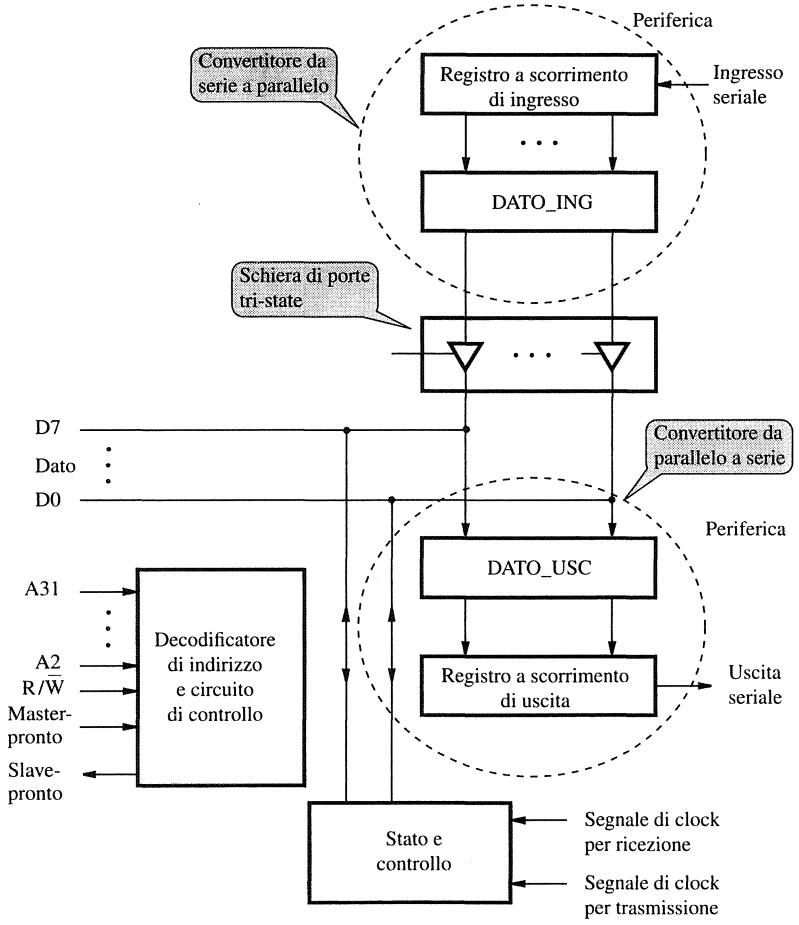


Figura 7.15 Schema a blocchi di porta seriale di ingresso e uscita.

libero e pronto per accogliere un dato in uscita; il bit viene portato a 0 non appena il processore ha effettuato l'operazione di scrittura, cioè scrive il dato nella porta, e riportato a 1 non appena il dato è stato trasferito nel registro a scorrimento di uscita.

Il fatto che anche i registri a scorrimento siano leggibili o scrivibili in parallelo non è senza ragione. Sarebbe infatti stato possibile togliere i due registri a scorrimento indipendenti e attribuire la rispettiva funzione ai due registri DATO_ING e DATO_USC, in tal modo semplificando lo schema di Figura 7.15. Ma così facendo si sarebbe alquanto irrigidito il funzionamento della porta imponendole vincoli molto stretti: dopo avere acquisito 8 bit dalla periferica essa non ne avrebbe potuti accettare di nuovi finché il processore non avesse completata l'operazione, leggendoli dal registro DATO_ING. Pertanto sarebbe stato necessario separare l'introduzione di due dati consecutivi inserendo una pausa onde dare al processore tempo per intervenire. Invece, prevedendo un registro a scorrimento indipendente, la periferica può iniziare a trasmettere il dato nuovo non appena quello vecchio è stato trasferito nel registro DATO_ING. Se dunque il processore è abbastanza veloce da leggere il contenuto di DATO_ING prima che la periferica abbia finito di emettere il nuovo dato, la porta seriale può ricevere un flusso continuo seriale di dati. La coppia formata dal registro a scorrimento in ingresso e dal registro parallelo DATO_ING costituisce un convertitore seriale-parallelo. Per la funzione di uscita valgono le medesime considerazioni.

Durante la trasmissione seriale, il ricevente ha bisogno di conoscere quando far scorrere ciascun bit nel suo registro di ingresso a scorrimento. Poiché non c'è una linea separata per portare un segnale di clock dal trasmittente al ricevente, l'informazione di sincronizzazione necessaria deve essere inclusa nei dati trasmessi usando uno schema di codifica. Ci sono due approcci di base. Il primo è noto come trasmissione asincrona, poiché il ricevente usa un clock che non è sincronizzato con il clock del trasmittente. Nel secondo approccio, il ricevente è in grado di generare un clock che è sincronizzato con il clock del trasmittente. Questa viene detta trasmissione sincrona. Questi approcci sono brevemente descritti nel seguito.

Trasmissione asincrona

Questo approccio usa una tecnica chiamata trasmissione *Start-Stop*. I dati sono organizzati in piccoli gruppi da 6 a 8 bit, con inizio e fine ben definiti. In un'organizzazione tipica, i caratteri alfanumerici codificati in 8 bit vengono trasmessi come mostrato in Figura 7.16. La linea che collega il trasmittente e il ricevente è nello stato 1 quando inattiva. Un carattere viene trasmesso come una sequenza che inizia con il bit 0, chiamato bit Start, seguito da 8 bit di dati e 1 o 2 bit Stop. I bit Stop hanno il valore logico 1. La transizione da 1 a 0 all'inizio del bit Start avvisa il ricevente che la trasmissione dei dati sta per cominciare. Usando il proprio clock, il ricevente determina la posizione dei prossimi 8 bit, che vengono posti nel suo registro d'ingresso. I bit Stop che seguono il carattere trasmesso, che sono uguali a 1, assicurano che il bit Start del prossimo carattere sia riconosciuto. Quando la trasmissione si arresta, la linea rimane nello stato 1 finché non viene trasmesso un altro carattere.

Per assicurare la corretta ricezione, il ricevente ha bisogno di campionare il dato in ingresso il più vicino possibile al centro di ciascun bit. Ciò si realizza usando un segnale di clock la cui frequenza, f_R, è notevolmente più alta di quella del clock di trasmissione, f_T. Tipicamente $f_R = 16f_T$, ovvero si hanno 16 impulsi del clock locale durante ogni intervallo di un bit di dato. Questo clock è usato per incrementare un contatore modulo 16, che viene azzerato quando si rileva il fronte iniziale di un bit Start. Il centro del bit Start viene raggiunto al conteggio 8. Lo stato della linea d'ingresso è campionato nuovamente a questo punto per confermare che è un bit Start valido (a valore logico 0), e il contatore viene azzerato. Da questo punto in avanti, il segnale

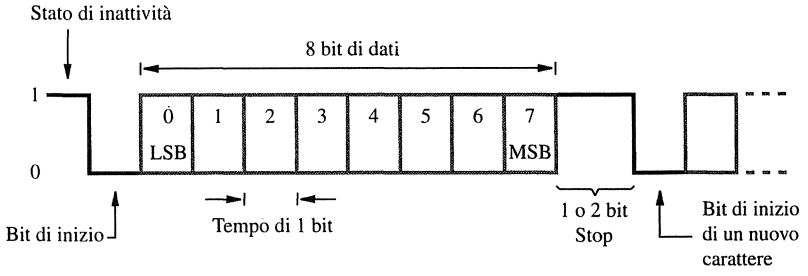


Figura 7.16 Trasmissione seriale asincrona di caratteri.

dei dati in ingresso viene campionato quando il contatore raggiunge 16 (cioè si riazzera), il che dovrebbe cadere in prossimità del centro di ciascun bit in ingresso. Quindi, purché $f_R/16$ sia sufficientemente prossimo a f_T, il ricevente caricherà correttamente i bit del carattere in ingresso.

Trasmissione sincrona

Nello schema Start-Stop descritto sopra, la posizione della transizione da 1 a 0 all'inizio del bit Start in Figura 7.16 è fondamentale per ottenere la corretta informazione di sincronizzazione. Questo schema è utile solo quando la velocità di trasmissione è sufficientemente bassa e le condizioni sul collegamento della trasmissione sono tali che le forme d'onda rettangolari mostrate in figura mantengono la loro forma. Per velocità più alte è necessario un metodo più affidabile affinché il ricevente recuperi l'informazione di sincronizzazione.

Nella trasmissione sincrona, il ricevente genera un clock che è sincronizzato a quello del trasmittente osservando le transizioni successive da 1 a 0 e da 0 a 1 nel segnale ricevuto. Esso aggiusta la posizione del fronte attivo del clock, così da cadere al centro della posizione del bit. Si usano una varietà di schemi di codifica per assicurare che avvengano abbastanza transizioni di segnale da abilitare il ricevente a generare un clock sincronizzato e a mantenere la sincronizzazione. Dal momento in cui si ottiene la sincronizzazione, la trasmissione di dati può continuare indefinitamente. I dati codificati vengono di solito trasmessi in grandi blocchi costituiti da varie centinaia o varie migliaia di bit. L'inizio e la fine di ciascun blocco sono marcati da appropriati codici e i dati in un blocco sono organizzati conformemente a un insieme di regole concordato. Le trasmissioni sincrone permettono frequenze di trasferimento dei dati molto alte.

7.5 Standard di interconnessione

Un tipico calcolatore da tavolo o portatile ha varie porte che possono essere usate per collegare dispositivi di I/O, come un mouse, una chiavetta di memoria o un'unità a disco. Si sono sviluppate interfacce standard per permettere ai dispositivi di I/O di usare interfacce indipendenti da qualsiasi processore specifico. Per esempio, una chiavetta di memoria che ha un connettore USB può essere usata con un qualsiasi calcolatore che abbia una porta USB. In questo paragrafo descriviamo brevemente alcuni degli standard di interconnessione in largo uso.

La maggior parte degli standard sono sviluppati grazie a una collaborazione tra un certo numero di aziende. In molti casi, l'IEEE (*Institute of Electrical and Electronics Engineers*) sviluppa ulteriormente questi standard e li pubblica come Standard IEEE.

7.5.1 Bus seriale universale (USB)

Il bus seriale universale (USB, *Universal Serial Bus*) [1] è lo standard di interconnessione più largamente usato. È disponibile una grande varietà di dispositivi con un connettore USB, fra i quali dispositivi di puntamento (*mouse*), chiavette di memoria, unità a disco, stampanti, macchine fotografiche digitali e molti altri. Il successo commerciale dell'USB è dovuto alla sua semplicità e basso costo. La specifica USB iniziale sostiene due velocità di funzionamento, chiamate bassa velocità (1,5 Megabit/s) e piena velocità (12 Megabit/s). Successivamente è stato introdotto l'USB 2, detto USB ad alta velocità. Questo permette trasferimenti di dati a velocità fino a 480 Megabit/s. A mano a mano che i dispositivi di I/O si sono evoluti con requisiti di velocità sempre più alta, è stato sviluppato l'USB 3 (detto superveloce), capace di trasferire dati fino a 5 Gigabit/s.

L'USB è stato progettato per ottenere vari obiettivi salienti.

- Fornire un sistema di interconnessione semplice, a basso costo e facile da usare.
- Servire una vasta gamma di dispositivi di I/O e tassi di trasferimento, incluse connessioni Internet e applicazioni audio e video.

- Migliorare la facilità d'uso attraverso una modalità di funzionamento "*plug-and-play*" ("inserisci e usa").

Si analizzeranno alcuni di questi obiettivi prima di discutere i dettagli tecnici dell'USB.

Caratteristiche del dispositivo

I generi di dispositivi che possono essere collegati a un calcolatore coprono un vasto insieme di funzionalità. I vincoli di velocità, volume e sincronizzazione associati ai trasferimenti di dati da e verso questi dispositivi può variare significativamente.

Nel caso di una tastiera, un dato di un byte è generato ogni volta che si preme un tasto, il che può accadere in qualunque momento. Questi dati dovrebbero essere trasferiti prontamente al calcolatore. Poiché l'evento di pressione di un tasto non è sincronizzato con nessun altro evento in un calcolatore, i dati generati dalla tastiera sono detti *asincroni*. Inoltre, la frequenza con cui i dati sono generati è piuttosto bassa. Questa frequenza è limitata dalla velocità dell'operatore umano, che è di circa 10 byte al secondo, ovvero meno di 100 bit al secondo.

Una varietà di dispositivi semplici che possono essere attaccati a un calcolatore generano dati di natura simile, cioè a bassa velocità e asincroni. Sistemi di puntamento per i calcolatori e alcuni altri controllori e manipolatori usati con i video-giochi ne sono validi esempi.

Si consideri adesso una sorgente di dati differente. Molti calcolatori hanno un microfono, interno o attaccato esternamente. Il suono rilevato dal microfono produce un segnale elettrico analogico, che deve essere convertito in un formato digitale prima che possa essere gestito dal calcolatore. Questo viene ottenuto campionando il segnale analogico periodicamente. Per ciascun campione, un convertitore da analogico a digitale (A/D) genera un numero a n bit che rappresenta l'ampiezza del segnale. Il numero di bit, n, viene selezionato in base alla precisione desiderata utile a rappresentare ciascun campione. Successivamente, quando questi dati vengono mandati a un altoparlante, un convertitore da digitale a analogico (D/A) viene usato per ripristinare il segnale analogico originale dal formato digitale. Un approccio simile è usato con le informazioni video di una fotocamera.

Il processo di campionamento genera un flusso continuo di campioni digitalizzati che arrivano a intervalli regolari, sincronizzati con il clock di campionamento. Tale flusso di dati è detto *isocrono*, a indicare che eventi successivi sono separati da intervalli di tempo uguali. Un segnale deve essere campionato con velocità sufficiente a tener traccia delle sue componenti di frequenza più alta. In generale, se la frequenza di campionamento è di s campioni al secondo, la componente di frequenza massima rilevata dal processo di campionamento è $s/2$. Per esempio, la voce umana può essere catturata adeguatamente con una frequenza di campionamento di 8 kHz, che registra segnali sonori aventi frequenze fino a 4 kHz. Per una qualità di suono più alta, necessaria in sistemi musicali, si usano frequenze di campionamento più alte. Una frequenza di campionamento standard per i suoni digitali è 44,1 kHz. Ogni campione è rappresentato da 4 byte di dati per coprire l'ampio intervallo di volume del suono (intervallo dinamico) necessario alla riproduzione del suono ad alta fedeltà. Questo porta a frequenze di dati di circa 1,4 Megabit/s.

Un requisito importante per gestire voce o musica campionata è quello di mantenere precisa la sincronizzazione nei processi di campionamento e riproduzione. Un'alta variabilità nella sincronizzazione dei campioni, o *jitter*, non è accettabile. Quindi, il meccanismo di trasferimento dei dati fra un calcolatore e un sistema musicale deve mantenere consistenti i ritardi da un campione al successivo, altrimenti saranno necessari complessi circuiti di bufferizzazione e risincronizzazione. D'altra parte, sono tollerabili errori occasionali e campioni mancanti. Questi non sono percepiti dall'ascoltatore oppure possono causare uno scatto non

intrusivo. Non occorre alcun meccanismo sofisticato per assicurare una consegna perfettamente corretta dei dati.

Trasferimenti di dati per immagini e video hanno requisiti simili, ma richiedono frequenze di trasferimento di dati molto più alte. Per mantenere la qualità dell'immagine della televisione commerciale, un'immagine dovrebbe essere rappresentata da circa 160 kilobyte e trasmessa 30 volte al secondo. Insieme alle informazioni di controllo, questo porta a una frequenza complessiva di 44 Megabit/s. Immagini a più alta qualità come per la TV ad alta definizione, o HDTV, richiedono frequenze più alte.

I grandi dispositivi di immagazzinamento come i dischi magnetici e ottici presentano requisiti differenti. Questi dispositivi fanno parte della gerarchia della memoria di un calcolatore, come sarà discusso nel Capitolo 8. La loro connessione al calcolatore richiede una larghezza di banda di trasferimento dati di almeno 40 o 50 Megabit/s. Ritardi dell'ordine di millisecondi vengono introdotti dal movimento dei componenti meccanici nel meccanismo del disco. Quindi, un piccolo ritardo aggiuntivo introdotto mentre si trasferiscono dati da o verso il calcolatore non è importante e il jitter non è un problema. Tuttavia, il meccanismo di trasferimento deve garantire la correttezza dei dati.

Plug-and-play

Quando un dispositivo di I/O viene collegato a un calcolatore, il sistema operativo ha bisogno di qualche informazione su di esso. Dovrà conoscere quale tipo di dispositivo è, così che possa usare il driver appropriato al dispositivo. Inoltre, dovrà conoscere gli indirizzi dei registri nell'interfaccia del dispositivo per essere in grado di comunicare con esso. Lo standard USB definisce sia l'hardware dell'USB sia il software che comunica con esso. La caratteristica di plug-and-play significa che quando un nuovo dispositivo viene collegato, il sistema rileva la sua esistenza automaticamente. Il software determina il tipo di dispositivo e come comunicare con esso, come pure qualsiasi requisito speciale che possa avere. Ne consegue che l'utente inserisce un dispositivo USB e quindi semplicemente inizia a usarlo, senza doversi curare di nessuno di questi dettagli.

L'USB si può inoltre inserire "a caldo", il che significa che un dispositivo può essere inserito o rimosso da una porta USB in presenza di alimentazione.

Architettura dell'USB

L'USB usa connessioni punto-punto e un formato di trasmissione seriale. Quando sono collegati più dispositivi, essi sono disposti in una struttura ad albero come mostrato in Figura 7.17. Ciascun nodo interno dell'albero ha un dispositivo chiamato *hub*, che agisce come punto di trasferimento intermedio tra il calcolatore e i dispositivi di I/O. Alla radice dell'albero, un *hub radice* collega l'intero albero al calcolatore. Le foglie dell'albero sono i dispositivi di I/O: un mouse, una tastiera, una stampante, una connessione a Internet, una fotocamera o un altoparlante. La struttura ad albero rende possibile il collegamento di molti dispositivi usando semplici collegamenti punto-punto.

Se si consente ai dispositivi di I/O di mandare messaggi in qualsiasi momento, due messaggi possono raggiungere l'hub allo stesso tempo e interferire l'un l'altro. Per questa ragione, l'USB funziona esclusivamente mediante polling. Un dispositivo può mandare un messaggio solo in risposta a un messaggio di interrogazione dal processore. Dunque, nessuna coppia di dispositivi può mandare messaggi allo stesso momento. Questo vincolo permette agli hub di essere dispositivi semplici e poco costosi.

A ciascun dispositivo sulla porta USB, sia esso un hub o un dispositivo di I/O, viene assegnato un indirizzo di 7 bit. Questo indirizzo è locale all'albero dell'USB e non ha alcuna relazione con lo spazio di indirizzamento del processore. L'hub radice dell'USB, che è connesso al processore, appare come un singolo dispositivo. Il software comunica con singoli

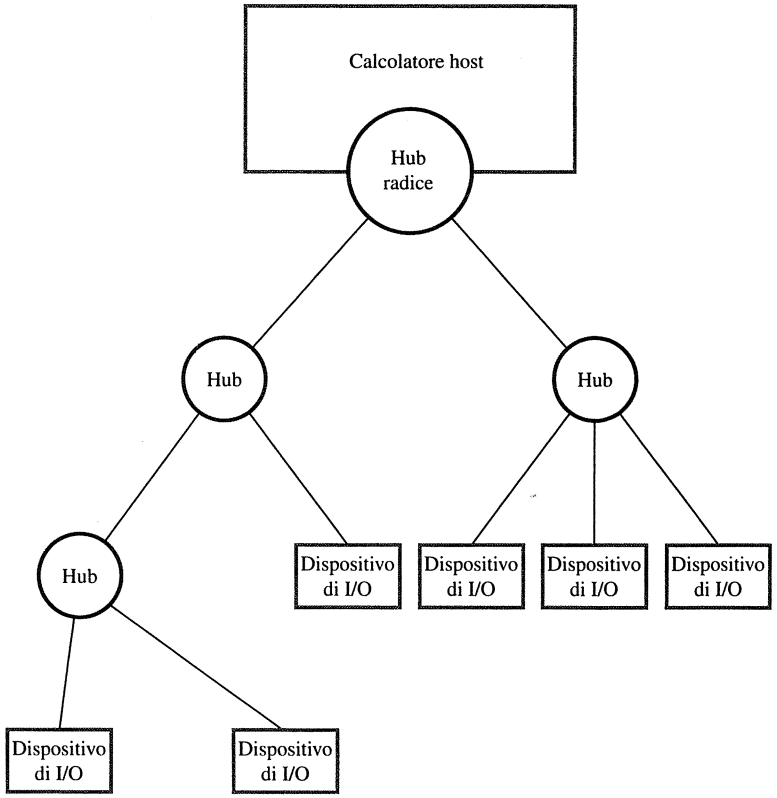


Figura 7.17 Struttura ad albero USB.

dispositivi mandando informazioni all'hub radice, che le inoltra al dispositivo appropriato nell'albero USB.

Quando un dispositivo è inizialmente collegato a un hub, o quando viene acceso, il suo indirizzo è 0. Il calcolatore, detto *host*, interroga periodicamente ciascun hub per raccogliere informazioni sullo stato e apprendere di nuovi dispositivi che siano stati aggiunti o rimossi. Quando l'host viene informato che è stato collegato un nuovo dispositivo, esso legge l'informazione in una memoria speciale nell'interfaccia USB del dispositivo per apprenderne le caratteristiche. Quindi assegna al dispositivo un indirizzo USB univoco e lo scrive in uno dei registri dell'interfaccia del dispositivo. Questa procedura di connessione iniziale conferisce all'USB la caratteristica plug-and-play.

Traffico isocrono su USB

Una caratteristica importante dell'USB è il suo supporto del trasferimento isocrono di dati in maniera semplice. Come detto in precedenza, i dati isocroni necessitano di essere trasferiti a intervalli regolari di tempo scanditi con precisione. Per servire questo tipo di traffico, l'hub

radice trasmette ogni millisecondo sull'albero USB una sequenza di bit univocamente riconoscibile. Questa sequenza di bit, detta carattere di inizio trama (*start of frame*), agisce come marcatore che indica l'inizio di dati isocroni, trasferiti dopo questo carattere. Quindi segnali audio e video digitali possono essere trasferiti a precisi intervalli di tempo regolari.

Caratteristiche elettriche

Le connessioni USB constano di quattro linee, di cui due per l'alimentazione, +5 V e Terra, e due per trasferire dati. Quindi i dispositivi di I/O che abbiano modesti requisiti di alimentazione possono essere alimentati direttamente dall'USB. Questo evita la necessità di un'alimentazione separata per dispositivi semplici come una chiavetta di memoria o un mouse.

Due sono i metodi usati per trasferire dati su un cavo USB. Quando si mandano dati a bassa velocità, viene trasmesso un alto voltaggio rispetto a Terra su una delle due linee per i dati per rappresentare uno 0 e sull'altra per rappresentare un 1. La linea Terra assorbe la corrente di ritorno in entrambi i casi. Tale schema in cui un segnale viene iniettato su una linea in relazione a Terra viene detto trasmissione *a terminazione singola* (*single-ended*).

La velocità con cui i dati possono essere mandati su un cavo è limitata dall'ammontare di rumore elettrico presente. Il termine *rumore* si riferisce a qualsiasi segnale che interferisce con il segnale dati desiderato e quindi può provocare errori. La trasmissione a terminazione singola è altamente suscettibile al rumore. Il voltaggio sulla linea Terra è comune a tutti i dispositivi collegati al calcolatore. I segnali mandati da un dispositivo possono causare piccole variazioni nel voltaggio della linea Terra e quindi interferire con i segnali mandati da un altro dispositivo. Le interferenze possono anche essere causate da una linea affetta da rumore propagato da altre linee.

L'USB ad alta velocità usa una disposizione alternativa conosciuta come *segnalazione differenziale*. Il segnale di dati è iniettato fra due linee di dati intrecciate. La linea Terra non è coinvolta. Il ricevente rileva la differenza di voltaggio direttamente fra le due linee del segnale, senza riferirsi alla Terra. Questa disposizione è molto efficace nella riduzione del rumore osservato dal ricevente, poiché qualsiasi rumore iniettato su una delle due linee di una coppia intrecciata è anche iniettato sull'altra. Poiché il ricevente è sensibile solo alla differenza di voltaggio tra le due linee, la componente di rumore viene cancellata. La linea Terra agisce da schermo per i dati sulla coppia intrecciata contro interferenze da linee vicine. La segnalazione differenziale permette l'uso di voltaggi molto più bassi e velocità molto più alte in confronto alla trasmissione a terminazione singola.

7.5.2 FireWire

FireWire è un altro standard di interconnessione diffuso. Fu originariamente sviluppato da Apple ed è stato adottato come Standard IEEE 1394 [2]. Come l'USB, usa collegamenti seriali punto-punto differenziali. Ecco alcune delle differenze salienti fra FireWire e USB.

- I dispositivi su un bus FireWire sono collegati a festone (*daisy chain*), anziché ad albero come sull'USB. Un dispositivo è collegato al calcolatore, un secondo dispositivo è collegato al primo, un terzo dispositivo è collegato al secondo, e così via.
- FireWire si adatta bene per collegare dispositivi audio e video. Può funzionare in un modo isocrono che è altamente ottimizzato per sostenere traffico isocrono ad alta velocità.
- I dispositivi di I/O collegati all'USB comunicano con il calcolatore host. Se i dati devono essere trasferiti da un dispositivo a un altro, per esempio da una fotocamera a uno schermo o stampante, essi vengono prima letti dall'host e quindi mandati allo schermo o stampante. FireWire invece permette di operare in un modo detto *peer-to-peer* (da pari a pari). Ciò significa che i dati possono essere trasferiti da un dispositivo di I/O a un altro senza intervento dell'host.

- Il connettore FireWire di base ha sei piedini. Vi sono due coppie di linee per i dati, una per la trasmissione in ciascuna direzione, e due per alimentazione e Terra. Le versioni ad alta velocità usano un connettore a nove piedini, con tre linee Terra aggiuntive per schermare le linee dei dati dalle interferenze.
- Il bus FireWire può fornire molta più potenza dell'USB. Quindi può supportare dispositivi aventi moderati requisiti di potenza.

FireWire è ampiamente usato con dispositivi audio e video. Per esempio, la maggior parte delle telecamere hanno una porta FireWire. Sono state definite svariate versioni dello standard che possono funzionare a velocità che vanno da 400 Megabit/s a 3,6 Gigabit/s.

7.5.3 Bus PCI

Il bus PCI (*peripheral component interconnect*, interconnessione di componenti periferici) [3] fu sviluppato come un bus a basso costo e indipendente dal processore. Esso è incorporato sulla scheda madre di un calcolatore e usato per collegare interfacce di I/O per una vasta varietà di dispositivi. Un dispositivo connesso al bus PCI appare al processore come se fosse connesso direttamente al bus del processore. Ai suoi registri di interfaccia sono assegnati indirizzi nello spazio di indirizzamento del processore.

Si inizierà descrivendo come funziona il bus PCI, poi si discuteranno alcune delle sue caratteristiche.

Struttura del bus

L'uso del bus PCI in un calcolatore è illustrato in Figura 7.18. Il bus PCI è collegato al bus del processore attraverso un controllore detto *ponte* (*bridge*). Il ponte ha una porta speciale per il collegamento alla memoria principale. Esso può anche avere un'altra porta speciale ad alta velocità per collegare dispositivi grafici. Il ponte traduce e ritrasmette comandi e risposte da un bus all'altro e trasferisce dati fra essi. Per esempio, quando il processore manda una richiesta di Read a un dispositivo di I/O, il ponte inoltra il comando e l'indirizzo al bus PCI. Quando il ponte riceve la risposta del dispositivo, esso inoltra i dati al processore usando il bus del processore. I dispositivi di I/O sono collegati al bus PCI, possibilmente attraverso porte che usano standard come Ethernet, USB, SATA, SCSI o SAS.

Il bus PCI supporta tre spazi di indirizzi indipendenti: memoria, I/O e configurazione. Il progettista del sistema può scegliere di usare lo spazio unificato degli indirizzi di memoria e di I/O persino con un processore che ha uno spazio di indirizzi di I/O separato. In effetti, questo è l'approccio raccomandato dallo standard PCI per ottenere una più ampia compatibilità. Lo spazio di configurazione serve per la caratteristica plug-and-play del bus PCI, come sarà spiegato fra breve. Un comando a 4 bit che accompagna l'indirizzo identifica quale dei tre spazi è usato in una certa operazione di trasferimento di dati.

I trasferimenti di dati su un bus del calcolatore spesso riguardano *raffiche* (*burst*) di dati invece che singole parole. Le parole immagazzinate in locazioni di memoria successive sono trasferite direttamente tra la memoria e un dispositivo di I/O come un disco o una connessione Ethernet. I trasferimenti di dati vengono avviati dall'interfaccia del dispositivo di I/O, che agisce come master del bus. Questo modo di trasferire dati direttamente tra la memoria e i dispositivi di I/O è discusso in dettaglio nel Capitolo 8. Il bus PCI è progettato principalmente per sostenere trasferimenti di molte parole. Un'operazione Read o Write di una singola parola è semplicemente trattata come una raffica di dati di lunghezza uno.

Il protocollo di segnalazione sul bus PCI è simile a quello mostrato in Figura 7.5, con un'importante differenza. Il bus PCI usa le stesse linee per trasferire sia indirizzi sia dati. In Figura 7.5, si è assunto che il master mantenga l'informazione di indirizzo sul bus finché il

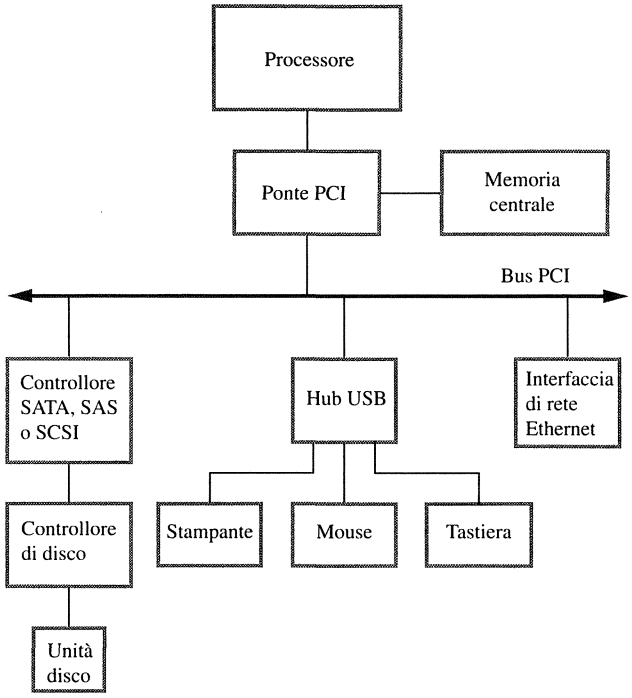


Figura 7.18 Uso del bus PCI in un calcolatore.

trasferimento di dati non sia completato. Tuttavia, questo non è necessario. Occorre mantenere l'indirizzo solo fino a quando lo slave viene selezionato, liberando così le linee per inviare dati nei cicli di clock successivi. Per trasferimenti di più parole, lo slave può immagazzinare l'indirizzo in un registro interno e incrementarlo per accedere a locazioni di memoria successive. Si può così ottenere una significativa riduzione di costo, poiché il numero di linee del bus è un fattore importante che influenza il costo di un calcolatore.

Trasferimento di dati

Per capire il funzionamento del bus e le sue varie caratteristiche, si esaminerà una tipica transazione su bus. Il master del bus, che è il dispositivo che avvia i trasferimenti di dati emettendo comandi Read e Write, è chiamato l'*iniziatore* (*initiator*) nella terminologia PCI. Il dispositivo indirizzato che risponde a questi comandi è chiamato *obiettivo* (*target*). I principali segnali del bus usati per trasferire dati sono elencati in Tabella 7.1. Vi sono 32 o 64 linee che portano indirizzi e dati usando uno schema di segnalazione sincrona simile a quello di Figura 7.5. Il segnale di Obiettivo-pronto, TRDY#, equivale al segnale Slave-pronto in quella figura. In aggiunta, il PCI usa un segnale di Iniziatore-pronto, IRDY#, per sostenere trasferimenti a raffica. Si descriveranno brevemente questi segnali per far comprendere al lettore le caratteristiche principali del bus.

Un'operazione di trasferimento completa sul bus PCI, che riguarda un indirizzo e una raffica di dati, è chiamata *transazione*. Si consideri una transazione del bus in cui un iniziatore

Tabella 7.1 Segnali di trasferimento di dati sul bus PCI

Nome	Funzione
CLK	Un clock a 33 MHz o 66 MHz
FRAME#	Inviato dall'iniziatore per indicare la durata di una trasmissione
AD	32 linee di indirizzi/dati, che possono essere opzionalmente incrementate a 64
C/BE#	4 linee per comando/abilita-byte (8 per un bus a 64 bit)
IRDY#, TRDY#	Segnali Iniziatore-pronto e Obiettivo-pronto
DEVSEL#	Una risposta dal dispositivo che indica che ha riconosciuto il suo indirizzo ed è pronto per una transazione di trasferimento di dati
IDSEL#	Selezione di dispositivo da inizializzare

legge quattro parole a 32 bit consecutive dalla memoria. La sequenza di eventi sul bus è illustrata in Figura 7.19.

Tutte le transizioni di segnali iniziano sul fronte di salita del clock. Come nel caso di Figura 7.5, sono mostrati i segnali che cambiano successivamente nel ciclo di clock per indicare i ritardi a cui vanno incontro. Un segnale il cui nome termina con il simbolo # è asserito quando è nello stato di basso voltaggio.

Il master del bus, che agisce come iniziatore, asserisce FRAME# nel ciclo di clock 1 per indicare l'inizio di una transazione. Allo stesso tempo, manda l'indirizzo sulle linee AD e un comando sulle linee C/BE#. In questo caso, il comando indicherà che è richiesta un'operazione Read e che viene usato lo spazio degli indirizzi in memoria.

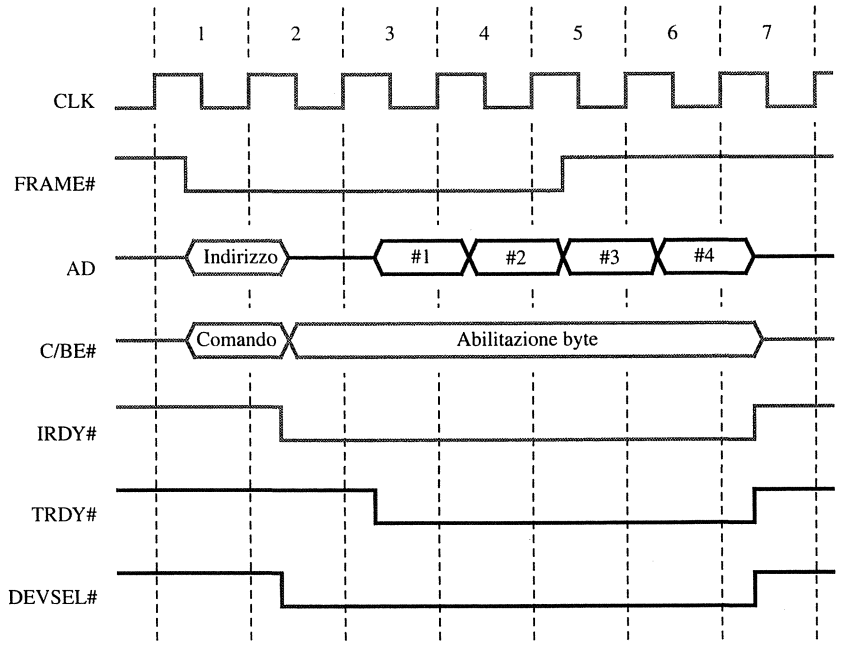


Figura 7.19 Un'operazione di lettura sul bus PCI.

Nel ciclo di clock 2, l'iniziatore rimuove l'indirizzo, disconnette i suoi piloti dalle linee AD e asserisce IRDY# per indicare che è pronto a ricevere dati. L'obiettivo selezionato asserisce DEVSEL# per indicare che ha riconosciuto il suo indirizzo ed è pronto a rispondere. Nello stesso momento, abilita i suoi piloti sulle linee AD, così che può mandare dati all'iniziatore nei cicli seguenti. Il secondo ciclo di clock è usato per dare spazio ai ritardi dovuti all'inversione delle linee AD, affinché l'iniziatore disabiliti i suoi piloti e l'obiettivo abiliti i suoi. L'obiettivo asserisce TRDY# nel ciclo di clock 3 e inizia a mandare dati. Esso mantiene DEVSEL# nello stato asserito fino alla fine della transazione.

Si è assunto che l'obiettivo sia pronto a mandare dati nel ciclo di clock 3. Se non lo è, ritarderà l'asserzione di TRDY# finché non diventerà pronto. L'intera raffica di dati non necessita di essere mandata in cicli di clock successivi. Sia l'iniziatore sia l'obiettivo possono introdurre una pausa disattivando il proprio segnale di pronto, quindi asserendolo nuovamente quando si è in grado di riprendere il trasferimento di dati.

Le linee C/BE#, che sono usate per mandare un comando del bus nel ciclo di clock 1, sono usate per uno scopo diverso durante il resto della transazione. Ognuna di queste quattro linee è associata a un byte sulle linee AD. L'iniziatore asserisce una o più delle linee C/BE# per indicare quali gruppi di linee da un byte devono essere usati per trasferire dati.

L'iniziatore usa il segnale FRAME# per indicare la durata della raffica. Esso disattiva questo segnale durante il trasferimento della penultima parola. In Figura 7.19, l'iniziatore mantiene FRAME# nello stato asserito fino al ciclo di clock 5, ovvero il ciclo in cui riceve la terza parola. In risposta, l'obiettivo manda un'altra parola nel ciclo di clock 6, quindi si arresta. Dopo aver mandato la quarta parola, l'obiettivo disattiva TRDY# e DEVSEL# e disconnette i suoi piloti dalle linee AD.

Configurazione del dispositivo

Quando un dispositivo di I/O viene collegato a un calcolatore, varie azioni sono necessarie per configurarne l'interfaccia e il software che comunica con essa. Analogamente all'USB, PCI ha una funzionalità di plug-and-play che semplifica notevolmente questo processo. In effetti, la caratteristica di plug-and-play è stata introdotta per prima dallo standard PCI. Un'interfaccia PCI è dotata di una piccola memoria ROM di configurazione che contiene le informazioni sul dispositivo di I/O collegato. Le ROM di configurazione di tutti i dispositivi sono accessibili nello spazio degli indirizzi di configurazione, dove esse vengono lette dal software di inizializzazione del PCI quando il sistema viene acceso o fatto ripartire. Leggendo l'informazione nella ROM di configurazione, il software determina se il dispositivo è una stampante, una fotocamera, un'interfaccia Ethernet o un controllore del disco. Esso può ulteriormente apprendere varie opzioni e caratteristiche del dispositivo.

Ai dispositivi collegati al bus PCI non sono assegnati indirizzi permanenti predeterminati nell'hardware delle loro interfacce di I/O. Invece, gli indirizzi dei dispositivi vengono assegnati dal software durante il processo di configurazione iniziale. Questo significa che quando si attiva l'alimentazione, non si può accedere ai dispositivi tramite i loro indirizzi nel solito modo, poiché a essi non è stato ancora assegnato alcun indirizzo. Si usa un meccanismo diverso per selezionare i dispositivi di I/O in tale fase.

Il bus PCI può avere fino a 21 connettori per il collegamento di schede di interfaccia di dispositivi di I/O. Ogni connettore dispone di un piedino detto Selettore di inizializzazione del dispositivo (IDSEL#). Questo piedino è collegato a una delle 21 linee indirizzo/dati più alte, da AD11 a AD31. Un'interfaccia di dispositivo risponde a un comando di configurazione se il suo ingresso IDSEL# è asserito. Il software di configurazione scandisce tutte le 21 locazioni per identificare dove siano presenti delle interfacce di dispositivi di I/O. Per ciascuna locazione, emette un comando di configurazione usando un indirizzo in cui la linea AD corrispondente a quella locazione è posta a 1 e le rimanenti 20 linee sono

poste a 0. Se un'interfaccia di dispositivo risponde, le si assegna un indirizzo, e questo viene scritto in uno dei suoi registri designati a tal fine. Usando lo stesso meccanismo di indirizzamento, il processore legge la ROM di configurazione del dispositivo ed effettua l'inizializzazione necessaria. Esso usa i bit di indirizzo di ordine più basso, da AD0 a AD10, per accedere a locazioni nella ROM di configurazione. Questo processo automatico permette all'utente di limitarsi semplicemente a inserire la scheda di interfaccia e attivare l'alimentazione. Il software fa il resto.

Il bus PCI ha guadagnato grande popolarità, particolarmente nel mondo dei PC. È anche usato in molti altri calcolatori, per avvantaggiarsi del vasto numero di dispositivi di I/O per i quali è disponibile un'interfaccia PCI. Sono disponibili sia la configurazione a 32 bit sia quella a 64 bit, con clock a 33 MHz o 66 MHz. È anche disponibile una variante ad alte prestazioni chiamata PCI-X, un bus a 64 bit che opera a 133 MHz. Versioni del PCI-X a prestazioni ancora più alte operano a velocità superiori, fino a 533 MHz.

7.5.4 Bus SCSI

L'acronimo SCSI indica *Small Computer System Interface* (*interfaccia per piccoli sistemi di calcolo*) [4]. Si riferisce a un bus standard definito dall'*American National Standards Institute* (ANSI). Il bus SCSI può essere usato per collegare una varietà di dispositivi a un calcolatore. È particolarmente adatto per unità a dischi. Lo si trova spesso in installazioni come basi di dati istituzionali o sistemi di posta elettronica, nei quali si impiegano molti dischi.

Nelle specifiche originali dello standard SCSI, i dispositivi sono collegati a un calcolatore attraverso un cavo da 50 linee, che può essere lungo fino a 25 metri e può trasferire dati a una frequenza fino a 5 Megabyte/s. Lo standard ha attraversato molte revisioni e la sua capacità di trasferimento dati è rapidamente cresciuta. Sono stati definiti SCSI-2 e SCSI-3 e ognuno ha parecchie opzioni. I dati sono trasferiti a gruppi di 8 bit o 16 bit in parallelo, con frequenze di clock fino a 80 MHz. Ci sono anche varie opzioni per lo schema di segnalazione elettrica. Il bus può usare trasmissioni a terminazione singola, dove ciascun segnale usa una sola linea, con una linea Terra di ritorno comune per tutti i segnali. In un'altra opzione si usa la segnalazione differenziale, con un coppia di linee per ciascun segnale.

Trasferimento di dati

I dispositivi collegati al bus SCSI non fanno parte dello spazio degli indirizzi del processore allo stesso modo dei dispositivi collegati al bus del processore o al bus PCI. Un bus SCSI può essere collegato direttamente al bus del processore, o più probabilmente a un altro bus di I/O standard come PCI, attraverso un controllore SCSI. Dati e comandi vengono trasferiti in forma di messaggi a più byte chiamati pacchetti. Per mandare comandi o dati a un dispositivo, il processore assembla l'informazione nella memoria e quindi istruisce il controllore SCSI per trasferirla al dispositivo. Analogamente, quando i dati vengono letti da un dispositivo, il controllore trasferisce i dati alla memoria e quindi informa il processore generando un'interruzione.

Per illustrare il funzionamento del bus SCSI si considera come lo si possa usare con un disco. La comunicazione con un disco differisce notevolmente dalla comunicazione con la memoria principale. I dati sono immagazzinati su un disco in blocchi chiamati *settori*, dove ciascun settore può contenere varie centinaia di byte. Quando un file di dati viene scritto sul disco, non sempre è immagazzinato in settori contigui. Alcuni settori possono già contenere informazioni precedentemente immagazzinate; altri possono essere difettosi e devono essere evitati. Quindi una richiesta di Read o Write può dare luogo ad accessi a vari settori del disco che non necessariamente sono contigui. Per i limiti del movimento meccanico del disco si ha un lungo ritardo, dell'ordine di diversi millisecondi, prima di raggiungere il primo settore da

o verso cui i dati devono essere trasferiti. Dopo di ciò viene trasferita una raffica di dati ad alta velocità. Un altro ritardo può intervenire per raggiungere il successivo settore, seguito da una raffica di dati. Una singola richiesta Read o Write può dare luogo a parecchie raffiche di dati. Il protocollo SCSI è progettato per facilitare questo modo di funzionamento.

Si esamina ora un'operazione Read completa come esempio. Nel seguito se ne dà una descrizione di alto livello semplificata, ignorando dettagli e convenzioni di segnalazione. Si assume che il processore voglia leggere un blocco di dati da una unità a disco e che questi dati siano contenuti in due settori del disco non contigui. Il processore invia un comando al controllore SCSI, che dà luogo alla serie di eventi che segue.

1. Il controllore SCSI contende per il controllo del bus SCSI.
2. Quando vince il processo di arbitraggio, il controllore SCSI invia un comando al controllore del disco specificando l'operazione Read richiesta.
3. Il controllore del disco non può iniziare a trasferire i dati immediatamente. Deve prima spostare la testina di lettura del disco sul settore desiderato, quindi invia un messaggio al controllore SCSI che indica che sospenderà temporaneamente la loro connessione. Il bus SCSI è adesso libero per essere usato da altri dispositivi.
4. Il controllore del disco invia un comando all'unità a disco per spostare la testina di lettura sul primo settore relativo all'operazione Read richiesta. Legge i dati immagazzinati in quel settore e li pone in un buffer dati. Quando è pronto a iniziare il trasferimento dei dati, richiede il controllo del bus. Dopo aver vinto l'arbitraggio, ristabilisce la connessione con il controllore SCSI, invia il contenuto del buffer dati, quindi sospende nuovamente la connessione.
5. Il processo viene ripetuto per leggere e trasferire il contenuto del secondo settore del disco.
6. Il controllore SCSI trasferisce i dati richiesti alla memoria principale e invia un'interruzione al processore, per indicare che i dati sono adesso disponibili.

Questo scenario mostra che i messaggi scambiati sul bus SCSI sono a un livello più alto di quelli scambiati sul bus del processore. I messaggi si riferiscono a operazioni più complesse che possono richiedere diversi passi per essere completate, a seconda del dispositivo. Né il processore né il controllore SCSI hanno bisogno di conoscere i dettagli di funzionamento del disco e di come avvenga lo spostamento da un settore al successivo.

Lo standard del bus SCSI definisce un'ampia varietà di messaggi di controllo che possono essere usati per gestire differenti tipi di dispositivi di I/O. Sono anche definiti messaggi per gestire vari errori o condizioni di fallimento che possono insorgere durante il funzionamento del dispositivo o il trasferimento di dati.

7.5.5 SATA

Ai primordi dell'era dei calcolatori personali, il bus di un diffuso calcolatore IBM detto AT, basato sul bus del microprocessore Intel 8080, diventò uno standard industriale. Esso era denominato ISA, che sta per *Industry Standard Architecture*. Una versione avanzata, che includeva una definizione del software di base necessario per supportare unità a disco, fu successivamente chiamata ATA, ovvero bus *AT Attachment*. Una versione seriale della stessa architettura divenne nota come SATA [5], ed è ora largamente usata come interfaccia per dischi. Come tutti gli standard, sono state sviluppate varie versioni di SATA con caratteristiche aggiuntive e velocità più alte. La versione parallela originale è stata ridenominata PATA, ma non è più usata nei nuovi dispositivi.

Il connettore SATA di base ha 7 piedini, e collega due coppie intrecciate e tre linee Terra. Viene usata la segnalazione differenziale, con frequenze di clock che vanno da 1,5 a 6,0 Gigabit/s. Alcune delle versioni recenti offrono una modalità di trasmissione isocrona per supportare dispositivi audio e video.

7.5.6 SAS

Questa è una realizzazione seriale del bus SCSI, donde il suo nome: *Serially Attached SCSI* [6]. È principalmente inteso per connettere dischi magnetici e unità CD e DVD. Impiega collegamenti seriali punto-punto simili a SATA. Un collegamento SAS può trasferire dati in entrambe le direzioni simultaneamente, a velocità fino a 12 Gigabit/s. Al livello del software, SAS è pienamente compatibile con SCSI.

7.5.7 PCI Express

I requisiti di prestazione posti alle interconnessioni di I/O sono in crescita continua. Connessioni Internet, dispositivi grafici sofisticati, flussi video (in *streaming*) e televisione ad alta definizione sono esempi di applicazioni che richiedono trasferimenti di dati a velocità molto elevate. Lo standard di interconnessione PCI Express (spesso chiamato PCIe) [7] è stato sviluppato per venire incontro a questi bisogni e per anticipare crescite ulteriori del tasso di trasferimento di dati, che sono inevitabili con l'introduzione di nuove applicazioni.

Il PCI Express usa collegamenti seriali punto-punto interconnessi tramite commutatori per formare una struttura ad albero, come mostrato in Figura 7.20. Il nodo radice dell'albero, detto il *complesso radice*, è collegato al bus del processore. Il complesso radice ha una porta speciale collegata alla memoria principale. Tutte le altre connessioni che emanano dal complesso radice sono collegamenti seriali a dispositivi di I/O. Alcuni di questi collegamenti possono connettere un commutatore che porta ad altri rami seriali, come mostrato in figura. Il commutatore può anche collegare interfacce ponte che supportano altri standard, come PCI o USB. Per esempio, uno dei rami dell'albero potrebbe essere un bus PCI, per trarre vantaggio dalla vasta varietà di dispositivi per i quali esistono già interfacce PCI.

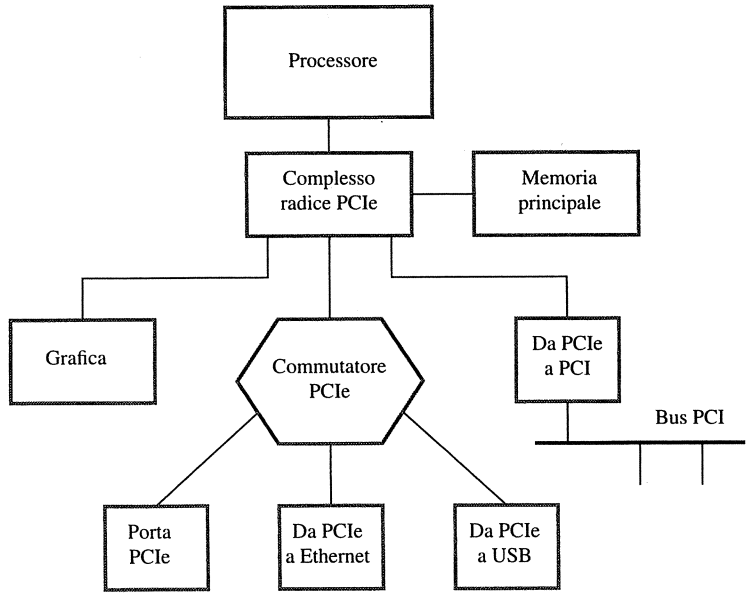


Figura 7.20 Connessioni PCI Express.

Il collegamento PCI Express di base consiste di due coppie di linee intrecciate, una per ciascuna direzione della trasmissione. I dati sono trasmessi alla frequenza di 2,5 Gigabit/s su ciascuna coppia intrecciata, usando lo schema di segnalazione differenziale descritto nel Paragrafo 7.5.1, fra le caratteristiche elettriche dei collegamenti USB ad alta velocità. I dati possono essere trasferiti simultaneamente in entrambe le direzioni. Inoltre, collegamenti a dispositivi diversi possono trasferire dati contemporaneamente, poiché non si ha un bus condiviso come nel caso di PCI o SCSI. In aggiunta, un collegamento può usare più di una coppia di linee intrecciate in ciascuna direzione. L'organizzazione di base con una coppia intrecciata per ciascuna direzione è detta corsia, riferita anche come connessione X1 (pronunciata "a 1"). Un collegamento può usare 2, 4, 8 o 16 corsie, nel qual caso è detto collegamento X2, X4, X8 o X16.

Il ricevente su un collegamento per trasmissione sincrona deve sincronizzare il suo clock con quello del trasmittente, come descritto nel Paragrafo 7.4.2. Per far sì che questo sia possibile, i dati trasmessi sono codificati per assicurare che transizioni da 0 a 1 e da 1 a 0 avvengano abbastanza frequentemente. Nel caso di PCIe, ciascun dato di 8 bit viene codificato usando 10 bit. Altri bit sono inseriti nel flusso per permettere vari controlli, come delimitare l'informazione di indirizzi e di dati. Tenendo conto dei bit aggiuntivi, una singola coppia intrecciata su cui sono trasmessi dati a 2,5 Gigabit/s trasferisce in realtà 1,6 Gigabit/s o 200 Mbyte/s di informazioni utili. Un collegamento X16 trasferisce dati alla frequenza di 3,2 Gigabyte/s in ciascuna direzione. Per confronto, un bus PCI da 64 bit che lavora a 64 MHz ha un picco di frequenza di trasferimento dati aggregati di 512 Megabyte/s. PCI Express ha il vantaggio aggiuntivo di usare un minore numero di linee, con il risultato di avere hardware a costo più basso.

I protocolli PCI Express sono pienamente compatibili con quelli PCI. Per esempio, sono usate le stesse procedure di configurazione iniziale. Quindi, un calcolatore che usa PCI Express può usare i sistemi operativi e le applicazioni esistenti che erano state sviluppate per un sistema basato su PCI.

Sommario

Questo capitolo ha introdotto la struttura di I/O di un calcolatore dal punto di vista dell'hardware. Dispositivi di I/O collegati a un bus sono stati presi quali esempi per illustrare gli schemi sincroni e asincroni di trasferimento dati.

L'architettura di reti di interconnessione per dispositivi di ingresso e di uscita è stata un'importante area di sviluppo, spinta da sempre più crescenti necessità di trasferimento di dati ad alta velocità, di riduzione dei costi, e di caratteristiche che migliorino la facilità d'uso quali il plug-and-play. In questo capitolo si sono brevemente descritti vari standard di I/O che illustrano gli approcci usati per raggiungere questi obiettivi. La tendenza attuale è di minore interesse ai bus paralleli, in favore dei collegamenti punto-punto. I collegamenti seriali hanno costi più bassi e possono trasferire dati ad alta velocità.

Note bibliografiche

1. *Universal Serial Bus Specification*, disponibile presso http://www.usb.org/developers.
2. *IEEE Standard for a High-Performance Serial Bus*, IEEE Std. 1394-2008, Ottobre 2008; disponibile presso http://www.1394ta.org.
3. Specifiche e documentazione di *PCI Local Bus* e *PCI Express* disponibili presso http://www.pcisig.com/developers.
4. *SCSI-3 Architecture Model (SAM)*, ANSI Standard X3.270, 1996; disponibile con altri documenti SCSI presso http://www.ansi.org.
5. Specifiche e documentazione di *SATA* disponibili presso http://www.serialata.org.
6. Informazione su *Serial SCSI (SAS)*, disponibile presso http://www.scsita.org.
7. Wilen A., Schade J.,Thornburg R., *Introduction to PCI Express, A Hardware and Software Developer's Guide*, Intel Press, 2003.

8 Sistema di memoria

In questo capitolo

- Circuiti di base per la memoria
- Organizzazione della memoria principale
- Tecnologie per la memoria
- Accesso diretto alla memoria come meccanismo di I/O
- Memoria cache, per ridurre l'effettivo tempo di accesso alla memoria
- Memoria virtuale, per accrescere la dimensione apparente della memoria principale
- Dischi magnetici e ottici usati per la memoria secondaria

Il programma correntemente in esecuzione e i dati da elaborare si devono trovare nella memoria centrale. Il capitolo è appunto dedicato a spiegare come sia fatta e funzioni questa parte così essenziale del calcolatore. Si tenga bene in mente fin da ora che la velocità di esecuzione del programma dipende fortemente dalla frequenza di trasferimento di istruzioni e dati tra processore e memoria. Inoltre la prestazione del calcolatore aumenta se esso è dotato di memoria grande, onde riuscire a contenere tutti i dati di cui il programma ha bisogno.

Idealmente si vorrebbe avere memoria grande, veloce e per giunta poco costosa, ma simili desideri sono mutuamente incompatibili, perché aumentare capacità e frequenza di trasferimento di dati fa inevitabilmente lievitare il costo della memoria. Per risolvere il problema la ricerca si è ingegnata di studiare numerose soluzioni strutturali atte ad aumentare capacità e velocità apparenti della memoria, tenendone però il costo a livello ragionevole.

La memoria di un calcolatore consiste di un gerarchia, che include una cache, la memoria principale e la memoria secondaria, come spiega il Capitolo 1. In questo capitolo, si descriveranno i componenti e le organizzazioni più comunemente usati per realizzare queste unità. La memoria ad accesso diretto viene introdotta come meccanismo per trasferire dati fra un dispositivo di I/O, come un disco, e la memoria principale, con il minimo coinvolgimento del processore. Si esaminerà la velocità della memoria e si discuterà come si possano ridurre i tempi di accesso alla memoria tramite le cache. Quindi si presenterà il concetto di memoria virtuale, che fa uso della grande capacità di immagazzinamento delle unità di memoria secondaria per accrescere la dimensione effettiva della memoria. Si inizierà con una presentazione di alcuni concetti di base, per estendere la discussione avviata nei Capitoli 1 e 2.

8.1 Concetti fondamentali

La capacità massima della memoria installabile nel calcolatore è determinata dallo schema di indirizzamento. Per esempio, se la parola di memoria è la minima unità indirizzabile, il

calcolatore con indirizzo da 16 bit può avere fino a 2^{16} = 64 K (64 Kilo)[1] parole di memoria, similmente con indirizzo da 32 bit può arrivare fino a 2^{32} = 4 G (4 Giga) parole e infine con 64 bit di indirizzo può avere fino a 2^{64} = 16 E (Exa) parole. Il numero totale di parole contenibili nella memoria rappresenta essenzialmente la misura della grandezza dello spazio di indirizzamento del calcolatore, o se si vuole la capacità di memoria.

La memoria è solitamente progettata per immagazzinare e recuperare dati in quantità di lunghezza di parola. Si consideri, per esempio, un calcolatore indirizzabile a byte le cui istruzioni generino indirizzi a 32 bit. Quando un indirizzo a 32 bit viene inviato dal processore all'unità di memoria, i 30 bit di ordine più alto determinano a quale parola si accederà. Se è specificata una quantità di un byte, i 2 bit di ordine più basso dell'indirizzo indicano quale locazione di un byte è interessata.

La connessione fra il processore e la sua memoria consiste di linee di indirizzo, di dato e di controllo, come mostrato in Figura 8.1. Il processore usa le linee di indirizzo per specificare la locazione di memoria interessata da un'operazione di trasferimento di dati e usa le linee di dato per trasferire il dato. Al contempo, le linee di controllo recano il comando che indica un'operazione Read o Write e l'indicazione se sia da trasferire un byte o una parola. Le linee di controllo recano inoltre la necessaria informazione di sincronizzazione e sono usate dalla memoria per indicare quando ha completato l'operazione richiesta. Quando l'interfaccia processore-memoria riceve la risposta dalla memoria, asserisce il segnale MFC mostrato in Figura 5.19. Questo è il segnale di controllo interno del processore che indica che l'operazione con la memoria richiesta è stata completata. Quando viene asserito, il processore procede al prossimo passo nella sua sequenza di esecuzione.

Per quantificare la velocità della memoria è utile definire come parametro l'intervallo di tempo intercorrente tra l'inizio dell'operazione e il rispettivo completamento, per esempio prendendone gli estremi tra le attivazioni del comando di lettura e di riscontro. Tale intervallo di tempo si chiama *tempo di accesso alla memoria* (*memory access time*) o anche *tempo di lavoro di memoria*. C'è un altro parametro importante da considerare, l'intervallo di tempo di durata minima tra l'inizio di due operazioni di memoria consecutive, chiamato *tempo di*

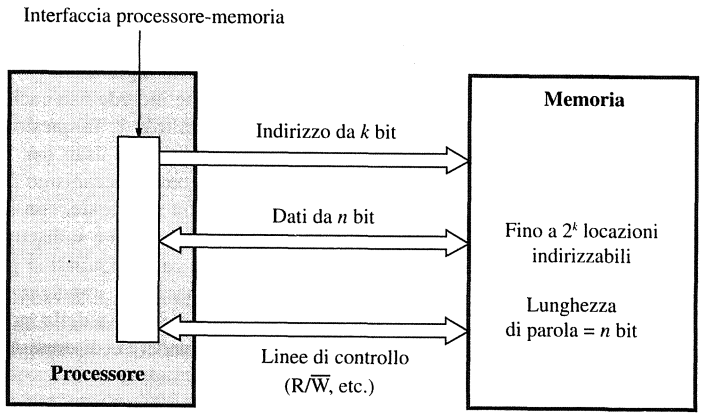


Figura 8.1 Connessione della memoria al processore.

[1] Qui K = 1024 = 2^{10} e si scrive in maiuscolo, ma si ricordi che k = 1000 = 10^3 va scritto in minuscolo come prescrivono le direttive del sistema metrico decimale (MKS).

ciclo di memoria (*memory cycle time*), per esempio misurandolo tra due attivazioni consecutive del comando di lettura nel bus. Di solito il tempo di ciclo dura un po' di più del tempo di accesso, secondo i dettagli tecnologici dell'unità di memoria.

Se è possibile avere accesso a qualsiasi parola di memoria in un intervallo di tempo di durata costante, sia in lettura sia in scrittura, indipendentemente dall'indirizzo, la memoria è detta *ad accesso casuale* o RAM (*Random Access Memory*). La definizione serve per distinguere rispetto ad altre forme di memoria che funzionano in modo più o meno seriale, come il disco magnetico o ottico e il nastro magnetico. Infatti queste due ultime forme di memoria di massa hanno tempo di accesso dipendente dall'indirizzo e dalla posizione del dato sul disco o nel nastro, rispettivamente.

Di norma la memoria centrale del calcolatore è interamente realizzata in tecnologia microelettronica ed è costituita da circuiti integrati a scala di integrazione molto grande (VLSI). Nei paragrafi seguenti si daranno alcune nozioni basilari intorno alla struttura interna e al funzionamento del componente integrato di memoria. Più avanti verranno invece illustrate e spiegate metodologie e tecniche per aumentare la prestazione della memoria centrale.

Cache e memoria virtuale

Di solito il processore del calcolatore sarebbe in grado di eseguire istruzioni ed elaborare dati più rapidamente di quanto essi non siano trasferibili tra memoria e processore stesso. Insomma, la durata del ciclo di memoria costituisce il limite principale di tutto il sistema. Una metodologia dimostratasi efficace nel diminuire la durata dell'intervallo di tempo di accesso consiste nell'introdurre la cosiddetta *memoria cache*. Si tratta di un'unità di memoria di capacità piccola ma velocità elevata, interposta tra processore e memoria centrale, molto più grande ma anche più lenta, destinata a contenere la porzione attiva di programma e i rispettivi dati su cui esso sta lavorando.

La *memoria virtuale* (*virtual memory*) è un'altra metodologia importante per aumentare la prestazione del sistema di memoria complessivo. Con questa tecnica solo le porzioni attive di un programma sono poste nella memoria principale; il resto è posto nel dispositivo di memoria secondaria, che è molto più grande. Parti del programma sono trasferite avanti e indietro tra la memoria principale e il dispositivo di memoria secondaria in maniera trasparente rispetto al programma dell'applicazione. Di conseguenza, il programma dell'applicazione vede una memoria che è molto più grande della memoria principale fisica del calcolatore.

Trasferimenti a blocco

La discussione qui sopra mostra che i dati si spostano frequentemente fra la memoria principale e la cache e fra la memoria principale e il disco. Questi trasferimenti non avvengono una parola alla volta. I dati sono sempre trasferiti in blocchi contigui che riguardano decine, centinaia o migliaia di parole. Pure i trasferimenti di dati fra la memoria principale e i dispositivi ad alta velocità come uno schermo grafico o un'interfaccia Ethernet coinvolgono grandi blocchi di dati. Quindi un parametro critico per le prestazioni della memoria principale è la sua possibilità di leggere o scrivere blocchi di dati ad alta velocità. Questa è una considerazione importante che ricorrerà spesso quando si discuterà di tecnologia della memoria e di organizzazione del sistema di memoria.

8.2 Memoria RAM a semiconduttori

Esistono numerose tecnologie microelettroniche di memoria a semiconduttori, caratterizzate da spettro ampio di velocità di funzionamento. Il tempo di accesso di tali memorie varia pressapoco da 100 ns (cento nanosecondi) a 1 ns (un nanosecondo). Nel paragrafo verranno illustrate le caratteristiche principali della tecnologia di memoria integrata a semiconduttori. Il

primo passo consiste nel vedere come disporre su una singola piastrina di silicio celle da un bit per costruire il *componente integrato di memoria* (*memory chip*), che poi fa da base per costruire il *banco di memoria* (*memory file* o *bank*).

8.2.1 Componente di memoria

La cella elementare di memoria ha capacità da un bit e di solito le celle elementari sono disposte in configurazione di matrice bidimensionale, come mostra la Figura 8.2. Ciascuna riga della matrice contiene una parola di memoria e le celle da un bit che la costituiscono sono tutte collegate in fila mediante una linea apposita pilotata dal decodificatore di indirizzo, integrato nella stessa piastrina di silicio su cui si trova la matrice; la riga, e per estensione anche la rispettiva linea di collegamento, si chiama *linea di parola* (*word line*). Le celle di ciascuna colonna sono collegate al *circuito di lettura e scrittura* (*sense amplifier*) tramite due *linee di bit* (*bit line*); a loro volta i circuiti di lettura e scrittura sono collegati ai morsetti di dato dell'intero componente, che sono bidirezionali come del resto lo è il bus dati del calcolatore. In lettura il circuito rileva e amplifica il valore di tensione elettrica contenuto nella cella interessata e lo manda al morsetto corrispondente di dato su cui esce come bit, mentre in scrittura il circuito rileva il valore di tensione elettrica presente sul morsetto corrispondente di dato in ingresso e lo manda alla cella interessata dove viene memorizzato come bit.

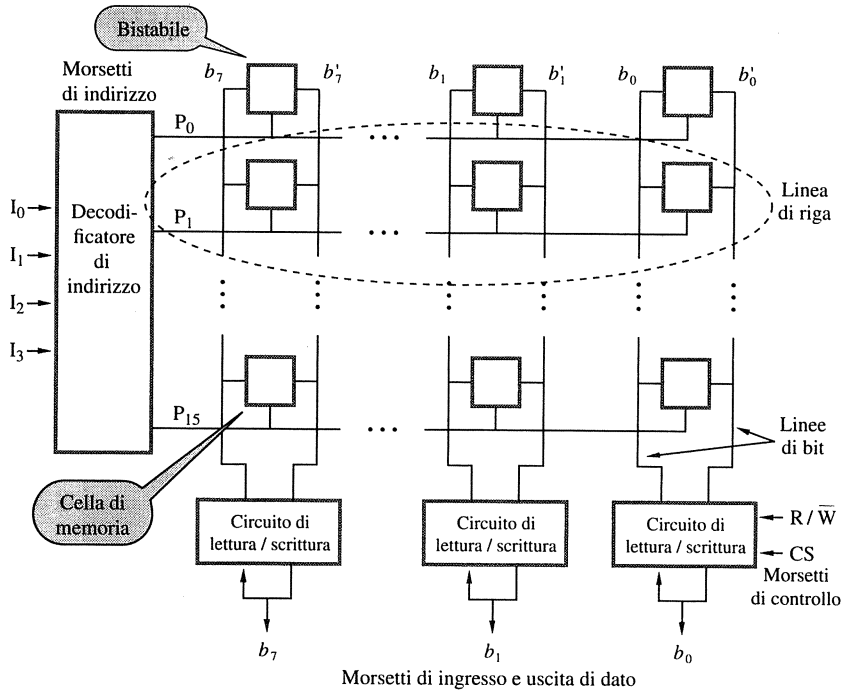


Figura 8.2 Componente di memoria 16 × 8 come matrice di celle da un bit.

La Figura 8.2 mostra un esempio di matrice di celle di capacità molto limitata: 16 parole da 8 bit ciascuna. Si esprime la capacità di tale memoria tramite l'indicazione 16 × 8, che si legge "sedici parole da otto bit" o, se si preferisce, "sedici byte"; tale notazione, molto usata, è più espressiva del valore complessivo 128 bit (= 16 × 8), che darebbe solo il numero totale di celle elementari senza però precisarne la configurazione matriciale, parametro di memoria significativo. Il morsetto (bidirezionale) di ingresso e uscita di dato del circuito di lettura e scrittura è collegato alla linea corrispondente di dato presente nel bus del calcolatore. In aggiunta il circuito ha due morsetti di ingresso con funzioni di controllo, R/$\overline{W}$ e *selezione del componente* o CS (*Chip Select*): il primo è collegato alla linea di controllo omonima nel bus del calcolatore e distingue tra le operazioni di lettura e scrittura, mentre il secondo serve per scegliere e attivare il componente integrato con cui lavorare all'interno del banco di memoria, che ne può contenere molti.

La capacità complessiva del componente di memoria di Figura 8.2 vale 128 bit e in totale esso ha 14 morsetti, cioè va collegato a 14 linee di bus esterne tra indirizzo, dato e controllo. Naturalmente gli occorrono anche almeno due morsetti di alimentazione, vale a dire tensione e massa, o anche più di due per distribuire meglio il carico elettrico; la memoria è un *dispositivo attivo*, cioè consuma energia, per quanto poca possa essere. Si consideri ora un componente di memoria di capacità un po' maggiore, per esempio con 1 Kbit in totale e configurazione matriciale 128 × 8, dunque con 19 morsetti in totale (7 morsetti di indirizzo + 8 morsetti di dato + 2 morsetti di controllo + 2 morsetti di alimentazione). In alternativa la configurazione matriciale potrebbe essere 1K × 1, con lunghezza di parola pari a un solo bit; in questo secondo caso occorrerebbero 10 morsetti di indirizzo ma solo uno di dato (bidirezionale), per un totale di solo 15 morsetti. Tuttavia si deve tenere presente che la configurazione, come è vista ai morsetti esterni, non coincide necessariamente con la struttura matriciale interna effettiva. Nella Figura 8.2 c'è coincidenza tra struttura interna e vista esterna del componente, ma dal punto di vista del programmatore la memoria, più che come matrice, è concepibile come vettore di parole, dove la lunghezza della parola spesso è ben inferiore al numero di parole, assai più grande.

La Figura 8.3 mostra la configurazione ai morsetti alternativa appena citata, dove però internamente l'indirizzo da 10 bit è suddiviso in due metà da 5 bit che costituiscono gli indirizzi di riga e colonna della matrice interna di celle, di tipo 32 × 32 (= 1K). L'indirizzo di riga specifica la linea di parola su cui lavorare tra le 32 esistenti, sui cui 32 bit costituenti si lavora in parallelo, mentre secondo l'indirizzo di colonna solo una delle 32 celle costituenti la linea di parola viene collegata al morsetto di dato esterno, che è unico, tramite il multiplatore di tipo a 5 ingressi di selezione (anch'esso bidirezionale come il morsetto di uscita cui è collegato). Sono possibili altre suddivisioni dell'indirizzo corrispondenti ad altrettante strutture matriciali interne, ferma restando la vista ai morsetti del tutto.

I componenti integrati di memoria disponibili sul mercato hanno capacità ben maggiore di quella dei due esempi illustrati nelle Figure 8.2 e 8.3, che hanno scopo didattico. Il componente di capacità grande non è strutturato diversamente da come si è visto qui, salvo contenere una matrice di celle di dimensione maggiore e avere morsetti esterni più numerosi. Per esempio, un chip da 1 Gigabit può avere un'organizzazione 256M × 4, nel qual caso è necessario un indirizzo a 28 bit e 4 bit vengono trasferiti da o verso il chip.

8.2.2 Memoria statica

La tecnologia di memoria capace di conservare il contenuto fintantoché è alimentata si chiama *memoria statica* (*static memory*). La caratteristica di perdere il contenuto (in se stessa sgradevole) quando l'alimentazione viene meno è anche chiamata *volatilità*. La Figura 8.4 mostra il circuito elettrico della cella di *memoria RAM statica* o SRAM (*Static RAM*). In sostanza

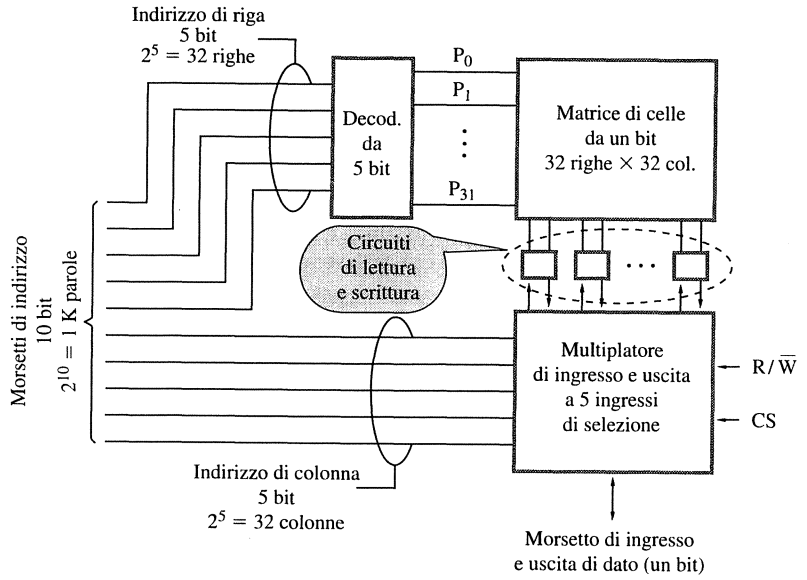


Figura 8.3 Componente di memoria 32 × 32 con capacità esterna 1 K × 8.

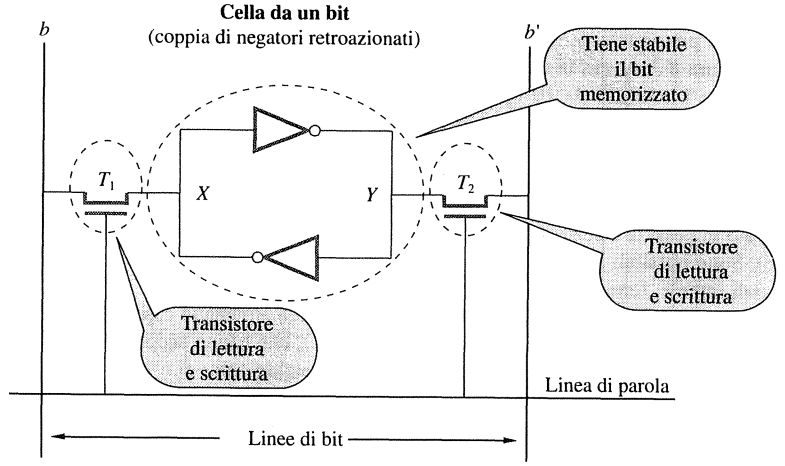


Figura 8.4 Cella elementare da un bit di memoria RAM statica.

si tratta di due negatori retroazionati, le cui uscite X e Y sono collegate alle due linee di bit tramite i transistori passanti T_1 e T_2, rispettivamente. Essi funzionano come interruttori e si possono aprire o chiudere pilotandoli mediante la linea di parola. Quando la linea di parola si trova a massa (vale a dire a valore logico 0) i due transistori T_1 e T_2 sono in interdizione

(cioè aperti), dunque la coppia di negatori è isolata e mantiene lo stato. Per esempio, si convenga che il bit memorizzato nella cella sia 1 quando l'uscita X ha valore logico 1 e di conseguenza Y ha valore 0 (e che sia 0 nel caso opposto, cioè quando X = 0 e Y = 1); fintantoché la linea di parola rimane a massa lo stato della cella non muta o, come si dice, è *stabile*.

Operazione di lettura

Per leggere lo stato della cella di memoria SRAM bisogna attivare la linea di parola portandola a 1; ciò mette in conduzione i due transistori T_1 e T_2 (cioè li chiude). Se il bit memorizzato nella cella è 1 le due linee di bit b e b' si portano allora a 1 e 0, rispettivamente; se il bit nella cella è 0 succede il contrario. Pertanto le due linee b e b' sono sempre complementari; il circuito di lettura e scrittura collegato a entrambe ne esamina lo stato di attivazione e pilota il morsetto di dato in uscita conformemente a tale stato.

Operazione di scrittura

Durante un'operazione Write, il circuito di lettura/scrittura pilota le linee di bit b e b', invece di rilevare il loro stato. Si pone il valore appropriato sulla linea di bit b e il suo complemento su b', attivando la linea di parola. Questo forza la cella nello stato corrispondente, che la cella mantiene quando la linea di parola viene disattivata.

Cella di memoria in tecnologia CMOS

La Figura 8.5 mostra lo schema circuitale dettagliato e il livello elettrico della cella di memoria di Figura 8.4, lì tratteggiata in modo ancora parzialmente di tipo logico. Le due coppie di transistori complementari T_3, T_5 e T_4, T_6 costituiscono i due negatori retroazionati (si veda l'Appendice), e il bit contenuto nella cella viene letto o scritto come si è appena spiegato. Per esempio, se il bit vale 1, la tensione al punto X ha livello elevato corrispondente al valore logico 1 e i due transistori T_3 e T_6 sono in conduzione (cioè sono chiusi), mentre i due transistori complementari T_5 e T_4 (cioè che funzionano in modo opposto ai primi due, rispettivamente) sono in interdizione (cioè sono aperti). Pertanto se i due transistori passanti T_1 e T_2

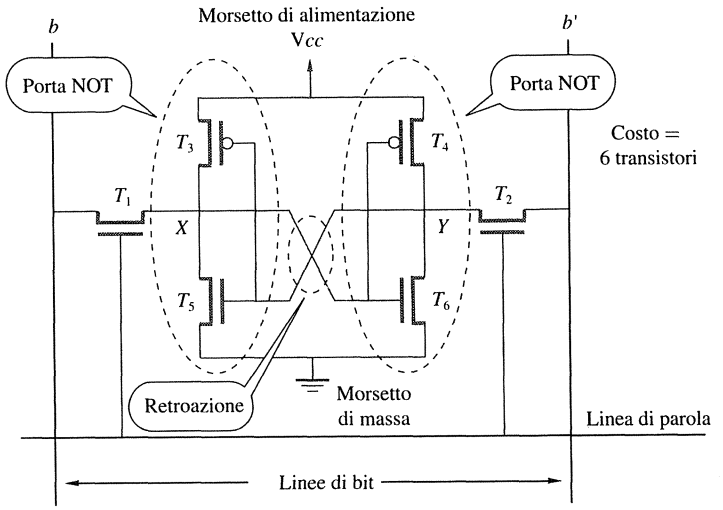


Figura 8.5 Cella elementare di memoria da un bit in tecnologia CMOS.

vengono messi in conduzione (cioè vengono chiusi) le due linee di bit b e b' assumono valore logico 1 e 0, rispettivamente.

Come si è accennato prima, la cella deve restare costantemente sotto alimentazione per mantenere lo stato; se l'alimentazione viene meno il bit nella cella va perso. Infatti, quand'anche la cella venisse rialimentata, il circuito si riposizionerebbe in uno stato stabile (in genere dopo avere oscillato per breve tempo), ma scelto in modo casuale e non necessariamente coincidente con quello di prima. Da ciò si vede appunto come la cella di memoria SRAM sia volatile perché il contenuto va perso se l'alimentazione viene meno.

Il vantaggio principale della cella di memoria SRAM è dato dal consumo di energia molto limitato: fluisce corrente elettrica solo quando si legge o scrive la cella; altrimenti i due transistori passanti T_1 e T_2, nonché un transistore in ciascuno dei due negatori, sono in interdizione, ed è facile vedere che allora non esiste continuità elettrica tra i morsetti di tensione e massa; ed è ben noto che se non passa corrente non ci può essere dissipazione di potenza.

L'intervallo di tempo di accesso della cella di memoria SRAM ha durata molto breve: i componenti di memoria integrati commerciali di tipo SRAM hanno tempo di accesso pari a pochi nanosecondi o anche meno. La memoria SRAM è usata soprattutto in applicazioni dove la velocità è un fattore critico.

8.2.3 Memoria dinamica asincrona

La memoria RAM statica è veloce, ma ha costo per bit elevato perché la cella richiede parecchi transistori.[2] Se si potesse ridurre il numero di transistori per bit si avrebbero tecnologie di memoria RAM decisamente meno costose. Ciò si può fare, ma al prezzo di tollerare che la cella non sia in grado di mantenere lo stato a tempo indefinito anche quando è correttamente alimentata; per tale motivo siffatta memoria si chiama *memoria dinamica* (*dynamic memory*), e quando è ad accesso casuale (come quasi sempre) *memoria RAM dinamica* o DRAM (*Dynamic RAM*).

Il bit contenuto nella cella di memoria dinamica è espresso sotto forma di quantità di carica elettrica depositata in un condensatore (elemento circuitale elettrico atto a trattenere o meno un pacchetto di carica più o meno elevata), ma la carica presente si dissipa spontaneamente entro qualche decina di millisecondi (da 10 a 100 ms). Siccome è ovvio che per scopi pratici la cella deve mantenere lo stato per intervalli di tempo ben più lunghi, bisogna rinnovare periodicamente la carica depositata nel condensatore o, come si dice, *rinfrescarne* (*refreshing*) il contenuto riportandolo al valore originale prima che si degradi tanto da non essere più riconoscibile. Questo avviene quando viene letto il contenuto della cella o quando vi viene scritta una nuova informazione.

La Figura 8.6 mostra un esempio di cella di memoria dinamica costituita da condensatore C e transistore T. Per scrivere il bit nella cella bisogna mettere T in conduzione (mediante la solita linea di parola come si faceva con la cella di memoria statica) e applicare un livello di tensione elettrica appropriato alla linea di bit, facendo così affluire alle armature del condensatore corrente elettrica più o meno intensa in funzione della quale si depone molta o poca carica per esprimere il valore logico del bit da memorizzare.

Dopo che il transistore T è stato posto in interdizione, il condensatore C comincia a scaricarsi a causa di effetti parassiti inevitabili per quanto modesti, che danno luogo a corrente di scarica pari a qualche picoAmpère (1 picoAmpère = 10^{-12} Ampère, un millesimo di miliardesimo di Ampère). Pertanto il valore logico del bit contenuto nella cella è leggibile senza errore solo quando viene rilevato prima che la carica del condensatore sia scesa sotto a una

[2] In tecnologia CMOS occorrono 6 transistori, come mostra la Figura 8.5, ed è un numero tipico.

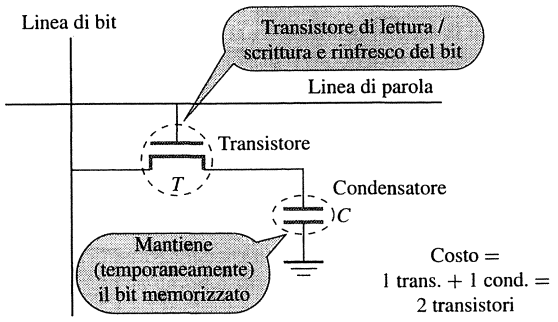


Figura 8.6 Cella elementare di memoria dinamica a un solo transistore.

soglia data. L'operazione di lettura del bit dà anche modo di rimediare al degrado prima che sia troppo tardi. Per leggere il contenuto della cella il transistore T viene messo in stato di conduzione e il circuito di lettura e scrittura collegato alla linea di bit rileva se la quantità di carica depositata nel condensatore C sia sopra la soglia. Se è così, il circuito pilota la linea di bit stessa alzandola al valore di tensione massimo, che rappresenta il valore logico 1 senza alcuna forma di degrado, pertanto il condensatore si ricarica completamente riacquistando la quantità di carica corretta. Se invece la carica rilevata è sotto la soglia, il circuito pilota la linea abbassandola a massa, il che ha l'effetto di scaricare completamente il condensatore rappresentando in modo esatto il valore logico 0. Si vede dunque come l'operazione di lettura del bit nella cella provveda anche automaticamente a rinfrescarne il valore. Tutte le celle appartenenti alla medesima riga vengono lette simultaneamente e dunque così si rinfresca lo stato dell'intera linea di parola. L'operazione di scrittura della cella si svolge in modo simile. Illustrare la struttura interna del circuito di lettura e scrittura non rientra negli scopi del libro e pertanto si rimanda alla letteratura scientifica.

La Figura 8.7 mostra il componente di memoria DRAM da 256 Mbit, in configurazione esterna 32M × 8. Internamente le celle sono disposte in matrice da 16K × 16K. Ciascuna riga contiene dunque 16.384 bit (= 16K) ed è suddivisa in 2.048 gruppi da 8 bit, ovvero 2.048 byte. Occorrono dunque 14 bit di indirizzo per scegliere la riga dove lavorare e altri 11 bit per scegliere il byte tra i 2.048 della riga. L'indirizzo complessivo è pertanto da 25 bit, e i 14 e 11 bit più e meno significativi costituiscono gli indirizzi di riga e colonna (o byte), rispettivamente. Onde ridurre il numero di morsetti esterni necessari per collegare al bus il componente, i due indirizzi di riga e colonna vengono multiplati su 14 morsetti solamente. Iniziando l'operazione di lettura o scrittura si manda per primo l'indirizzo di riga, che viene scritto nel registro omonimo in risposta all'attivazione della linea di *campionamento di indirizzo di riga* o RAS (*Row Address Strobe*). Poi l'operazione prosegue e le celle della riga indirizzata vengono lette e rinfrescate. Poco dopo l'indirizzo di riga si manda ai morsetti quello di colonna e lo si scrive nel registro omonimo sotto il controllo della linea di *campionamento di indirizzo di colonna* o CAS (*Column Address Strobe*). Il contenuto del registro viene quindi decodificato, così scegliendo il gruppo di 8 circuiti di lettura e scrittura da attivare. Se la linea di controllo R/$\overline{W}$ del bus ordina l'operazione di lettura, i valori in uscita ai circuiti vengono emessi sulle linee di dato del bus, D_{7-0}. Se invece ordina la scrittura, il byte presente su D_{7-0} viene preso e mandato ai circuiti attivati, da dove viene poi mandato alle 8 celle del byte in cui vanno memorizzati. Nei componenti di memoria commerciali i due segnali di controllo

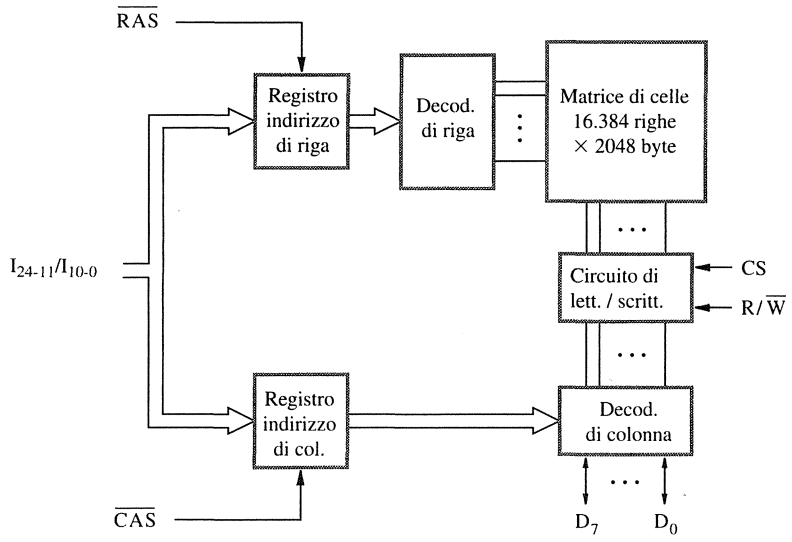


Figura 8.7 Componente di memoria dinamica con capacità da 2M × 8.

RAS e CAS sono attivi bassi, cioè l'indirizzo viene scritto nel registro sul fronte di discesa del segnale, pertanto negli schemi vengono abitualmente denotati soprallineati (ma nel testo si citano in forma naturale).

La sincronizzazione del funzionamento della DRAM descritta sopra è controllata dai segnali RAS e CAS. Questi segnali sono generati da un circuito di controllo della memoria esterno al chip quando il processore emette un comando Read o Write. Durante un'operazione Read, i dati di uscita vengono trasferiti al processore dopo un ritardo equivalente al tempo di accesso alla memoria. Tali memorie sono dette *DRAM asincrone*. Il controllore della memoria è anche responsabile per il rinfresco dei dati immagazzinati nei chip della memoria, come sarà descritto successivamente.

Modo veloce di pagina

Quando si legge il componente di memoria DRAM di Figura 8.7 si rilevano i bit di tutte le 16.384 celle della riga, ma di questi solo gli 8 indirizzati vengono poi emessi sulle linee di dato D_{7-0} del bus del calcolatore. Il byte di interesse è specificato dai bit I_{10-0} delle linee di indirizzo del bus. Con una modifica semplice al circuito è possibile leggere anche gli altri byte nella stessa riga senza bisogno di riattivarla di nuovo. Basta aggiungere un bit all'uscita di linea di bit di ciascun circuito di lettura e scrittura, e quando si manda l'indirizzo di riga si devono attivare i bit corrispondenti ai bit della riga selezionata. Per leggere i vari byte nella riga ed emetterli sul bus dati basta poi applicare differenti indirizzi di colonna, purché si resti dentro la riga corrente. L'operazione di scrittura di byte consecutivi funziona in modo analogo.

Lo schema di distribuzione di indirizzi alla memoria che più conviene gestire in modo efficiente è quello di sequenza di indirizzi consecutivi, e si ottiene mandando gli indirizzi di colonna in rapida successione sotto il controllo di un treno di impulsi CAS. In tale modo si può trasferire un intero blocco di dati molto più rapidamente di quanto non si riesca a ottenere

avendo una sequenza di indirizzi casuale. La funzione di trasferimento di blocco viene chiamata *modo veloce di pagina* (*fast page mode*).[3]

In precedenza era stato indicato che la grande maggioranza di transazioni nella memoria principale riguarda il trasferimento di blocchi. La maggiore rapidità ottenibile nel modo di pagina veloce rende le RAM dinamiche particolarmente adatte a questo ambiente.

8.2.4 Memoria dinamica sincrona

Nei primi anni '90 sono stati prodotti componenti di memoria DRAM le cui operazioni sono scandite da segnale di ciclo; tale tecnologia si chiama *memoria DRAM sincrona* o SDRAM (*synchronous DRAM*). La Figura 8.8 mostra lo schema circuitale di un componente di memoria SDRAM: la matrice di celle da un bit è strutturata come quella della DRAM, e indirizzo e dato transitano attraverso registri buffer. La caratteristica distintiva di una SDRAM è l'uso di un segnale di clock, la cui disponibilità rende possibile incorporare un circuito di controllo sul chip che fornisce tante caratteristiche utili. Per esempio, le SDRAM incorporano un circuito di rinfresco, con un contatore di rinfresco per fornire gli indirizzi delle righe da selezionare per il rinfresco. Di conseguenza, la natura dinamica di queste memorie è quasi invisibile per l'utente. Si noti in particolare che i circuiti di lettura e scrittura contengono registri

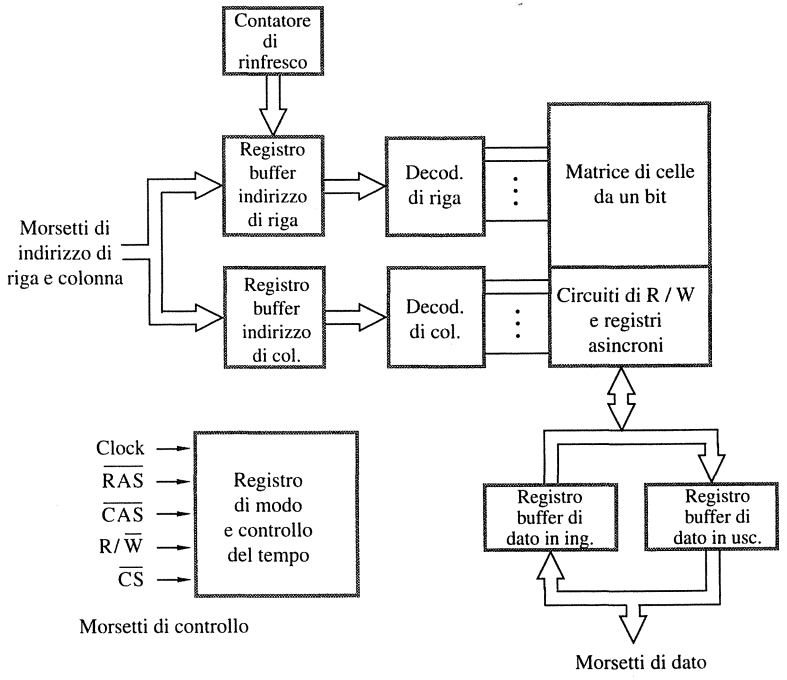


Figura 8.8 Componente di memoria RAM dinamica sincrona.

[3] Comunemente si chiama blocco un gruppo piccolo di dati, pagina uno più grande; ma la vera distinzione tra blocco e pagina sta nel modo in cui si gestiscono, non in quanto sono grandi.

asincroni collegati alle linee di bit che si diramano nella matrice di celle. Se si deve svolgere l'operazione di lettura i bit contenuti nelle celle della riga selezionata vengono mandati a tali registri asincroni e ivi scritti, ma se la lettura ha solamente il fine di rinfrescare il contenuto della riga, e non è dunque richiesta dal processore via bus bensì è generata dal circuito interno di rinfresco, il contenuto dei registri non viene modificato. I dati contenuti nei registri asincroni corrispondenti alla colonna selezionata vengono mandati ai registri buffer di uscita e da lì ai morsetti di uscita di dato del componente (bidirezionali, sono anche di ingresso di dato). I registri buffer sono utili per trasferire grandi blocchi di dati ad altissima velocità. Isolando le connessioni esterne dai circuiti interni del chip, diventa possibile iniziare una nuova operazione di accesso mentre i dati vengono trasferiti da o verso i registri.

La memoria SDRAM può funzionare in svariati modi differenti selezionabili programmando opportunamente il registro di modo contenuto nel componente; per esempio si può programmare l'operazione di trasferimento di blocco specificandone anche la lunghezza, e il componente reagisce svolgendola secondo la tecnica di modo veloce di pagina vista prima. Per selezionare colonne in successione non occorre fornire dall'esterno alla memoria SDRAM il treno di impulsi sulla linea di controllo CAS, perché tutti i segnali di controllo necessari vengono generati internamente al componente tramite i contatori e il segnale di clock. Si può mandare alla memoria un dato per ogni ciclo di clock e tutte le azioni del componente partono in corrispondenza del fronte di salita del segnale di clock, che delimita la transizione tra cicli consecutivi.

La Figura 8.9 mostra il diagramma temporale tipico per l'operazione di lettura di blocco di lunghezza quattro. All'inizio l'indirizzo di riga viene campionato e scritto nel registro buffer relativo sotto il controllo del segnale RAS. La memoria consuma di solito cinque o sei cicli di clock per selezionare la riga e attivarla (in Figura 8.9 i cicli consumati sono due). Poi l'indirizzo di colonna viene campionato e scritto nel registro buffer relativo sotto il controllo del segnale CAS, e con ritardo di un ciclo di clock la prima parola del blocco (D0) viene mandata ai morsetti di uscita di dato del componente. Il componente incrementa in modo automatico l'indirizzo di colonna per leggere le altre tre parole del blocco (D1, D2 e D3) e le manda ai morsetti di uscita nei tre cicli di clock successivi.

Le DRAM sincrone possono inviare dati a velocità molto alta, poiché tutti i segnali di controllo necessari sono generati all'interno del chip. Le prime SDRAM commerciali degli anni '90 erano progettate per frequenze di clock fino a 133 MHz. Con l'evoluzione della tecnologia, sono stati sviluppati chip SDRAM molto più veloci. Le SDRAM di oggi funzionano a frequenze di clock che possono superare 1 GHz.

Tempo di latenza e banda passante

Il trasferimento di dati tra processore e memoria comprende spesso blocchi di parole. Velocità ed efficienza di tali operazioni di trasferimento influiscono significativamente sulla prestazione complessiva del calcolatore. Due cifre di merito importanti per quantificare la prestazione del sistema di memoria sono *intervallo di tempo di latenza* (*time latency*) e *banda passante* (*bandwidth*), in breve *latenza* e *banda*. Il parametro di latenza esprime il tempo necessario per trasferire una parola di memoria. Quando l'operazione di lettura o scrittura coinvolge una sola parola la latenza quantifica in modo completo la prestazione della memoria. Ma se si deve leggere o scrivere un blocco di parole l'intervallo di tempo totale necessario per trasferirlo tutto dipende anche dalla frequenza di trasferimento delle parole successive alla prima e dalla dimensione del blocco. Per il blocco la latenza esprime solo il tempo necessario a trasferire la prima parola e di solito tale intervallo di tempo iniziale ha durata molto maggiore di quanto non duri il trasferimento di ciascuna delle parole successive. Per esempio, in riferimento al diagramma temporale mostrato in Figura 8.9, l'operazione di trasferimento inizia con l'attivazione del segnale RAS, ma la prima parola del blocco viene trasferita solo cinque cicli di

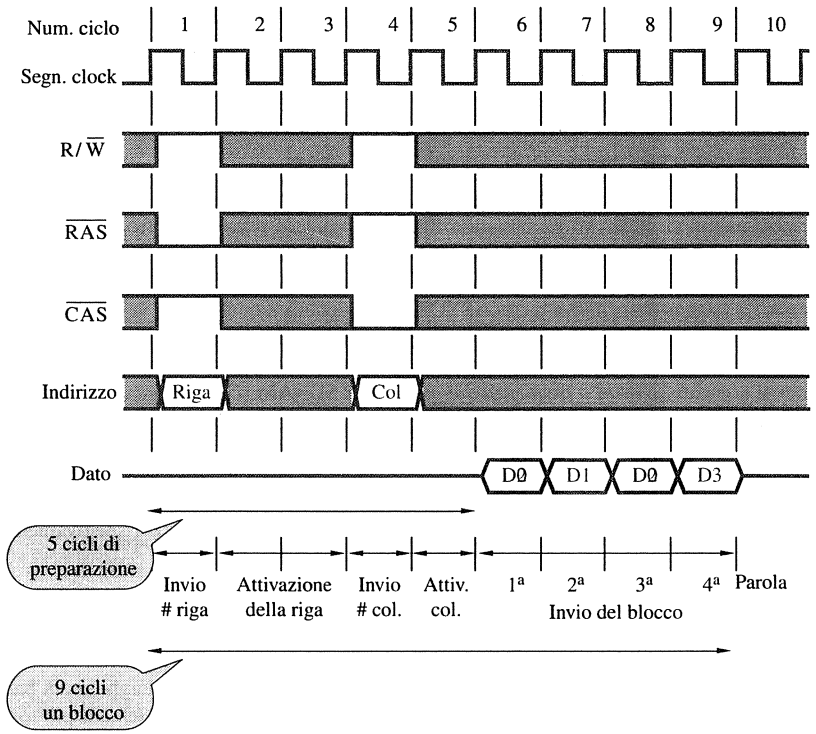


Figura 8.9 Sequenza di lettura di blocco di lunghezza quattro in memoria RAM dinamica sincrona.

clock dopo, dunque la latenza vale altrettanto; tuttavia le altre tre parole vengono trasferite a seguito della prima in tre cicli esatti.

È pertanto importante riuscire a calcolare il tempo di trasferimento del blocco di parole completo. Siccome la dimensione del blocco non è fissa, conviene misurare tale cifra di merito come numero di byte o bit trasferibili nell'unità di tempo, per esempio un secondo. Il parametro che ne deriva si chiama banda passante (o semplicemente banda) e dimensionalmente è analogo alla frequenza, giacché è il reciproco del tempo: # bit $\times s^{-1}$ oppure # byte $\times s^{-1}$; si misura, per esempio, in Mbit/s oppure Mbyte/s (Megabit o Megabyte al secondo), e simili. La banda dell'unità di memoria, che può essere costituita da uno o più componenti integrati, dipende dalla velocità con cui si può accedere al contenuto e dal numero di bit trasferibili in parallelo. Tuttavia la banda passante complessiva del calcolatore, determinata dal flusso di istruzioni e dati tra processore e memoria, non dipende solo dalla velocità di quest'ultima, ma anche dalla capacità di trasferimento del sistema di collegamento tra processore e memoria, vale a dire il bus. Come si è già accennato, i componenti integrati di memoria di largo consumo sono progettati in modo da adattarsi senza alcuno sforzo ai bus conformi agli standard di diffusione maggiore. Per quanto attiene la capacità di trasferimento del bus, essa dipende dal tempo consumato per imporre il valore logico sulla linea, dal tempo che il bit impiega per

propagarsi lungo la linea e dal numero di bit trasmissibili in parallelo, vale a dire il numero di linee presenti nel bus. Pertanto la banda passante del bus risulta dal prodotto della frequenza di trasferimento di dato (e anche di accesso al dato) e la dimensione del bus stesso, cioè il numero di linee (si ricordi che in sostanza la banda ha la stessa unità di misura della frequenza). Per questo motivo, le interconnessioni usate assicurano sempre che la banda disponibile per i trasferimenti di dati fra processore e memoria sia molto alta.

Memoria SDRAM veloce

La corsa continua all'aumento di prestazione ha condotto allo sviluppo di versioni ancora più veloci di memoria SDRAM. La memoria SDRAM standard si attiva sul fronte di salita del segnale di clock. La versione potenziata di SDRAM gestisce la matrice di celle come fa quella normale, ma è capace di trasferire il dato su entrambi i fronti del segnale di clock, salita e discesa. La latenza della nuova memoria SDRAM è identica a quella della versione standard, ma siccome i dati vengono trasferiti su due fronti invece di uno solo la banda passante è raddoppiata, in particolare quando si deve trasferire un blocco di parole. La nuova tecnologia prende il nome di *SDRAM a frequenza doppia* o DDR SDRAM (*Double Data Rate SDRAM*).

Sono state sviluppate varie versioni di chip DDR. La versione iniziale è nota come DDR. Successive versioni, dette DDR2, DDR3 e DDR4, hanno caratteristiche migliori. Offrono maggiore capacità di memoria, minore potenza e velocità di clock più alte. Per esempio, DDR2 e DDR3 possono funzionare a frequenze di clock di 400 e 800 MHz, rispettivamente. Quindi trasferiscono dati usando le velocità di clock effettive di 800 e 1600 MHz, rispettivamente.

Memoria Rambus

La frequenza di trasferimento di dati fra memoria e processore è una funzione sia della larghezza di banda della memoria sia di quella della sua connessione al processore. Rambus è una tecnologia di memoria che ottiene alte frequenze di trasferimento di dati per mezzo di un'interfaccia ad alta velocità fra la memoria e il processore. Un modo per accrescere la larghezza di banda di questa connessione è usare un percorso di dati più ampio. Tuttavia questo richiede più spazio e più piedini, facendo così crescere il costo del sistema. L'alternativa è di usare un numero minore di linee con una velocità di clock più alta. Questo è l'approccio seguito da Rambus.

La caratteristica predominante della tecnologia Rambus è l'uso di una tecnica di segnalazione differenziale per trasferire i dati da e verso i chip di memoria. L'idea di base della segnalazione differenziale è descritta nel Paragrafo 7.5.1, fra le caratteristiche elettriche USB. Nella tecnologia Rambus, i segnali sono trasmessi usando piccole variazioni di voltaggio di 0,1 V sopra e sotto un valore di riferimento. Sono state sviluppate varie versioni di questo standard, con velocità di clock fino a 800 MHz e frequenze di trasferimento di dati di vari Gigabyte al secondo.

La tecnologia Rambus è in diretta competizione con la tecnologia DDR SDRAM. Ciascuna ha certi vantaggi e svantaggi. Una considerazione non tecnica è che la specifica di DDR SDRAM è uno standard aperto che può essere usato gratuitamente. Rambus invece è uno schema proprietario che deve essere usato con licenza dai costruttori del chip.

8.2.5 Banco di memoria

Finora si è esaminata la struttura interna del componente integrato di memoria isolato, ma occorre anche vedere come si possano combinare e collegare più componenti per costruire unità di memoria di capacità superiore.

Banco di memoria statica

Si consideri come esempio l'unità di memoria di capacità 2M parole da 32 bit ciascuna (in totale 2.097.152 parole). La Figura 8.10 mostra come costruire tale unità di memoria tramite componenti integrati di memoria statica da 512K × 8, che va sotto il nome di *banco di*

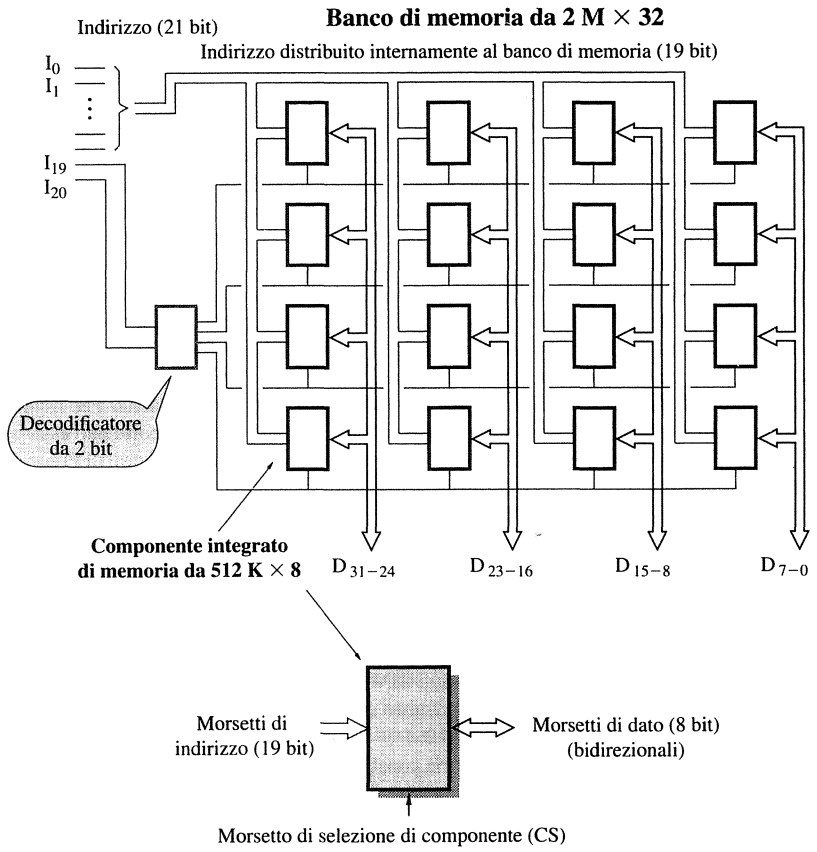


Figura 8.10 Struttura di banco di memoria con capacità da 2M × 32 costruito tramite componenti integrati di memoria statica con capacità da 512 K × 8.

memoria (*memory bank* o *memory file*). In sostanza il banco di memoria è strutturato come matrice (bidimensionale) di componenti; la colonna della matrice contiene quattro componenti del tipo indicato e corrisponde a uno dei quattro byte che formano la parola da 32 bit. Quattro colonne affiancate costituiscono pertanto la matrice 2M × 32 completa. Ciascuno dei componenti il banco è dotato di ingresso di *selezione di componente* o CS (*Chip Select*) (attivo alto). Quando il morsetto CS è posto a 1, il componente è attivo, accetta comandi, indirizzi e scambia dati con l'esterno; altrimenti (cioè se CS vale 0) i morsetti di dato del componente sono posti in stato di alta impedenza (si veda l'Appendice) e dunque il componente è elettricamente scollegato dai circuiti circostanti, limitandosi a mantenere il contenuto ma rifiutando di interagire con l'esterno. Per indirizzare una parola nel totale di 2M parole contenute nel banco occorrono 21 bit: i due più significativi vengono decodificati per individuare la riga di componenti da attivare, pertanto ciascuna linea in uscita al decodificatore va a pilotare il morsetto CS dei quattro componenti nella riga corrispondente, mentre i 19 meno significativi sono usati per indirizzare la parola all'interno di ciascuno dei quattro componenti appartenenti

alla riga selezionata. Il comando di bus R/$\overline{W}$ viene distribuito a tutti i componenti il banco giacché in ogni caso solo quelli nella riga selezionata lo prendono in considerazione, essendo tutte le altre righe senz'altro inattive (in Figura 8.10 la rete di distribuzione del comando R/$\overline{W}$ non è mostrata).

Banco di memoria dinamica

La maggior parte dei calcolatori attuali è dotata di memoria di capacità piuttosto elevata; per esempio, perfino un calcolatore da tavolo di qualità modesta potrebbe benissimo contenere almeno 1 Gbyte di memoria RAM, mentre la stazione di lavoro parte in genere da una memoria RAM da 4 Gbyte. Aumentare la capacità di memoria migliora la prestazione del calcolatore perché si possono tenere in memoria più programmi e dati, riducendo così la frequenza con cui si scambiano informazioni tra memoria centrale e di massa. Grazie alla loro alta densità di bit e basso costo, le RAM dinamiche, principalmente del tipo sincrono, sono ampiamente usate nelle unità di memoria dei calcolatori. Esse sono più lente delle RAM statiche, tuttavia usano meno potenza e hanno un costo per bit notevolmente più basso. I chip disponibili hanno capacità fino a 2 Gigabit, e sono in via di sviluppo persino chip di maggiore capacità. Per ridurre il numero di chip di memoria necessari in un dato calcolatore, un chip di memoria può essere organizzato per leggere o scrivere un certo numero di bit in parallelo, come nel caso di Figura 8.7. Si fabbricano chip in diverse organizzazioni, per offrire flessibilità nel progetto di sistemi di memoria. Per esempio, un chip da 1 Gigabit può essere organizzato come 256M × 4 o 128M × 8.

Considerazioni di confezione hanno portato all'*unità di memoria modulare in linea* o SIMM (*Single In line Memory Module*) e a quella *modulare in linea doppia* o DIMM (*Double In line Memory Module*). In entrambi i casi si tratta di una schiera di componenti integrati di memoria disposti su una schedina a circuito stampato di taglia limitata, che si può installare verticalmente sulla scheda madre del calcolatore inserendola in uno dei connettori appositi disposti in file parallele e ravvicinate. Moduli SIMM o DIMM di taglia differente si possono adattare (entro certi limiti) allo stesso connettore sulla scheda madre. Per esempio, i moduli DIMM da 128M × 64, 256M × 64, e 512M × 64 bit usano tutti lo stesso attacco a 240 piedini. Tale tipología di modulo di memoria consuma pochissimo spazio sulla scheda madre e permette facilmente di espandere la memoria aggiungendo nuovi moduli o sostituendo quelli già installati con altri di capacità maggiore.

Controllore di memoria

Onde ridurre il numero di morsetti, spesso l'indirizzo in ingresso al componente integrato di memoria dinamica è multiplato. In sostanza l'indirizzo è suddiviso in due parti, come spiegato prima: la metà più significativa, che serve per selezionare la riga attiva della matrice di celle, viene mandata per prima, campionata e scritta nel registro buffer apposito all'interno del componente, il tutto sotto il controllo del segnale RAS. Poi, riusando gli stessi ingressi di indirizzo, vengono mandati i bit meno significativi di indirizzo, che servono per selezionare la colonna attiva, e sotto il controllo del segnale CAS vengono anch'essi campionati e scritti nel registro buffer riservato per tale uso.

Il processore tipico manda sul bus l'indirizzo di memoria completo, cioè tutti i bit in parallelo. La multiplazione dell'indirizzo viene effettuata dal circuito *controllore di memoria* (*memory controller*), che è interposto tra processore e memoria. Quando vede attivarsi il segnale di richiesta di operazione di memoria il controllore acquisisce il comando R/$\overline{W}$ e l'indirizzo completo emessi dal processore, poi lo suddivide in indirizzo di riga e colonna, manda i due semiindirizzi all'unità di memoria nell'ordine appropriato e pilota anche i segnali di controllo RAS e CAS. Dunque complessivamente il controllore effettua la multiplazione di indirizzo, attiva i segnali RAS e CAS con la tempistica corretta, pilota il segnale CS di selezione del componente e gli inoltra anche il comando R/$\overline{W}$. Le linee di dato del bus collegano

direttamente processore e memoria senza transitare per il controllore. Si tenga presente che se il componente di memoria è in tecnologia SDRAM, gli occorre anche il segnale di clock.

Quando il controllore è usato per gestire componenti di memoria DRAM privi di circuito di rinfresco interno, deve prendersi in carico anche tale funzione e pertanto necessita di tutte le informazioni relative. In tale caso il controllore contiene anche il contatore di rinfresco, usato per generare in successione gli indirizzi di riga, e ha il compito di effettuare periodicamente il rinfresco del contenuto di ogni riga entro il tempo massimo prescritto dalla tecnologia di memoria.

Costo del rinfresco di memoria

Una RAM dinamica non può rispondere a richieste di lettura o scrittura mentre ha luogo un'operazione di rinfresco interno. Tali richieste vengono ritardate fino a che il ciclo di rinfresco non è concluso. Tuttavia, il tempo perso per dare luogo alle operazioni di rinfresco è molto piccolo. Per esempio, si consideri una SDRAM in cui ciascuna riga necessiti di essere rinfrescata un volta ogni 64 ms. Si supponga che il tempo minimo fra due accessi alla riga sia di 50 ns e che le operazioni di rinfresco vengano sistemate in modo che tutte le righe del chip siano rinfrescate in 8K (8,192) cicli di rinfresco. Occorrono quindi 8,192 × 0,050 = 0,41 ms per rinfrescare tutte le righe. Il costo del rinfresco è 0,41/64 = 0,0064, che è meno dell'1% del tempo totale disponibile per accedere alla memoria.

Scelta della tecnologia

La scelta di un chip di RAM per una certa applicazione dipende da diversi fattori. In primo luogo fra questi vi sono costo, velocità, dissipazione di potenza e dimensione del chip.

Le RAM statiche sono caratterizzate dal loro funzionamento molto veloce. Tuttavia, il loro costo e la densità di bit sono penalizzati dalla complessità del circuito che realizza la cella di base. Esse sono principalmente usate quando è necessaria una memoria piccola ma molto veloce. Le RAM dinamiche, d'altra parte, hanno un'alta densità di bit e un basso costo per bit. Le DRAM sincrone sono la scelta predominante per realizzare la memoria principale.

8.3 Memoria a sola lettura

Entrambe le tecnologie di memoria DRAM e SDRAM sono di tipo *volatile*, cioè il contenuto va perso se l'alimentazione del componente integrato di memoria viene meno. Ma vi sono numerose applicazioni dove la memoria deve mantenere il contenuto informativo anche quando non è alimentata. Per esempio, il Capitolo 4 descrive la necessità di immagazzinare un piccolo programma in tale memoria, per essere usato per iniziare il processo di avvio del caricamento del sistema operativo da un disco fisso alla memoria principale. Le applicazioni embedded descritte nel Capitolo 10 sono un altro esempio importante. Molte applicazioni embedded non usano un disco fisso e richiedono memorie non volatili per immagazzinare il loro software.

Nel corso degli anni sono state ideate e messe a punto numerose tecnologie di memoria elettronica persistente. In lettura la memoria persistente funziona in modo del tutto analogo alla memoria DRAM o SDRAM, mentre l'operazione di scrittura ha carattere speciale e richiede una procedura apposita.

8.3.1 Memoria ROM

Una memoria è chiamata *memoria a sola lettura* o ROM (*Read-Only Memory*) quando l'informazione può essere scritta in essa una sola volta al momento della produzione. La Figura 8.11 mostra la struttura circuitale della cella di memoria ROM da un bit. Se l'interruttore

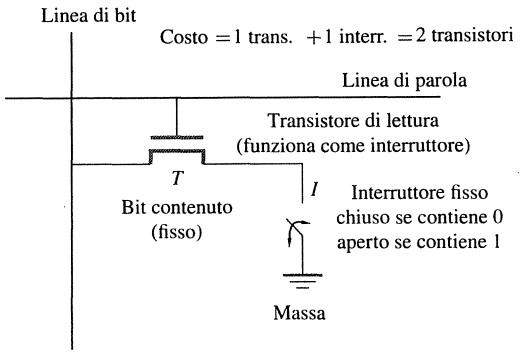


Figura 8.11 Cella elementare di memoria ROM da un bit.

I è chiuso, la cella contiene valore logico 0, altrimenti contiene 1. L'interruttore I è fisso, cioè il suo stato (chiuso o aperto) viene deciso in sede di fabbricazione del componente di memoria e in seguito non si può più cambiare. Tramite resistenza elevatrice (che nella Figura 8.11 non è mostrata) la linea di bit è collegata ad alimentazione. Il transistore T funziona come interruttore apribile e chiudibile; quando la cella non è in uso, la linea di parola è tenuta a massa e il transistore T è aperto, pertanto l'interruttore I risulta isolato.

Per leggere il bit contenuto nella cella si deve attivare la linea di parola portandola a livello di tensione di alimentazione, di conseguenza il transistore T viene chiuso e il livello di tensione sulla linea di bit scende a massa (dunque a 0) oppure resta a tensione di alimentazione (dunque a 1) secondo che l'interruttore I sia chiuso oppure aperto, rispettivamente. La linea di bit è collegata al *circuito di lettura e scrittura* (*sense amplifier*) che ne rileva il livello di attivazione e lo manda sul morsetto di uscita di dato del componente contenente la cella. Lo stato della connessione a terra in ciascuna cella viene determinato quando il chip è prodotto, usando una maschera con un modello che rappresenta l'informazione da immagazzinare.

8.3.2 Memoria PROM

Esiste tutta una gamma di varianti tecnologiche di memoria elettronica dove è possibile scrivere in tutto o in parte il contenuto del componente anche dopo la fabbricazione, una o più volte secondo la tecnologia specifica. Tuttavia la sostituzione più o meno completa del contenuto è un'operazione straordinaria, cioè va fatta con mezzi speciali e consuma tempo. In sostanza, tali varianti tecnologiche sono una via di mezzo tra la memoria a sola lettura pura e semplice, la ROM, dove la scrittura è impossibile, e la memoria a lettura e scrittura effettiva, la RAM, dove l'operazione di scrittura costa circa come quella di lettura, lavora anch'essa sulla singola parola e consuma lo stesso tempo o appena poco di più. La funzione di sostituzione del contenuto va sotto nome di *programmabilità sul campo di lavoro* (*field programmability*) e, come si vedrà nel seguito, ammette molte forme che si differenziano secondo il fenomeno fisico sfruttato. La memoria così fatta si chiama genericamente *memoria a sola lettura programmabile* o PROM (*Programmable Read Only Memory*), ma all'acronimo si aggiunge spesso un prefisso per precisarne meglio le caratteristiche tecnologiche.

Nella forma più semplice, la cella di memoria PROM è fatta come quella di memoria ROM mostrata in Figura 8.11, dove però l'interruttore fisso I è sostituito da un elemento fusibile (in sostanza, ancora un interruttore). Subito dopo la fabbricazione, il componente di memoria PROM è *vergine*, ovvero le celle contengono tutte il bit 0, vale a dire i fusibili sono chiusi. Si può a scelta inserire il bit 1 nella cella bruciando il fusibile (ciò che equivale ad aprirlo) tramite un impulso di corrente elettrica breve ma intenso. L'operazione richiede un dispositivo programmatore apposito, capace di generare e immettere l'impulso appropriato. Eseguendo tale operazione su gruppi di celle si programma il contenuto effettivo della memoria. Chiaramente l'operazione di programmazione è irreversibile perché il fusibile bruciato non è reintegrabile.

La memoria PROM è più flessibile della ROM. La memoria ROM è economica in produzione di massa per applicazioni dove occorre tenere permanentemente in memoria programmi e dati. Ma se il volume di produzione di componenti ROM è limitato, il costo di preparazione del processo di fabbricazione (che comporta il disegno delle maschere di fotoimpressione specifiche per il contenuto da preimpostare nella memoria) li rende piuttosto costosi e inoltre fabbricarli da zero richiede tempo. In tale caso, la memoria PROM fornisce una soluzione nettamente meno costosa e quasi pronta all'uso, bisogna solo programmarla sul campo di lavoro.

8.3.3 Memoria EPROM

Una variante tecnologica di memoria PROM è nota come *memoria a sola lettura programmabile e cancellabile* o EPROM (*Erasable Programmable Read Only Memory*). Essa funziona come la memoria PROM, ma il contenuto è cancellabile e sostituibile una o più volte. Simile facoltà la rende piuttosto flessibile e utile specialmente nello sviluppo di prototipi. Dato che la memoria EPROM può mantenere stabile il contenuto per tempi piuttosto lunghi (almeno alcuni anni), essa risulta valida sostituta della memoria ROM quando si prevede di dovere fare cambiamenti, vale a dire in fase di progetto e sviluppo.

La cella di memoria EPROM ha struttura circuitale simile a quella di memoria ROM mostrata in Figura 8.11. L'interruttore I è però sostituito da un transistore speciale: normalmente esso è fisso in stato o chiuso o aperto, ma lo si può riprogrammare cambiandone lo stato. Per programmare lo stato di interruttore aperto si deve iniettare nel transistore un pacchetto di carica elettrica che resta lì intrappolato; per tornare allo stato di chiuso si elimina la carica intrappolata. L'iniezione del pacchetto di carica è ottenuta tramite impulso di corrente elettrica. Per il resto la cella di memoria EPROM funziona come quella di memoria ROM.

Come si è detto all'inizio, la novità della memoria EPROM rispetto alla memoria PROM consiste nel poterne riscrivere il contenuto un certo numero di volte. Cancellare il contenuto del componente significa riportarlo nello stato vergine ovvero disperdere la carica intrappolata in ogni transistore. La tecnica più semplice per farlo consiste nell'esporre la piastrina di silicio alla radiazione ionizzante (nella fattispecie ultravioletta) per un intervallo di tempo prestabilito. Per consentire tale operazione, la piastrina di silicio contenuta nel package (involucro) del componente EPROM è protetta mediante finestrella trasparente. Per riprogrammare il componente EPROM lo si estrae dalla sede di lavoro e lo si espone alla radiazione (tramite lampada UV, ma non quella per abbronzatura bensì una apposita), poi lo si inserisce nella macchina programmatrice che genera gli impulsi di corrente e infine, se tutto è andato bene, lo si reinstalla di nuovo nella sede di lavoro.

8.3.4 Memoria EEPROM

Il limite principale della memoria EPROM è dato dalla necessità di estrarre il componente dalla sede di lavoro per riprogrammarlo e dal fatto che ogni volta bisogna riscriverne il contenuto intero, anche se si desidera cambiarne un solo bit. Si può perfezionare la tecnologia

EPROM ottenendo memoria PROM sia programmabile sia cancellabile (cioè riportabile nello stato vergine) in modo puramente elettrico. Essa va sotto nome di *memoria a sola lettura programmabile e cancellabile elettricamente* o EEPROM (*Electrically Erasable Programmable Read Only Memory*). Il componente EEPROM non va estratto dalla sede di lavoro, bensì lo si può riprogrammare in modo elettrico tramite morsetti appositi, inoltre se ne può cancellare selettivamente la singola cella. Il solo limite è dato dal fatto che per cancellare, programmare e leggere occorre fare uso di livelli di tensione elettrica differenti, ciò che rende un po' complesso pilotare il componente. Tuttavia, questo svantaggio è superato dai molti vantaggi delle EEPROM. In pratica, esse hanno rimpiazzato le EPROM.

8.3.5 Memoria flash

In tempi più recenti la tecnologia di memoria EEPROM ha dato origine alla tecnologia di *memoria flash* (*flash memory*).[4]

La cella di memoria flash si basa su un transistore controllato da carica intrappolata, proprio come la cella di memoria EEPROM. Sebbene ci siano alcune somiglianze, ci sono anche differenze significative. Sia nella memoria EEPROM sia in quella flash si può leggere la singola cella, ma nella prima la si può anche scrivere individualmente mentre nella seconda si lavora su blocco di celle e prima di poterla scrivere l'intero blocco che la contiene va cancellato. Tuttavia la memoria flash ha densità maggiore di bit per unità di superficie di silicio, il che le conferisce capacità più elevata e ne riduce il costo per bit; inoltre essa richiede un solo livello di tensione di alimentazione e consuma meno energia.

Il consumo di energia modesto della memoria flash la rende attraente nei sistemi mobili, molto dipendenti dalla vita della batteria elettrica. Sistemi tipici sono i seguenti: calcolatore palmare, telefono cellulare, fotocamera digitale, riproduttore musicale portatile (per esempio MP3) e altri ancora. Nel calcolatore portatile e nel telefono cellulare la memoria flash contiene il software di sistema che governa le funzioni del dispositivo, rendendo inutile il disco magnetico o di altro tipo. Nella fotocamera digitale essa ha la funzione di contenere le immagini, mentre nel riproduttore musicale portatile contiene la traccia sonora da riprodurre. Telefono cellulare, fotocamera digitale e riproduttore musicale portatile sono in effetti buoni esempi di *sistemi embedded* (o incorporati), discussi nel Capitolo 10.

Un solo componente integrato di memoria flash non avrebbe capacità sufficiente per i sistemi appena citati, pertanto occorre costruire moduli (o banchi) di memoria flash più capaci aggregando numerosi componenti. Esistono due modi diversi ma molti diffusi per fare ciò: sono la scheda e la chiave di memoria flash, brevemente descritte nel seguito.

Scheda di memoria flash

Si possono montare numerosi componenti integrati di memoria flash su una sola scheda piatta, così ottenendo un modulo o banco molto più capace: la *scheda di memoria flash* (*flash card*). Esistono svariati modelli di scheda, ma tutte hanno interfaccia standard per renderle compatibili con una vasta gamma di prodotti; per usare la scheda basta inserirla nel connettore appropriato. La capacità della scheda varia, ma le schede più grandi possono contenere fino a 32 Gbyte. Come termine di paragone si tenga presente che la traccia sonora in formato MP3 con durata pari a un minuto impiega circa un Mbyte di memoria, e che pertanto una scheda flash da 32 Gbyte può contenere approssimativamente 500 ore di musica.

[4] Non starebbe male tradurre in italiano con *memoria lampo*, giacché è ampiamente impiegata come memoria portatile di uso rapido, ma il termine non è in uso.

Chiave di memoria flash

Sono stati ideati e messi a punto anche moduli o banchi di memoria flash di capacità decisamente molto grande, con l'intento di sostituire il disco rigido magnetico. Tale tipo di unità di memoria si chiama genericamente *chiave di memoria flash* (*flash memory stick* o *flash drive* o *pen drive*), a motivo della forma esteriore. La chiave di memoria flash è concepita per riprodurre tutte le funzionalità altrimenti offerte dal disco magnetico, tanto che si può adattare allo stesso tipo di interfaccia. Ma nonostante tanti sforzi la capacità della chiave di memoria oggi è ancora inferiore a quella del disco magnetico. Correntemente la capacità della chiave è dell'ordine di 64 Gbyte fino a 128 Gbyte. Per confronto, le unità a disco hanno capacità che eccedono un Terabyte. Inoltre, le unità a disco hanno un costo per bit molto basso.

Tuttavia la chiave di memoria flash ha il vantaggio indubbio di non contenere parti meccaniche in movimento, giacché si tratta di dispositivo elettronico allo stato solido. Ne consegue che la chiave ha tempi di *ricerca* (*seek time*) e di *accesso* (*access time*) più brevi di quelli caratteristici del disco, pertanto lavora più velocemente. Inoltre essa consuma meno energia, il che la rende attraente nei sistemi la cui sorgente di energia è la batteria elettrica, e infine è anche insensibile alle vibrazioni, che invece con facilità danneggiano il disco.

8.4 Tecnica di DMA

I blocchi di dati sono spesso trasferiti fra la memoria principale e dispositivi di I/O come i dischi. Questo paragrafo discute una tecnica per controllare tali trasferimenti senza che il processore eserciti un intervento frequente controllato da programma.

La discussione nel Capitolo 3 si concentra su trasferimenti di dati a singola parola o singolo byte fra il processore e i dispositivi di I/O. I dati vengono trasferiti da un dispositivo di I/O alla memoria innanzitutto leggendoli dal dispositivo di I/O per mezzo di un'istruzione come

```
Load    R2, DATAIN
```

che carica i dati in un registro del processore. Quindi i dati letti vengono posti in una locazione di memoria. Il processo inverso ha luogo per trasferire i dati dalla memoria verso un dispositivo di I/O. Un'istruzione per il trasferimento di dati in ingresso o uscita viene eseguita solo dopo che il processore abbia determinato che il dispositivo di I/O è pronto, tramite la scansione del suo registro di stato o l'attesa di una richiesta di interruzione. Ma in entrambi i casi c'è un costo rilevante da pagare: per ogni dato trasferito (byte o parola) il processore deve eseguire svariate istruzioni macchina. Infatti, se il metodo usato è la scansione, il processore consuma istruzioni per esaminare il bit di stato, calcolare gli indirizzi di periferica e contare i dati trasferiti. Se invece si adotta il metodo di interruzione, il processore consuma tempo per salvare in pila il contatore di programma e il registro di stato, cui si aggiungono le eventuali istruzioni necessarie per salvare e poi ripristinare anche i registri.

Tuttavia, quando occorre trasferire un blocco di dati grosso, a velocità elevata, esiste un terzo metodo alternativo ai due appena citati. Si può pensare di introdurre un'unità di controllo speciale dedicata esclusivamente al trasferimento del blocco tra periferica e memoria senza alcun bisogno di intervento da parte del processore, che resta escluso dall'operazione di I/O. Tale metodo è chiamato *accesso diretto alla memoria* o DMA (*Direct Memory Access*), dove l'attributo "diretto" mette in evidenza come il processore non vi abbia parte alcuna, o comunque molto limitata.

L'operazione di trasferimento di I/O tramite DMA è gestita dal cosiddetto *controllore di DMA* (*DMA controller*), che può fare parte dell'interfaccia di I/O della periferica oppure essere un'unità funzionale indipendente. In entrambi i casi il controllore è collegato da un lato con il bus del calcolatore e dall'altro con la periferica. In sostanza il controllore si assume

in carico tutte le operazioni che altrimenti verrebbero svolte da parte del processore. Siccome però il controllore è progettato ad hoc per tali operazioni, le esegue in modo rapido ed efficiente, più di quanto non potrebbe fare il processore, il quale è progettato per svolgere anche molti altri compiti, dunque con gli inevitabili compromessi. Il processore deve, in una prima fase, programmare il controllore informandolo su dove stia in memoria il buffer in cui mettere o tenere il blocco di dati, vale a dire mandare al controllore indirizzo iniziale e dimensione del buffer, e deve anche dirgli se l'operazione di I/O sarà di lettura o scrittura. Una volta programmato, il controllore è autonomo e in una seconda fase procede a trasferire il blocco (lettura o scrittura dello stesso) su richiesta della periferica, che stabilisce quand'è il momento; per tutta la seconda fase il controllore non abbisogna del processore, anzi lo ignora. Terminato il trasferimento, il controllore informa il processore mandandogli esso stesso un'interruzione apposita (non la periferica). Si noti come nella prima fase il controllore venga visto dal processore come se fosse una periferica ordinaria, contenente alcuni registri con cui interagire; solo nella seconda fase il controllore manifesta la sua natura, ben più attiva di quella di una normale interfaccia di I/O. I registri del controllore sono collocati a determinati indirizzi nello spazio di indirizzamento di I/O, come quelli di qualsiasi altra periferica.

La Figura 8.12 mostra i registri che il controllore di DMA deve contenere per svolgere le sue funzioni. Tali registri vengono programmati all'inizio da parte del processore e poi il controllore li usa. Si suppone siano collocati a certi indirizzi nello spazio di indirizzamento di I/O. Ci sono due registri (in basso) per contenere l'indirizzo iniziale e la dimensione, rispettivamente, del buffer di memoria per il blocco di dati; la dimensione va espressa in numero di dati (byte o parola) che costituiscono il blocco. Il terzo registro (in alto) contiene i bit di stato e di controllo. Il bit R/$\overline{W}$ in posizione 1, ovvero il bit di *lettura/scrittura*, indica il verso del trasferimento, ovvero se si tratti di immissione (cioè lettura della periferica) o emissione

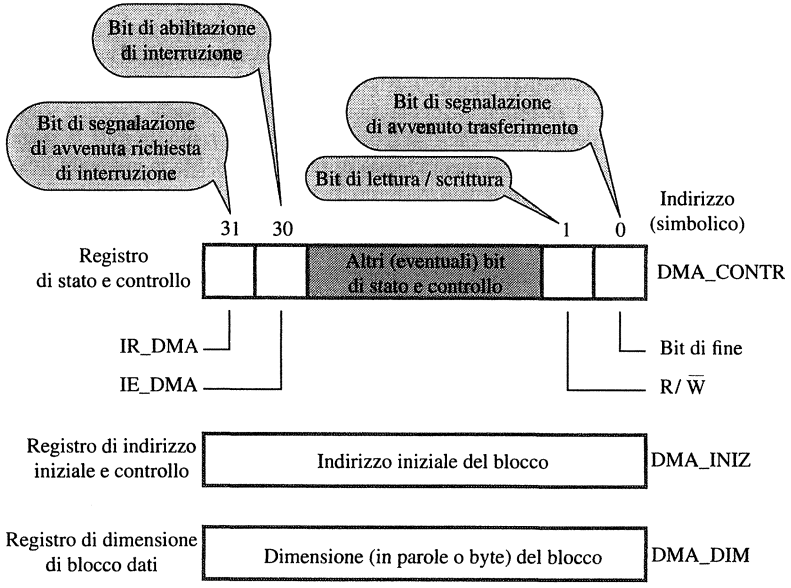


Figura 8.12 Registri del controllore DMA.

(cioè scrittura della periferica); la notazione R/$\overline{W}$ significa che si ha lettura se il bit vale 1, scrittura se vale 0. Non appena il controllore ha terminato di trasferire il blocco porta a 1 il bit di fine (in posizione 0), segnalando così di essere pronto per un nuovo trasferimento. Il bit IE_DMA (in posizione 30) serve per abilitare il meccanismo di interruzione dal lato di controllore: quando vale 1, al controllore è permesso di richiedere l'interruzione non appena ha finito di trasferire il blocco. Infine, il bit IR_DMA (in posizione 31) serve per segnalare che il controllore ha richiesto l'interruzione, quando assume valore 1.

La Figura 8.13 mostra come usare i controllori DMA in un calcolatore come quello in Figura 7.18. Vi sono il controllore che collega la scheda di rete ad alta velocità (= 100 Mbit/s) al bus del calcolatore (un bus PCI nel caso di Figura 7.18), e il controllore del disco che ha in carico la gestione di due unità disco differenti. Per ogni unità di I/O gestita in DMA si dice che esiste un *canale di DMA* (*DMA channel*). Pertanto il calcolatore di Figura 8.13 ha tre canali: due per i due dischi (il controllore del disco è dunque capace di mantenere due canali indipendenti) e uno per la scheda di rete ad alta velocità. Il controllore dei dischi riesce a gestire due trasferimenti di I/O indipendenti come se ciascun disco disponesse di un controllore riservato. Per ogni canale esistente il controllore contiene un gruppo apposito di tre registri come quelli indicati in Figura 8.12. Pertanto il calcolatore in esame ha tre gruppi: due sono nel controllore dei dischi e uno nel controllore della scheda di rete.

Si supponga di volere scrivere su disco un blocco di dati che stia in memoria (per esempio, aggiungendolo in coda a un file). Come prima cosa il programma deve predisporre i registri del canale di DMA associato all'unità disco di interesse, scrivendovi l'indirizzo iniziale del

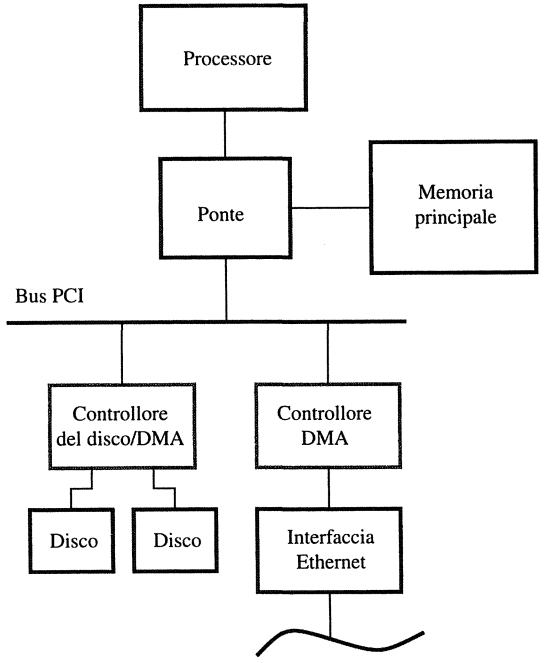


Figura 8.13 Calcolatore equipaggiato con controllori di DMA.

buffer di memoria contenente il blocco di dati e la rispettiva dimensione; esso deve anche definire il verso del trasferimento, nel caso specifico scrittura, dunque portando a 0 il bit R/$\overline{W}$ del registro di stato e controllo. Prima o poi avverrà la scrittura del blocco, vale a dire il controllore di DMA dei dischi lo avrà scritto sul disco; durante il trasferimento, il controllore lavora in modo indipendente dal processore. Non appena la scrittura è completa, il controllore annota il fatto portando a 1 il bit di fine e nello stesso istante, se il bit IE_DMA vale 1 e dunque il controllore è autorizzato a servirsi del meccanismo di interruzione, esso richiede l'interruzione al processore e porta a 1 il bit IR_DMA per lasciare traccia del fatto; altrimenti il controllore non si serve di interruzione. Beninteso i vari bit di controllo sono quelli del canale di DMA di interesse; il registro di stato e controllo, che li contiene, ha però spazio bastante anche per una certa varietà di informazioni ausiliarie ma utili: per esempio, uno o più bit per segnalare se durante il trasferimento del blocco si siano verificati errori, e altro ancora secondo il tipo di controllore.

8.5 Gerarchia di memoria

Si è già sottolineato come l'unità di memoria ideale debba essere capace, veloce ed economica. Dopo la discussione condotta nel Paragrafo 8.2 dovrebbe essere ormai chiaro che se si mira alla velocità la tecnologia giusta da adottare è quella di memoria statica SRAM. Ma essa è costosa perché le sue celle di base sono più grandi e consumano più potenza delle celle di RAM dinamica. Ne consegue che, per motivi di costo, è poco pratico costruire unità di memoria capaci servendosi di componenti integrati di memoria SRAM. L'alternativa consiste nell'usare componenti integrati di memoria dinamica DRAM, dove la cella da un bit è molto più semplice e piccola, di conseguenza, i componenti sono molto più economici; ma la tecnologia di memoria dinamica è anche molto più lenta di quella statica.

Sebbene oggi si riescano a costruire, a costo ragionevole, banchi di memoria dinamica di capacità dell'ordine dei Gigabyte, si tratta ancora di quantitativi di memoria insufficienti a fronte delle esigenze di programmi e applicazioni complessi. La soluzione viene dall'uso di memoria di massa, per lo più costituita da dischi rigidi magnetici (o anche ottici, ormai), adatti a costruire sistemi di memoria veramente molto grandi, misurabili in Tbyte (Terabyte, cioè circa 10^{12} byte). Dischi di capacità grandissima sono disponibili a prezzi ragionevoli e si usano in modo massiccio nei sistemi di calcolo. Ma il disco è ben più lento della memoria elettronica a semiconduttori, dunque si arriva alla conclusione seguente: per avere memoria di grandissima capacità si deve ricorrere alla tecnologia di disco rigido magnetico, e per avere memoria di capacità grande e più veloce la scelta giusta è la tecnologia di memoria dinamica DRAM. Tutto ciò lascia alla tecnologia di memoria statica SRAM spazio solo laddove la velocità sia l'aspetto più importante, da curare senza badare troppo al costo almeno entro certi limiti: tale ambito, tuttavia, esiste ed è quello di memoria cache.

Tutte e tre le forme di memoria appena elencate sono effettivamente usate nel calcolatore. La Figura 8.14 le mostra mettendole in relazione le une con le altre: la struttura che ne risulta va sotto nome di *gerarchia di memoria* (*memory hierarchy*) o, se si preferisce, *sistema di memoria a strati* o anche *piramide di memoria* a motivo del modo di rappresentazione. L'unità di memoria più veloce è il banco di registri interni al processore, che è anche la più piccola, ed è posta in cima alla piramide.

Subito sotto la cima della piramide si trova memoria di capacità limitata, anch'essa integrata nello stesso componente ospitante il processore. Tale unità di memoria si chiama *memoria cache di processore* (*processor cache*) e contiene una copia di parte delle istruzioni e dei dati del programma; il resto sta in memoria esterna, che è molto più capace. Si ricorderà come il concetto di memoria cache sia già stato brevemente introdotto nel Paragrafo 1.2.2,

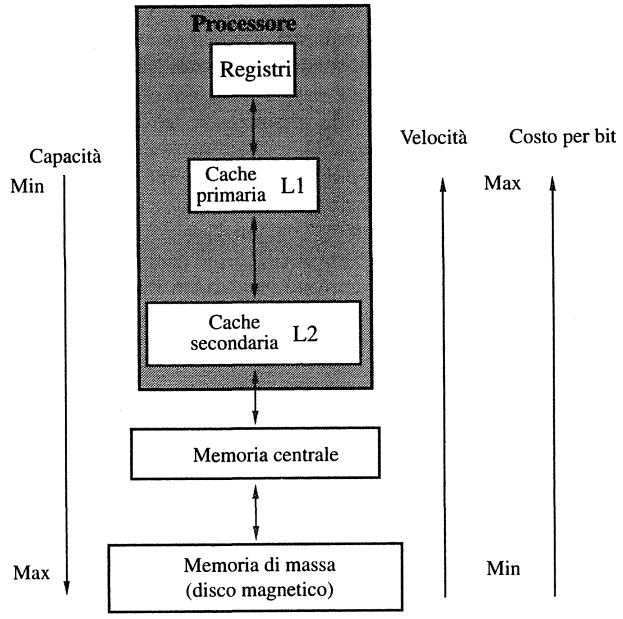


Figura 8.14 Sistema di memoria del calcolatore come piramide a strati.

ma esso verrà ripreso e discusso in profondità nel Paragrafo 8.6. Comunque la memoria cache è suddivisa in più unità, tipicamente due, poste in due strati adiacenti nella gerarchia di memoria. L'unità di *memoria cache primaria*, detta anche *di processore* o ancora di *primo livello*, indicata con la sigla L1, è sempre integrata nello stesso componente ospitante il processore. Essa è di capacità limitata perché c'è poco spazio e va spartito con il processore, che dopotutto svolge funzioni essenziali. L'unità di *memoria cache secondaria* o di *secondo livello*, indicata con la sigla L2, è più capace di quella primaria. Spesso la cache L2 viene anch'essa posta nel chip del processore.

Alcuni calcolatori hanno una cache di *livello 3* (L3) di dimensioni ancora più grandi, in aggiunta alle cache L1 e L2. Una cache L3, anch'essa realizzata con tecnologia SRAM, può o meno risiedere nello stesso chip con il processore e le cache L1 e L2.

Scendendo nella piramide lo strato adiacente alla memoria cache costituisce la *memoria centrale* o *principale* (*central* o *main memory*) del calcolatore. L'unità di memoria centrale di norma è piuttosto capace e pertanto è costruita in tecnologia DRAM di memoria dinamica, tipicamente come schiera di moduli DIMM, come descritto nel Paragrafo 8.2.5. La memoria centrale è molto più capace della memoria cache, ma è anche sensibilmente più lenta. In un calcolatore con un clock del processore di 2 GHz o più, il tempo di accesso alla memoria principale può essere fino a 100 volte più lungo del tempo di accesso alla cache L1.

Infine, alla base della piramide si trova la memoria di massa, per lo più costituita da disco rigido magnetico: essa fornisce capacità grandissima a costo minimo per bit. La memoria di massa magnetica è lentissima a confronto della memoria elettronica a semiconduttori, anche solo rispetto ai componenti di memoria dinamica impiegati per costruire la memoria centrale che tra tutti sono quelli più lenti.

La Figura 8.14 mostra che salendo nella piramide di memoria aumentano velocità e costo per bit, mentre scendendo aumenta la capacità. Garantire accesso molto rapido alla memoria durante l'esecuzione del programma è fattore di importanza massima. Per riuscirci bisogna gestire il sistema di memoria complessivo in modo appropriato: istruzioni e dati che nel futuro immediato sono destinati ad andare in uso vanno copiati verso l'alto nella piramide di memoria, cercando di avvicinarli il più possibile al processore (al limite portandoli proprio nel banco di registri del processore, se ciò è fattibile). Per ottenere tale scopo bisogna fare uso delle metodologie e tecniche che verranno spiegate nei paragrafi seguenti; si comincerà illustrando struttura e funzionamento della memoria cache.

Si ricorderà che i termini "caricamento" e "memorizzazione" sono stati ampiamente usati nei capitoli precedenti per indicare trasferimento di parole da memoria a processore e viceversa, rispettivamente. Guardando la piramide di memoria di Figura 8.14 si comprende ora come tali termini indicassero allora operazioni di trasferimento di informazioni verso l'alto o verso il basso, rispettivamente, tra la cima e gli strati sottostanti della piramide. Per estensione i termini *caricamento* e *memorizzazione* (*load* e *store*) indicano anche qui, e in generale sempre nel libro, il trasferimento del blocco di informazioni (che talvolta potrebbe consistere in una sola parola) verso l'alto o il basso, nel secondo caso sovrascrivendolo al blocco originale, tra strati qualsiasi della piramide di memoria, adiacenti (caso più comune) o meno; vale a dire flusso verticale di informazione. Il termine *copia* (*copy*) indica il trasferimento del blocco da posizione di partenza a posizione di arrivo, entrambe collocate all'interno dello stesso strato della piramide, vale a dire flusso orizzontale di informazione.[5] In concreto l'operazione di copia, tranne forse quella tra registri interni al processore, non è molto frequente benché naturalmente sia possibile e talvolta perfino necessaria.

8.6 Memoria cache

La cache è una memoria piccola e molto veloce, interposta fra il processore e la memoria principale. Ha lo scopo di mostrare al processore un sistema di memoria complessivo più veloce di quanto non sia fisicamente la sola memoria centrale.

L'efficacia della memoria cache riposa su proprietà ben dimostrate dei programmi. Ciò che conta è la proprietà di *località* che il programma esibisce nel suo complesso. Siccome si tratta di una caratteristica importante, merita di avere una discussione breve ma esauriente. L'analisi statistica dei programmi dimostra come essi passino la maggior parte del tempo eseguendo ripetutamente un numero ridotto di routine, contenenti cicli con corpo di lunghezza breve e numero di ripetizioni elevato. Insomma ci sono poche istruzioni macchina che vengono rieseguite molte volte. La distribuzione di tali istruzioni segue schemi di vario genere: esse potrebbero essere contenute in cicli semplici, in cicli annidati o fare parte di routine che si chiamano ricorsivamente le une le altre. Ma la disposizione effettiva delle istruzioni ripetitive non è importante, ciò che conta è osservare come in un certo intervallo di tempo poche istruzioni, concentrate in un segmento di programma di lunghezza limitata, vengano eseguite ripetutamente, mentre il resto del programma non va mai o raramente in esecuzione. Tale comportamento complessivo del programma, che oltre alle istruzioni riguarda anche i dati, va per l'appunto sotto il nome di *proprietà di località* (*locality of reference*). Visto così lo spunto, ora lo si amplia e sviluppa in modo sistematico.

[5] Si intende solo dire che il flusso di informazione parte e arriva nello stesso strato; può darsi che il moto sia tutto interno allo strato oppure che ne esca attraversando altri strati senza fermarsi per rientrare in quello di partenza.

Si può studiare in modo rigoroso la proprietà di località del programma suddividendola in due principi indipendenti: *località temporale* e *località spaziale* (*temporal locality* e *spatial locality*, rispettivamente). Il principio di località temporale dice quanto segue: *se un'istruzione macchina viene prelevata da parte del processore nel ciclo* (di clock) *numero i, allora con probabilità elevata la stessa istruzione verrà nuovamente prelevata nel ciclo numero* $i + p$ *di poco posteriore* (dove $p \neq 0$ è un intero positivo e ha valore piccolo); quello di località spaziale dice invece: *se il dato collocato all'indirizzo i viene usato* (letto o scritto) *da parte del processore, allora con probabilità elevata anche il dato collocato all'indirizzo vicino* $i + q$ *verrà usato* (dove $q \neq 0$ è un intero positivo o negativo e ha valore assoluto piccolo).[6] Si noti che il secondo principio non menziona il tempo: non è detto infatti che dati vicini debbano essere usati in tempi ravvicinati; mentre il primo non menziona lo spazio, giacché non c'è motivo di pensare che istruzioni ripetitive debbano anche trovarsi a breve distanza. Insomma i due principi sono *logicamente indipendenti* ed è pertanto impossibile dedurre l'uno dall'altro.

La località temporale è caratteristica delle istruzioni macchina e deriva dal fatto che il programma contiene cicli e strutture simili, come si è osservato prima. La località spaziale è caratteristica dei dati e deriva dal fatto che essi si presentano in genere aggregati in strutture dati come vettori, matrici e simili; è ben evidente che se il vettore entra in uso, tutto o gran parte del suo contenuto verrà letto o scritto dal processore, seppure non necessariamente entro tempo breve; non si capirebbe altrimenti perché il programmatore li abbia aggregati in un oggetto strutturato. Sebbene siano noti casi dove vale uno dei due principi ma non l'altro, essi in genere sono verificati simultaneamente. Ciò dipende dal fatto che quasi tutti i linguaggi di programmazione di alto livello sono di tipo ben strutturato, vale a dire la parte esecutiva è costituita da cicli ben annidati e i dati sono dichiarati sotto forma di strutture aggregate; ne consegue che per i motivi appena illustrati entrambi i principi devono valere. Si prenda come esempio la scansione del vettore, operazione quanto mai comune: il ciclo di aggiornamento dell'indice, che va incrementato ripetutamente, impone la località temporale e il vettore stesso, costituito da elementi adiacenti in memoria, quella spaziale. Insomma le istruzioni lavorano sui dati trasformandoli e i dati a loro volta condizionano il flusso di istruzioni, dunque siccome le prime e i secondi rispettano i principi di località temporale e spaziale, rispettivamente, l'insieme di entrambi, vale a dire il programma, li deve rispettare tutti e due. Se ne conclude che, benché siano indipendenti, i due principi di località si presentano spesso accoppiati nella pratica.

Quasi sempre si possono allora riunire i due principi in uno solo, chiamato principio di *località spaziale-temporale*. Eccolo: *se una parola di memoria (istruzione o dato) viene usata (letta o scritta) da parte del processore, con probabilità elevata entro breve tempo e per più volte la parola stessa e anche le parole a breve distanza da essa verranno usate*. Ma come spesso accade, accostando due nozioni importanti e indipendenti si ottiene qualcosa di più della semplice somma delle due. Da quanto appena detto è chiaro che le parole di memoria si possono immaginare riunite per piccoli gruppi o *blocchi* di parole adiacenti in memoria cioè collocate a indirizzi consecutivi, e il principio di località spaziale-temporale prende allora la forma seguente, più espressiva e illuminante: *se un blocco di parole del programma va in uso da parte del processore, con probabilità elevata entro breve tempo e per più volte esso verrà usato nuovamente*. Così si intuisce come, se si mira ad aumentare la prestazione di memoria, sia più opportuno concepire il programma come insieme di blocchi di parole piuttosto che di parole isolate, e tale osservazione ha grande peso per quanto seguirà.

Se dunque il blocco di programma, quando entra in uso, venisse messo anche in memoria cache veloce, il tempo di esecuzione totale si ridurrebbe considerevolmente. Il principio di

[6] Nello spazio ci si può muovere avanti e indietro, nel tempo solo in avanti.

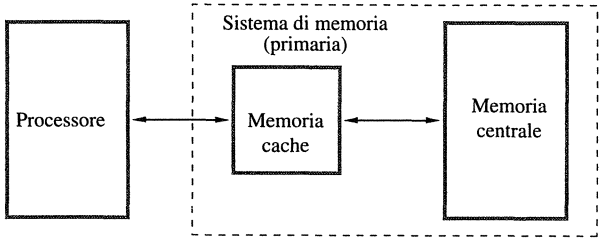


Figura 8.15 Collegamento tra processore e sistema di memoria con cache.

funzionamento della memoria cache è molto semplice: l'unità di controllo della cache sfrutta il principio di località spaziale-temporale, cioè quando il processore usa una parola di memoria il blocco che la contiene (e dunque anche la parola stessa) viene duplicato e scritto in copia nella cache, dove il processore lo potrà trovare non appena lo dovesse usare nuovamente.

Si consideri lo schema di memoria mostrato in Figura 8.15. Quando il processore manda una richiesta al sistema di memoria (memoria centrale e cache), il blocco contenente la parola interessata viene caricato dalla memoria centrale nella memoria cache, trasferendolo una parola per volta. Nella cache il blocco caricato va a occupare una porzione di memoria grande esattamente quanto basta per contenerlo; a tale regione si dà nome di *posizione di cache* o anche di *linea di cache* (*cache line*) (o ancora *blocco* se non c'è pericolo di confusione con la memoria centrale). Ciò fatto, ogniqualvolta il processore dovesse usare una qualsiasi delle parole del blocco la potrà trovare già caricata in cache dentro la posizione ospitante la copia del blocco. Di solito la cache ha posizioni bastanti a contenere un numero discreto di (copie di) blocchi di parole, ma si tratta pur sempre di un valore limitato rispetto al numero totale di blocchi presenti in memoria centrale. La corrispondenza tra blocchi di memoria centrale e posizioni di cache non può dunque essere biunivoca e va pertanto precisata secondo il caso. Quando tutte le posizioni sono occupate da blocchi e il processore richiede una parola (istruzione o dato) che non si trova in cache, il controllore di cache deve individuare una posizione da liberare per caricarvi da memoria centrale il blocco contenente la parola nuova. La procedura di scelta della posizione da liberare si chiama *algoritmo di sostituzione*, e ha un ruolo cruciale per il buon funzionamento del sistema di memoria.

Cache hit

Al processore non occorre essere informato esplicitamente intorno alla presenza di memoria cache, perché esso si limita a vedere il sistema di memoria complessivo senza sapere come sia strutturato internamente. Il processore manda al sistema di memoria comandi di lettura o scrittura generando indirizzi che si riferiscono allo spazio di indirizzamento di memoria centrale e il controllore di cache esamina se la parola indirizzata si trovi in cache. Se è così l'operazione di lettura o scrittura viene eseguita sulla parola di cache individuata: si dice che si è verificato evento di *hit di lettura di cache* o *hit di scrittura di cache* (*cache read hit* e *cache write hit*), rispettivamente, o semplicemente di *hit* (*di lettura o di scrittura*); volendo si potrebbe rendere con *centro (di scrittura o di lettura)*.

In caso di hit di lettura il processore legge la parola direttamente dalla cache e la memoria centrale non è coinvolta in alcun modo, ma se si ha hit di scrittura la situazione è più complessa e il controllore può procedere in due modi differenti. Il primo, detto *modo di scrittura immediata* (*write through*), prevede che sia la parola in cache sia la parola originale in memoria

centrale vengano aggiornate insieme. Il secondo, chiamato modo di *scrittura differita* (*write back* o anche *copy back*), prevede di aggiornare solo la parola in cache e di marcare come modificata la posizione che la contiene attivando un bit apposito, associato alla posizione e chiamato appunto *bit di modifica* (*dirty bit* o *modified bit*). Così parola in cache e originale in memoria centrale non sono più identiche, ma la parola in memoria centrale verrà aggiornata più tardi quando occorrerà liberare la posizione contenente il blocco, ricopiandolo in memoria centrale andando a sovrascrivere il blocco originale, per fare spazio in cache a un blocco nuovo. La tecnica di scrittura immediata è più semplice di quella differita, ma obbliga a scrivere in memoria centrale anche quando non sarebbe necessario, come succederebbe se la parola in cache venisse aggiornata più volte durante il soggiorno nella cache. Si noti che comunque anche la tecnica di scrittura differita dà luogo a operazioni di scrittura in memoria centrale superflue, perché quando il blocco viene riportato da cache in memoria centrale occorre ricopiarne tutte le parole, anche se una sola o poche di esse fossero state modificate. Il protocollo write-back è usato molto spesso, per trarre vantaggio dall'alta velocità con cui i blocchi di dati possono essere trasferiti ai chip della memoria.

Cache miss

Se l'operazione è di lettura e la parola indirizzata non si trova in memoria cache si verifica evento di *miss di lettura di cache* (*cache read miss*) o semplicemente di *miss* (*di lettura*); volendo si potrebbe rendere con *manco* (*di lettura*). In caso di miss di lettura il processore va messo temporaneamente in stato di sospensione. Il blocco contenente la parola incorsa in miss viene caricato da memoria centrale in memoria cache, in una posizione già libera o liberata al momento (mercé l'algoritmo di sostituzione). Dopo avere caricato per intero il blocco in cache la parola interessata viene letta dal processore, che così può ripartire. Tale modo di gestire la cache va sotto il nome di *lettura differita* (*read back*). In alternativa la parola potrebbe essere mandata al processore non appena fosse caricata da memoria centrale in cache, senza attendere di avere completato il caricamento del blocco (che comunque va fatto fino in fondo), e non appena la avesse letta il processore potrebbe ripartire.[7] Tale modo di procedere va sotto il nome di *lettura immediata* o anche di *ripartenza rapida* (*load through* o *early restart*, rispettivamente). Esso riduce un po' la durata dell'attesa da parte del processore, ma a prezzo di maggiore complessità circuitale.

Infine, se l'operazione è di scrittura e la parola indirizzata non si trova in memoria cache si verifica evento di *miss di scrittura di cache* (*cache write miss*) o semplicemente di *miss* (*di scrittura*). Se si adotta il modo di gestione di scrittura immediata, la parola incorsa in miss viene aggiornata direttamente in memoria centrale (visto che in cache non c'è) e il processore non va affatto sospeso, mentre se si adotta quello di scrittura differita prima bisogna sospendere il processore e caricare il blocco contenente la parola da memoria centrale in cache (trovandogli una posizione libera o liberandone una, come prima in lettura), poi, a caricamento completato, bisogna aggiornare la copia della parola ormai in cache (e così copia in cache del blocco e originale in memoria centrale divergono, ma vale quanto detto prima in caso di lettura), e in ultimo il processore può ripartire.

Si ricordi dal Paragrafo 6.7 che la limitazione di risorse in un processore in pipeline può causare lo stallo dell'esecuzione di istruzioni per uno o più cicli. Questo può avvenire se un'istruzione Load o Store richiede l'accesso ai dati in memoria allo stesso tempo del prelievo di un'istruzione successiva. Quando ciò avviene, il prelievo dell'istruzione viene ritardato fino a che l'ac-

[7] Tranne incorrere entro breve di nuovo in miss qualora indirizzasse una parola appartenente alla porzione non ancora trasferita del blocco tuttora in fase di caricamento; ma a medesima situazione si applica identica soluzione.

cesso ai dati non sia stato completato. Per evitare stalli alla pipeline, molti processori usano cache separate per istruzioni e dati, rendendo possibile alle due operazioni di procedere in parallelo.

8.6.1 Schema di indirizzamento

Per illustrare come si possa definire la corrispondenza tra blocchi di memoria centrale e posizioni di cache dove caricarli si farà riferimento a un esempio piccolo ma espressivo. Si consideri la memoria cache di capacità pari a 128 posizioni di dimensione identica, ciascuna in grado di contenere un blocco di 16 parole per un totale di 2.048 parole (2 K parole) e si supponga che la memoria centrale abbia indirizzo da 16 bit, dunque capacità pari a 64 K parole. La memoria centrale contiene dunque 64 K parole/16 parole = 4 K blocchi da 16 parole ciascuno. La lunghezza di parola non è importante, potrebbe essere byte oppure parola da 16 o 32 bit; qui ci si limita a dire che indirizzi consecutivi fanno riferimento a parole adiacenti in memoria centrale.

Indirizzamento diretto

La corrispondenza più semplice tra blocchi di memoria centrale e posizioni di cache è data dallo schema di *indirizzamento diretto* (*direct mapping*). Esso è una *funzione*: ogni blocco di memoria centrale è caricabile in una sola posizione di cache; si tratta dunque di una funzione vera e propria tra blocchi e posizioni, nel senso matematico del termine. Naturalmente più blocchi sono caricabili nella medesima posizione (dunque la funzione non è iniettiva), ovviamente uno solo per volta. Ecco come è strutturata la funzione: numerando indipendentemente blocchi e posizioni a partire da 0, il blocco numero $i \geq 0$ è caricabile solo nella posizione numero i mod 128 (dove 128 è il numero totale di posizioni di cache), come mostra la Figura 8.16, in cui colorazione e linee tratteggiate aiutano a riconoscere la funzione. Dunque i blocchi numero 0, 128, 256, ..., 3.968 sono caricabili nella posizione numero 0; i blocchi 1, 129, 257, ..., 3.969 sono caricabili nella posizione 1; e così via; i blocchi numero 127, 255, ..., 4.095 sono caricati nella posizione 127. Siccome più blocchi sono caricabili nella medesima posizione, si può avere conflitto quando la posizione è occupata, benché magari la cache non sia tutta piena. Per esempio in memoria centrale ci potrebbe essere un programma la cui sequenza di istruzioni macchina iniziasse nel blocco 1 e continuasse nel blocco 129 (essendo i due tratti giuntati tramite istruzione di salto). Quando il programma entrasse in esecuzione, entrambi i blocchi andrebbero caricati nella posizione 1, prima il blocco 1 e poi il blocco 129, ma ciò è impossibile. Il conflitto è risolubile sovrascrivendo il contenuto della posizione, cioè quando il blocco 129 va in esecuzione viene caricato in posizione 1 sovrascrivendone il contenuto precedente, ovvero la copia del blocco 1 ivi precedentemente scritta.

Quale sia la posizione di cache cui il blocco è destinato è stabilito dall'indirizzo di memoria centrale. Come mostra la Figura 8.16, esso è suddiviso in tre campi. Il campo *spiazzamento* contenente i 4 bit meno significativi rappresenta lo spiazzamento che individua la parola all'interno del blocco di $16 = 2^4$ parole. Il campo *blocco* contenente i 7 bit centrali rappresenta il numero della posizione (che è unica) cui il blocco è destinato tra le $128 = 2^7$ presenti. Il campo *etichetta* (*tag*) contenente i 5 bit più significativi viene registrato come etichetta associata alla posizione dove è stato caricato il blocco. L'etichetta identifica uno tra i 4.096 blocchi/128 posizioni $= 32 = 2^5$ blocchi caricabili nella posizione. Nella Figura 8.16 sono anche riportate le formule generali, del tutto ovvie, per calcolare la dimensione in bit dei vari campi.

Quando il processore genera un indirizzo di memoria centrale il campo blocco indica direttamente la posizione di cache da esaminare e il campo etichetta viene confrontato con l'etichetta correntemente associata con la posizione sotto esame. Se etichetta di posizione e campo etichetta di indirizzo coincidono, la parola di interesse è in cache nella posizione sotto esame (con tutto il blocco di parole cui essa appartiene); il campo spiazzamento indica quale collocazione la parola abbia dentro il blocco. Se non coincidono, la parola e il blocco cui appartiene

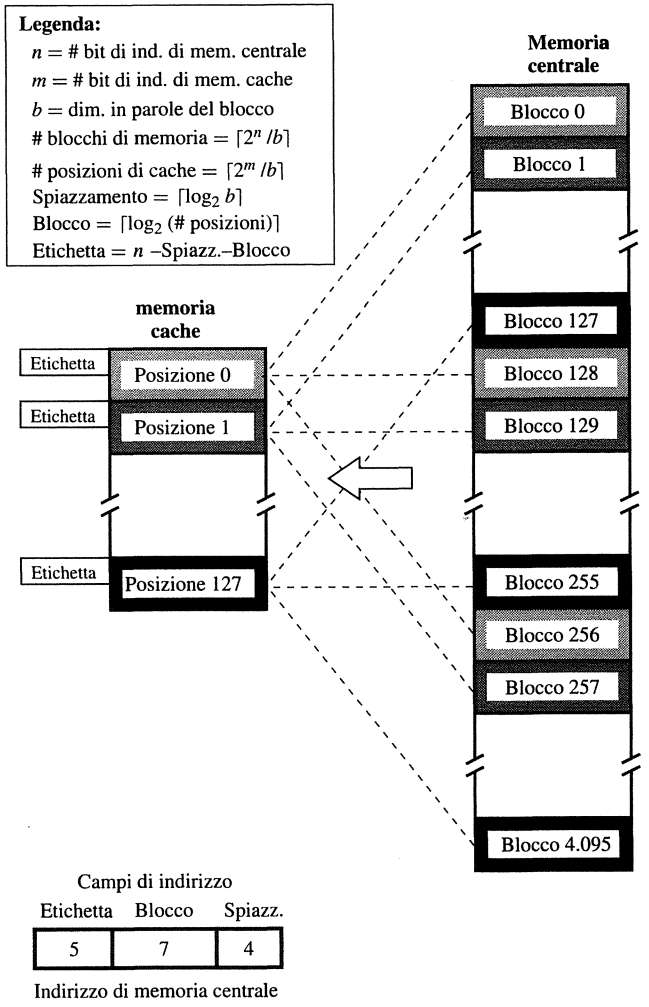


Figura 8.16 Schema di indirizzamento di tipo diretto.

non sono in cache; ovviamente stanno senz'altro in memoria centrale e di là si possono caricare in cache, tenendo conto delle osservazioni fatte in precedenza secondo il tipo di operazione di memoria (lettura o scrittura). Come si vede, lo schema di indirizzamento è facile da mettere in opera, ma non è granché flessibile.

Indirizzamento associativo

La Figura 8.17 mostra uno schema di indirizzamento più flessibile (con formule di dimensionamento), dove ogni blocco di memoria centrale è caricabile in qualsiasi posizione di cache senza restrizioni (sempre però una posizione può ospitare un solo blocco per volta). Le linee

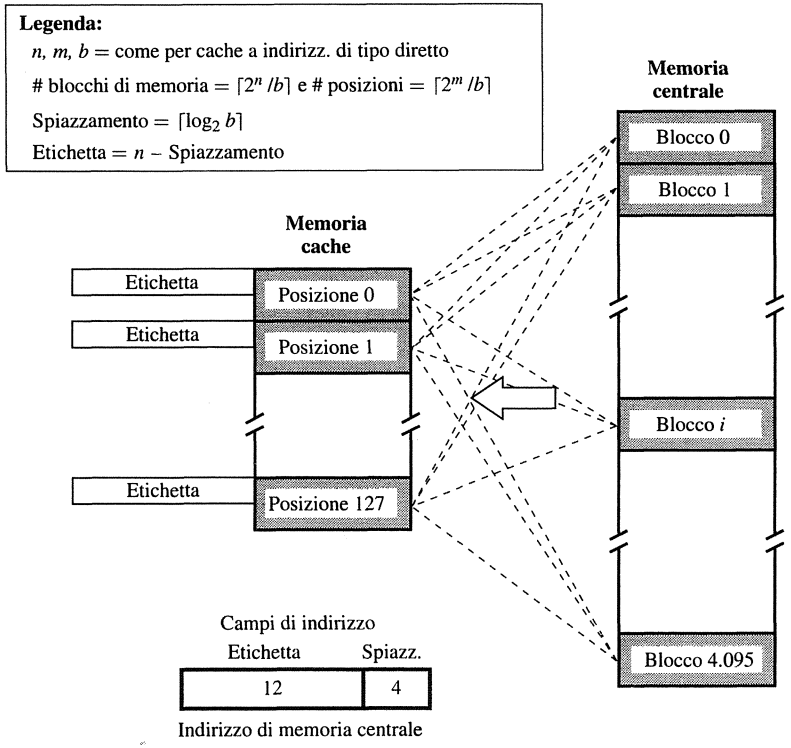


Figura 8.17 Schema di indirizzamento di tipo associativo.

tratteggiate di Figura 8.17 aiutano a riconoscere la corrispondenza tra blocchi e posizioni, e ben si vede come essa non sia una funzione (dai blocchi alle posizioni), bensì una semplice *relazione binaria*[8] oppure, se si preferisce, una *funzione a più valori* o *polidroma*.

L'indirizzo di memoria centrale è suddiviso in due soli campi, spiazzamento ed etichetta (manca il campo blocco). Il campo spiazzamento occupa 4 bit, mentre il campo etichetta occupa gli altri 12 bit necessari per identificare quale tra i $4.096 = 2^{12}$ blocchi possibili sia caricato nella posizione sotto esame. Lo schema di *indirizzamento associativo* (*associative mapping*) dà libertà completa nella scelta della posizione dove caricare il blocco, pertanto lo spazio di cache è usato in modo più efficiente. È chiaro che si ha conflitto di caricamento solo se la cache è piena, altrimenti c'è sempre una posizione libera dove caricare il blocco. Se la cache è piena e si deve caricare un blocco nuovo, bisogna liberare una posizione e dunque occorre un algoritmo per scegliere quale. Come si vedrà nel Paragrafo 8.6.2, ci sono molti algoritmi adatti allo scopo. La cache associativa ha costo maggiore di quella diretta perché per stabilire se la parola di interesse stia in cache bisogna esaminare tutte le 128 etichette di posizione registrate confrontandole con il campo etichetta di indirizzo della parola, cercando il blocco

[8] Anzi, qui si tratta della *relazione universale*, cioè ogni blocco è in relazione con ogni posizione.

cui la parola appartiene. Tale modo di localizzare un elemento in un insieme si chiama *ricerca associativa* (*associative search*); per ragioni di efficienza, la ricerca associativa dell'etichetta va condotta in parallelo su tutte le etichette presenti.

Indirizzamento associativo a gruppi

Si possono combinare gli schemi di indirizzamento diretto e associativo riunendo le posizioni di cache in gruppi e sostituendo la relazione tra blocchi di memoria centrale e posizioni di cache con una tra blocchi e gruppi (di posizioni). Dire che il blocco è in relazione con il gruppo significa dire che è caricabile a scelta in una qualsiasi delle posizioni appartenenti al gruppo (ancora, solo se la posizione è libera). Il problema di conflitto che tanto penalizza lo schema di indirizzamento diretto qui è alleviato, benché non quanto con lo schema associativo. Al tempo stesso il costo circuitale della cache si riduce perché la ricerca associativa dell'etichetta non si estende su tutta la cache ma solo sul gruppo (o sui gruppi) sotto esame. Nella pratica si usa lo schema di *indirizzamento associativo a gruppi* (*set associative mapping*), in cui la relazione tra blocchi e gruppi è una funzione (come nella cache diretta); la relazione tra blocchi e posizioni, però, *non è una funzione* perché il gruppo contiene più di una posizione. La Figura 8.18 mostra un esempio di cache associativa a gruppi (con formule di dimensionamento): il gruppo contiene 2 posizioni e la cache contiene 64 gruppi. I blocchi numero 0, 64, 128, ..., 4.032 sono caricabili nel gruppo 0, vale a dire a scelta tra la posizione numero 0 o la 1 (si intende che non si carica mai il blocco in entrambe le posizioni, sarebbe solo spreco di spazio); i blocchi 1, 65, 129, ..., 4.033 sono caricabili nel gruppo 1 (cioè in una delle due posizioni di tale gruppo); e così via, in modo simile alla cache diretta; in ultimo i blocchi 63, 255, ..., 4.095 sono caricabili nel gruppo 63. In Figura 8.18 colorazione e linee tratteggiate aiutano a riconoscere la corrispondenza tra blocchi, gruppi e in definitiva posizioni.

L'indirizzo di memoria centrale è suddiviso in tre campi: spiazzamento (meno significativo), *gruppo* (centrale) ed etichetta (più significativo); il campo gruppo generalizza il campo blocco che valeva per la cache diretta. Giacché la cache di Figura 8.18 contiene $64 = 2^6$ gruppi, il campo gruppo consta di 6 bit. Come nel caso diretto, il campo gruppo indica il gruppo dove potrebbe trovarsi la copia del blocco cui appartiene la parola di interesse. Qui però il gruppo contiene più posizioni di per sé equivalenti e dunque bisogna, dentro il gruppo sotto esame, ricercare il blocco. Il campo etichetta identifica il numero del blocco cui la parola appartiene. Esso va confrontato, mediante ricerca associativa, con le etichette di posizione registrate in cache e appartenenti al gruppo sotto esame, e la posizione in cui ci fosse coincidenza sarebbe quella dove si troverebbe la copia del blocco; altrimenti il blocco non starebbe in cache e allora valgono le solite considerazioni intorno a come comportarsi a seconda che l'operazione sia di lettura o di scrittura. Nel caso specifico il campo etichetta consta di 6 bit, giacché ogni gruppo è posto in corrispondenza con 4.096 blocchi/64 gruppi $= 64 = 2^6$ blocchi (non fa regola che i campi etichetta e gruppo siano di lunghezza identica, qui succede per caso). Siccome nell'esempio il gruppo contiene solo 2 posizioni, la ricerca associativa dell'etichetta è semplice e poco costosa.

In caso di schema associativo a gruppi si dice che la memoria cache è a n vie (n way cache) ($n \geq 2$) se il gruppo contiene n posizioni di cache. Lo schema di Figura 8.18 appena visto come esempio è uno di cache a *due vie* (*two way cache*). Lo schema associativo a gruppi è il più generale: quello associativo ne rappresenta il caso limite con un solo gruppo contenente tutte le posizioni, quello diretto ne rappresenta il caso limite con tanti gruppi quante posizioni, ciascun gruppo contenendo una sola posizione (pertanto il concetto di gruppo coincide con quello di posizione). Nondimeno è bene tenerli separati, perché hanno costo e prestazione differenti.

Il numero di posizioni per gruppo è uno dei parametri di configurazione della memoria cache e va definito sulle caratteristiche del calcolatore. Ritornando all'esempio di cache associativa a gruppi di Figura 8.18, si potrebbe rendere la cache a 4 vie con campo gruppo da 5 bit, o a 8 vie con campo gruppo da 4 bit, e così via.

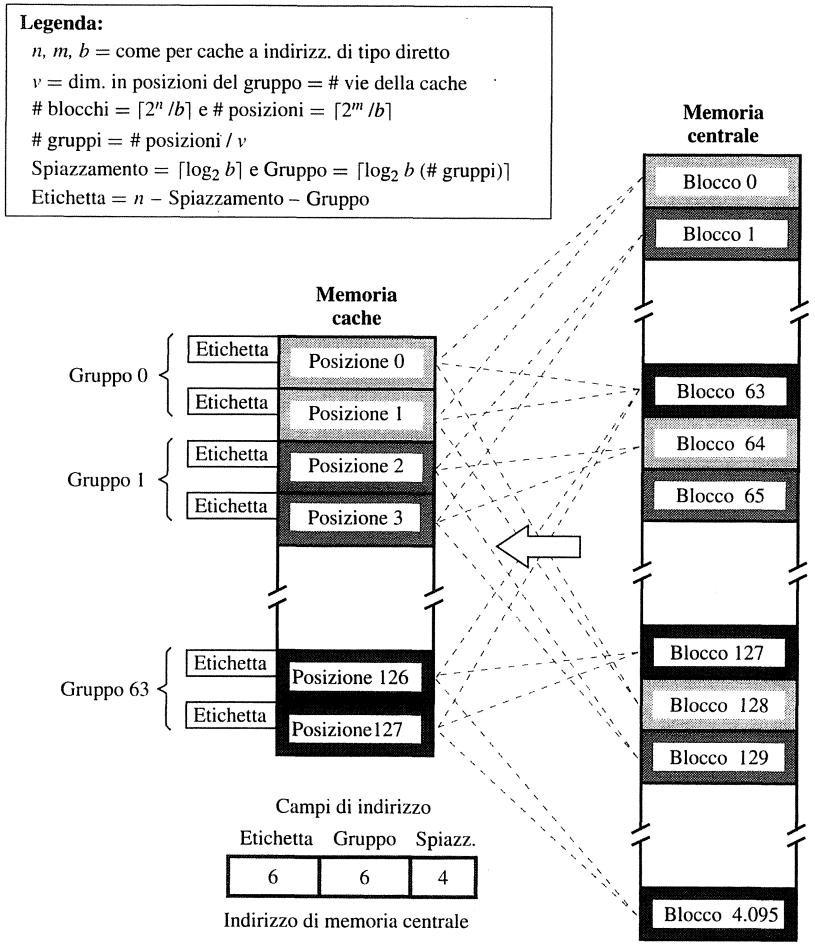


Figura 8.18 Schema di indirizzamento di tipo associativo a gruppi.

Dati scaduti

A ciascuna posizione di cache occorre associare (oltre all'etichetta) anche il *bit di validità* (*valid bit*), che indica se la posizione sia occupata dalla copia di un blocco oppure libera. Esso non va confuso con il *bit di modifica* spiegato prima: quest'ultimo si limita a segnalare se la copia del blocco diverga dall'originale in memoria centrale e occorre solo se la gestione dell'evento di miss di cache non segue il modo di scrittura immediata. All'accensione del calcolatore oppure quando nella memoria centrale si caricano da disco nuovi programmi o dati, i bit di validità di tutte le posizioni vanno posti a 0. Si tenga presente che il trasferimento da disco a memoria centrale di solito è gestito secondo la tecnica di accesso diretto a memoria o DMA, che non coinvolge affatto la memoria cache (i blocchi caricati da disco non vanno in cache, bensì direttamente in memoria centrale). Il motivo per scavalcare la cache è dato

da costo e prestazione. Ovviamente il blocco prima o poi verrà caricato in cache, e non appena ciò sarà avvenuto il bit di validità della posizione di cache riempita andrà posto a 1. Ogniqualvolta il blocco originale in memoria centrale viene aggiornato scavalcando la cache (come nel già citato caso di DMA da disco) si controlla se la copia stia in cache; se è così, il bit di validità della posizione va riportato a 0. Ciò assicura che nella cache non restino abbandonati *dati scaduti* (*stale data*), cioè copie di blocchi i cui originali sono mutati e non hanno più alcuna relazione con esse.

Difficoltà analoga insorge quando si trasferiscono dati tramite DMA da memoria centrale a disco e la memoria cache è gestita in modo di scrittura differita. Potrebbe infatti accadere che i blocchi originali in memoria centrale non riflettano ancora i cambiamenti più recenti apportati solo alle copie in cache; essi allora finirebbero comunque su disco pur non avendo più valore. Una soluzione al problema consiste nello *svuotare la cache* (*cache flush*) ricopiando forzatamente in memoria centrale i blocchi contenuti nelle posizioni con bit di modifica attivato, prima di trasferirli su disco tramite DMA. Il sistema operativo può farlo facilmente e ciò influisce poco sulla prestazione del calcolatore perché il trasferimento da memoria centrale a disco è operazione poco frequente e comunque ben più lenta dello svuotamento della cache. Il problema generale di garantire che due unità funzionali differenti (qui processore e controllore di DMA) facciano riferimento a copie identiche di blocchi di dati su cui lavorare va sotto il nome di *problema di coerenza di cache* (*cache coherence problem*).

8.6.2 Algoritmo di sostituzione

Nella memoria cache a indirizzamento diretto la posizione in cui può andare il blocco è predeterminata dall'indirizzo stesso, pertanto non occorre nessuna strategia di sostituzione, mentre nella cache associativa, eventualmente a gruppi, ci sono alcuni gradi di libertà. Quando si deve caricare in cache il nuovo blocco e tutte le posizioni dove secondo lo schema di indirizzamento vigente esso potrebbe andare sono occupate, il controllore di cache ha il compito di decidere quale sovrascrivere. Si tratta di una questione importante che potrebbe influire significativamente sulla prestazione del calcolatore. In generale l'obiettivo è quello di tenere in cache le copie dei blocchi che hanno buona probabilità di essere usati nel futuro prossimo, ma non è facile determinare quali siano. Tuttavia la proprietà di località del programma aiuta a trovare una strategia ragionevole. Siccome il programma tende a restare in regioni di memoria ben localizzate per intervalli di tempo di una certa durata c'è probabilità elevata che il blocco usato di recente lo debba essere di nuovo entro breve tempo. Pertanto quando si tratta di individuare la posizione da sovrascrivere è sensato puntare l'attenzione su quella contenente la copia di blocco che da più tempo risulta essere inutilizzata. Si dice che tale blocco è quello *usato meno di recente* (*least recently used*, LRU) e la procedura di sostituzione che ne deriva è denominata *algoritmo LRU*.

Per mettere in opera l'algoritmo LRU, però, è necessario che, mentre l'esecuzione avanza, il controllore di cache tenga traccia dell'utilizzo di tutti i blocchi in cache. Si supponga di dovere individuare il blocco LRU nella memoria cache associativa a gruppi a 4 vie (quattro posizioni per gruppo). A ogni posizione di cache bisogna associare un contatore modulo 4, cioè da due bit. L'algoritmo LRU lavora aggiornando i contatori del gruppo corrente, cioè quello cui appartiene la posizione dove si è verificato hit o miss di cache; i contatori degli altri gruppi non vengono toccati.

- Se c'è hit, il contatore della posizione interessata va azzerato e i contatori inferiori a quello da azzerare vanno incrementati di un'unità, mentre i contatori rimanenti (vale a dire quelli superiori) restano come sono.
- Se c'è miss e il gruppo corrente non è pieno, si carica il blocco nuovo (in una qualsiasi delle posizioni libere) e si aggiorna l'etichetta, il contatore della posizione appena occupata va azzerato e gli altri tre contatori vanno incrementati di un'unità.

- Se c'è miss e il gruppo è pieno, la posizione con contatore massimo, cioè 3, va liberata e subito rioccupata sovrascrivendovi la copia del blocco nuovo, l'etichetta di posizione va naturalmente allineata con quella del blocco nuovo, il contatore rispettivo va azzerato e gli altri tre contatori vanno incrementati di un'unità.

Non è difficile dimostrare che in ogni gruppo i quattro contatori sono sempre diversi l'uno dell'altro. L'algoritmo LRU si adatta in modo ovvio alla cache associativa, dove c'è un solo gruppo grande quanto la cache stessa.

L'algoritmo LRU è ampiamente impiegato nella pratica, ma, sebbene in media abbia prestazione buona, ci sono alcuni casi particolari (anche se non infrequenti) dove si comporta male. Per esempio, esso dà esito insoddisfacente quando capita di scandire sequenzialmente un vettore di parole contigue in memoria che sia di poco troppo lungo per stare tutto quanto in cache (si vedano il Paragrafo 8.6.3 e il Problema proposto 8.11 nell'Eserciziario). Si migliora la prestazione complessiva dell'algoritmo LRU perturbandolo saltuariamente in modo casuale quando prende la decisione intorno a quale blocco sostituire.

In concreto oltre a LRU si impiegano parecchi altri algoritmi di sostituzione. Per esempio, stando all'intuito sarebbe ragionevole sostituire la copia di blocco più vecchia (cioè il blocco arrivato in cache per primo). Si tratta dell'algoritmo FIFO (*first in first out*), giacché il blocco primo arrivato in cache sarà il primo a uscirne. Ma tale algoritmo non tiene conto della storia recente del blocco, la quale ha maggiore peso dell'età nel determinarne il futuro, e di solito nello scegliere il blocco da sostituire non è efficace quanto l'algoritmo LRU. La strategia più semplice consiste nello scegliere a caso con distribuzione uniforme. Potrà sembrare strano ma quest'ultimo algoritmo, pur ingenuo com'è, si dimostra molto efficace nella pratica.

8.6.3 Esempio di indirizzamento

Nel seguito si considererà un esempio adatto a illustrare i vari schemi di indirizzamento di cache. Si supponga che il processore disponga di due unità di memoria cache indipendenti, una per le istruzioni e l'altra per i dati. Per limitare la difficoltà dell'esempio si supponga che la cache di dato contenga 8 posizioni. Il blocco consta di esattamente una parola da 16 bit (dunque qui i concetti di blocco e parola coincidono), la memoria centrale è indirizzabile per parola e ha indirizzo da 16 bit, dunque capacità da $2^{16} = 64$ K parole (beninteso, tali parametri sarebbero irrealistici nel calcolatore reale, ma qui servono solo per facilitare il ragionamento). Infine si supponga che l'algoritmo di sostituzione in uso nella cache di dato sia LRU.

Ciò che si vuole fare è esaminare come cambi il contenuto della memoria cache di dato quando si esegue il programma seguente che lavora su matrice di interi. La matrice bidimensionale V di tipo 4 righe × 10 colonne, con elementi interi di dimensione pari a una parola, è collocata nella memoria centrale del calcolatore in parole di memoria contigue a partire dall'indirizzo $7A00_{hex}$ fino a $7A27_{hex}$ (sono appunto 40 elementi). In memoria la matrice V è disposta per colonne, come mostra la Figura 8.19, cioè gli elementi sono disposti in parole consecutive di memoria scandendo prima tutti gli elementi della prima colonna, poi tutti quelli della seconda e così via fino all'ultima colonna. La Figura 8.19 mostra anche come calcolare l'etichetta di ciascun blocco (ovvero parola) derivandola dall'indirizzo di memoria centrale. Si tenga presente che nell'esempio il campo spiazzamento di indirizzo è sempre assente, proprio perché qui blocco e parola coincidono; del resto il campo spiazzamento sarebbe quello più facile da comprendere e calcolare, e non c'è bisogno di esemplificarne il funzionamento.

Il programma normalizza (cioè scala) gli elementi della prima riga della matrice V rispetto alla media degli elementi nella riga stessa; esso non tocca (né legge né scrive) le altre 9 righe

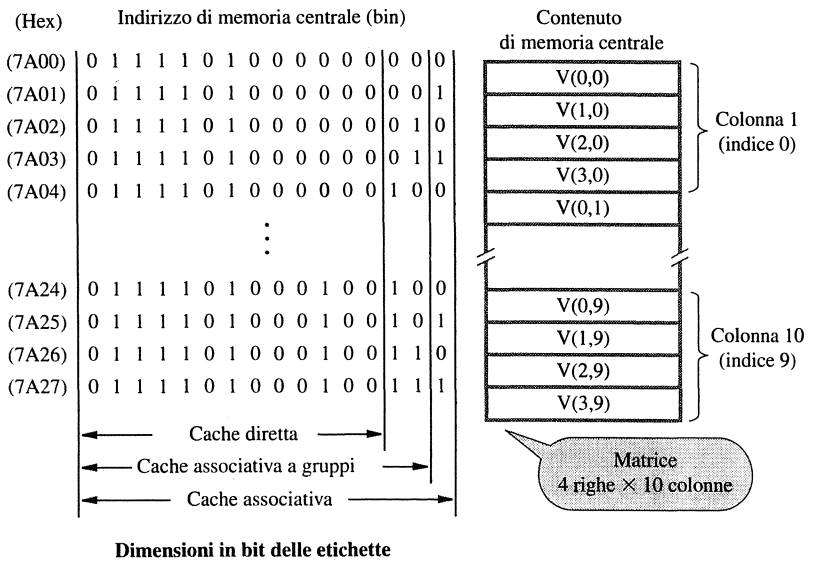


Figura 8.19 Collocazione in memoria per colonne di matrice di interi.

della matrice. Pertanto dapprima calcola la media della prima riga della matrice V, poi scala tutti gli elementi della riga per la media così calcolata. La formula seguente esprime in forma chiusa l'algoritmo del programma:

$$V[0,q] = \frac{V[0,q]}{\left(\sum_{p=0}^{9} V[0,p]/10\right)} \qquad \text{con } 0 \le q \le 9$$

La Figura 8.20 fornisce la struttura di un programma che corrisponde a questo compito. Si usano le variabili SOMMA e MEDIA per tenere rispettivamente i valori somma e media. Queste variabili, nonché le variabili indice p e q, sono tenute nei registri del processore durante l'elaborazione.

```
SOMMA := 0
  for p := 0 to 9 do
     SOMMA := SOMMA + V(0,p)
end
MEDIA := SOMMA/10
for q := 9 downto 0 do
     V(0,q) := V(0,q)/MEDIA
end
```

Figura 8.20 Programma che lavora su matrice di interi.

Indirizzamento diretto

Se la memoria cache di dato adotta lo schema di indirizzamento diretto (con indirizzo diviso in campo etichetta e blocco da 13 bit e 3 bit, rispettivamente; si veda la Figura 8.19) il contenuto di cache nel corso dell'esecuzione del programma di Figura 8.20 è quello mostrato in Figura 8.21. Gli indici di riga indicano le 8 posizioni di cache, che in effetti sono parole di cache. Il programma consta di due cicli a conteggio di tipo "for" governati dai contatori interi p e q che partono da 0 e 9, rispettivamente, e vengono aggiornati in fondo al corpo del rispettivo ciclo subito prima di girarlo nuovamente (o di terminare). Gli indici di colonna si riferiscono dunque all'istante di tempo in cui avviene il giro (o il termine) di ciclo, individuato dandone il valore di contatore (subito prima dell'aggiornamento); tale valore permette anche di stabilire quante volte il ciclo sia già girato. Infine il contenuto di ciascuna cella della tabella è l'elemento della matrice V che si trova in copia nella posizione corrispondente alla fine del giro corrente.

Per esempio in fondo al secondo giro del primo ciclo "for", cioè quando $p = 1$ (si ricordi che p parte da 0), la memoria cache contiene copie degli elementi $V[0,0]$ e $V[0,1]$ che si trovano in posizione 0 e 4, rispettivamente, come stabiliscono i tre bit meno significativi dei rispettivi indirizzi di memoria centrale (si veda la Figura 8.19). Dopo altri due giri, cioè quando $p = 3$, gli elementi $V[0,0]$ e $V[0,1]$ nelle posizioni 0 e 4 saranno stati sostituiti da $V[0,2]$ e $V[0,3]$, rispettivamente. Si va avanti così e si vede che gli elementi della prima riga della matrice V vanno sempre nelle medesime due posizioni di cache, mentre le rimanenti sei restano inutilizzate e mantengono il contenuto che avevano prima di lanciare il programma. Al termine del primo ciclo "for" saranno stati effettuati 10 carichi di blocco (parola), di cui 2 iniziali in posizioni libere e 8 di sostituzioni successive in posizioni occupate.

Al termine del primo ciclo "for", cioè quando $p = 9$, le posizioni di cache 0 e 4 contengono gli elementi $V[0,8]$ e $V[0,9]$, rispettivamente. Siccome il secondo ciclo "for" scandisce di nuovo la prima riga della matrice V ma in ordine inverso, i primi due giri (vale a dire $q = 9,8$) trovano già gli elementi in cache. Al giro $q = 7$ l'elemento $V[0,9]$ viene sostituito con $V[0,7]$ e al giro $q = 6$ successivo $V[0,8]$ con $V[0,6]$ (che è quanto si vede in Figura 8.21 passando dalla colonna $p = 9$ a $q = 6$). Procedendo in questo modo, al termine del secondo ciclo "for" saranno state effettuate 8 sostituzioni.

Si tenga ben presente che a ciascuna posizione di cache è associata l'etichetta e al momento della sostituzione anche l'etichetta va aggiornata di conseguenza. Per semplicità in Figura 8.21 le etichette di posizione e i rispettivi aggiornamenti non sono mostrati, ma si possono aggiungere facilmente.

Indice di conteggio di ciclo "for"

Posizione	$p = 1$	$p = 3$	$p = 5$	$p = 7$	$p = 9$	$q = 6$	$q = 4$	$q = 2$	$q = 0$
0	V(0,0)	V(0,2)	V(0,4)	V(0,6)	V(0,8)	V(0,6)	V(0,4)	V(0,2)	V(0,0)
1									
2									
3									
4	V(0,1)	V(0,3)	V(0,5)	V(0,7)	V(0,9)	V(0,7)	V(0,5)	V(0,3)	V(0,1)
5									
6									
7									

Figura 8.21 Contenuto di memoria cache a indirizzamento diretto.

Indirizzamento associativo

La Figura 8.22 mostra come cambia la situazione se la memoria cache adotta lo schema di indirizzamento associativo (con campo etichetta di indirizzo da 16 bit, giacché qui il campo blocco manca; si veda la Figura 8.19); si ricordi che ora bisogna ricorrere all'algoritmo LRU. Durante i primi 8 giri del primo ciclo "for", gli elementi della prima riga di V vengono caricati in posizioni di cache consecutive, supponendo che esse fossero inizialmente libere. Al nono giro ($p = 8$) l'algoritmo LRU sceglie di sostituire l'elemento $V[0,0]$ con $V[0,8]$. Al termine del ciclo ($p = 9$) l'algoritmo LRU sostituisce $V[0,1]$ con $V[0,9]$. Si sono così avuti 10 carichi di blocco, di cui 8 iniziali in posizioni libere e 2 di sostituzione in posizioni occupate.

Accade che nei primi 8 giri del secondo ciclo "for" ($q = 9,8, ..., 2$) gli elementi richiesti si trovino già in cache, ma quando $q = 1$ (penultimo giro) l'elemento richiesto è $V[0,1]$ e viene caricato sostituendo in cache quello usato meno di recente, vale a dire $V[0,9]$. Al termine del ciclo ($q = 0$) l'elemento $V[0,0]$ va a sostituire in cache $V[0,8]$.

Al termine del secondo ciclo "for" con cache associativa si sono dunque avute 2 sole sostituzioni, mentre con cache diretta se ne erano avute 8. È chiaro che la cache associativa trae beneficio dalla libertà completa lasciatale di caricare il blocco ovunque si giudichi più opportuno. In entrambi i casi, al buon uso della cache contribuisce anche il fatto che il secondo ciclo "for" del programma scandisca la prima riga della matrice V in ordine inverso. Sarebbe interessante andare a vedere che cosa sarebbe successo se il secondo ciclo avesse scandito la riga nello stesso ordine del primo. Simulando i cambiamenti di contenuto di cache, si vedrebbe che tutte le posizioni di cache verrebbero sostituite nel corso del ciclo (l'esercizio è lasciato al lettore). Si può osservare che se in tale caso l'algoritmo di sostituzione non fosse LRU ma quello puramente casuale, il degrado di prestazione non si verificherebbe (o sarebbe molto minore).

Indirizzamento associativo a gruppi

Si supponga infine che la memoria cache adotti lo schema di indirizzamento associativo a gruppi. La cache contiene 2 gruppi da 4 posizioni (parole) ciascuno o, se si preferisce, è a 4 vie. Siccome ci sono solo due gruppi, basta un bit per discriminarli, pertanto l'indirizzo di

Indice di conteggio di ciclo

Posizione	$p = 7$	$p = 8$	$p = 9$	$q = 1$	$q = 0$
0	V(0,0)	V(0,8)	V(0,8)	V(0,8)	V(0,0)
1	V(0,1)	V(0,1)	V(0,9)	V(0,1)	V(0,1)
2	V(0,2)	V(0,2)	V(0,2)	V(0,2)	V(0,2)
3	V(0,3)	V(0,3)	V(0,3)	V(0,3)	V(0,3)
4	V(0,4)	V(0,4)	V(0,4)	V(0,4)	V(0,4)
5	V(0,5)	V(0,5)	V(0,5)	V(0,5)	V(0,5)
6	V(0,6)	V(0,6)	V(0,6)	V(0,6)	V(0,6)
7	V(0,7)	V(0,7)	V(0,7)	V(0,7)	V(0,7)

Figura 8.22 Contenuto di memoria cache a indirizzamento associativo.

Indice di conteggio di ciclo "for"

	$p = 3$	$p = 7$	$p = 9$	$q = 4$	$q = 2$	$q = 0$	**Posizione**
Gruppo 0	V(0,0)	V(0,4)	V(0,8)	V(0,4)	V(0,4)	V(0,0)	0
	V(0,1)	V(0,5)	V(0,9)	V(0,5)	V(0,5)	V(0,1)	1
	V(0,2)	V(0,6)	V(0,6)	V(0,6)	V(0,2)	V(0,2)	2
	V(0,3)	V(0,7)	V(0,7)	V(0,7)	V(0,3)	V(0,3)	3
Gruppo 1							4
							5
							6
							7

Figura 8.23 Contenuto di memoria cache a indirizzamento associativo a gruppi.

memoria centrale è diviso in campo etichetta e blocco da 15 bit e 1 bit, rispettivamente (si veda la Figura 8.19). L'algoritmo di sostituzione è ancora LRU.

La Figura 8.23 mostra come cambia il contenuto di memoria cache nel corso dell'esecuzione del programma. Siccome tutti i blocchi di interesse hanno indirizzo di memoria centrale dispari sono caricabili solo nel gruppo 0. È come se la cache fosse dimezzata; simulandone l'andamento si vedrebbe che durante il primo ciclo "for" (colonne p = 3,7,9) si hanno 10 carichi di blocco, di cui 4 iniziali in posizioni libere e 6 di sostituzione in posizioni occupate, mentre nel secondo ciclo "for" (colonne q = 4,2,0) si hanno 6 sostituzioni.

Sebbene l'esempio così discusso sia alquanto semplificato, dimostra come in generale lo schema di indirizzamento associativo sia il migliore, quello associativo a gruppi mediocre e quello diretto il peggiore. Tuttavia lo schema associativo completo è complesso, e si può allora concludere che lo schema associativo a gruppi rappresenta il compromesso pratico valido.

8.7 Considerazioni di prestazione

I fattori chiave che spiegano il successo commerciale di un dato modello di calcolatore sono prestazione e costo. Ciò che occorre fare è dare la massima prestazione al minimo costo possibile. Si possono pensare tante alternative di progetto del calcolatore, ma la sfida è sempre quella di riuscire ad aumentare la prestazione senza fare lievitare il costo. Comunemente si quantifica il successo del calcolatore in esame dandone come cifra di merito il *rapporto tra costo e prestazione* (*price/performance ratio*). Il Capitolo 6 mostra come la pipeline accresca la velocità di esecuzione del programma. In questo capitolo, ci si concentra sul sottosistema della memoria.

La gerarchia di memoria descritta nel Paragrafo 8.5 è proprio il risultato della ricerca del rapporto costo/prestazione ottimale. Lo scopo principale dello schema a piramide è quello di strutturare il sistema di memoria in modo che il processore veda il tutto come se avesse tempo di accesso breve e capacità grande. Quando si usa una cache, il processore è in grado di accedere a istruzioni e dati più velocemente quando i dati delle locazioni di memoria a cui si riferisce sono nella cache. Quindi, la misura in cui la cache migliora le prestazioni è dipendente dalla frequenza con cui le istruzioni o i dati richiesti sono trovati nella cache. In questa parte si esamina tale questione quantitativamente.

8.7.1 Guadagno di prestazione

Un ottimo indicatore di efficienza per la gerarchia di memoria è il tasso di successo nell'accesso al contenuto, calcolato per ciascuno strato della piramide. Il rapporto tra numero di accessi al contenuto incorsi in evento di hit e numero totale di accessi (comprendente anche quelli incorsi in evento di miss) si chiama *tasso di hit* (*hit rate*). Il rapporto tra numero di accessi incorsi in miss e numero totale di accessi si chiama invece *tasso di miss* (*miss rate*). Va da sé che *tasso di hit* + *tasso di miss* = 1 e ovviamente si ha $0 \leq$ tasso (di hit o miss) ≤ 1. Comunemente i due tassi sono espressi in percentuale; per esempio 0,5 equivale a 50%.

Idealmente la gerarchia di memoria dovrebbe apparire al processore come un'unità di memoria individuale, funzionante alla frequenza dello strato più veloce della piramide, vale a dire la memoria cache in cima, e dotata di capacità pari a quella dello strato più grande della piramide, vale a dire la memoria di massa alla base, che spesso è un disco rigido magnetico.[9] Tuttavia quanto la realtà si avvicini a tale modello ideale dipende in larga misura dai tassi di hit dei vari strati intermedi della piramide. In generale occorrono tassi di hit elevati, superiori a 90% (cioè 0,9), se si vogliono costruire calcolatori con prestazione elevata.

La prestazione si degrada in conseguenza delle azioni da intraprendere per rimediare all'evento di miss. Si penalizza la prestazione a causa del tempo ulteriore necessario per trasferire un blocco di dati da un'unità più lenta nella gerarchia della memoria a un'unità più veloce. Durante tale periodo il processore è in stallo, in attesa di istruzioni o dati. Il tempo di attesa dipende dai dettagli di funzionamento della cache; per esempio, dipende dall'uso o meno del modo di lettura immediata. Il tempo totale di accesso visto dal processore quando accade una miss è detto *penalità di miss*. Si consideri ora l'effetto che la presenza di memoria cache ha sulla prestazione complessiva del calcolatore. Si denoti con h ($0 \leq h \leq 1$) il tasso di hit, con M il tempo di penalità di miss, cioè il tempo necessario per caricare il blocco da memoria centrale in cache, e con H il tempo di accesso della cache. Il tempo medio di accesso $\Delta T_{accesso}$ del sistema di memoria centrale con cache è dato dalla formula (fondamentale) seguente:

$$\Delta T_{accesso} = hH + (1 - h)M \tag{8.1}$$

L'esempio seguente illustra l'influenza dei valori di questi parametri sul tempo medio di accesso.

Esempio 8.1

Si consideri un calcolatore che ha i seguenti parametri. I tempi di accesso alla cache e alla memoria principale sono rispettivamente τ e 10τ. Quando si ha un evento di miss, un blocco di 8 parole viene trasferito dalla memoria principale alla cache. Occorre un tempo 10τ per trasferire la prima parola del blocco, mentre le rimanenti 7 parole vengono trasferite alla frequenza di una parola ogni τ. La penalità di miss include pure un ritardo di τ per l'accesso iniziale alla cache, che si rivela in una miss, e un altro ritardo di τ per trasferire la parola al processore dopo che il blocco sia stato caricato nella cache (assumendo che non si usi il modo di lettura immediata). Quindi la penalità di miss in questo calcolatore è data da:

$$M = \tau + 10\tau + 7\tau + \tau = 19\tau$$

Si supponga che circa il 30% delle istruzioni eseguite dal programma tipico comprenda un'operazione di lettura o scrittura di memoria, cioè che si verifichino 100 + 30 = 130 accessi a memoria (100 prelievi di istruzione e 30 letture e scritture di dato) ogni 100 istruzioni eseguite. Si supponga poi

[9] Vale la pena di ribadire come la capacità complessiva della piramide di memoria coincida con quella dello strato di base, la memoria di massa; *non ha alcun senso* cumulare le capacità dei vari strati, perché tutti tranne la base contengono solo copie di blocchi esistenti in originale negli stati inferiori, e dunque non contengono informazioni nuove.

che il tasso di hit di cache valga 0,95 per le istruzioni e 0,9 per i dati. Si supponga infine che la penalità di miss di cache sia identica in caso di lettura e scrittura. La formula seguente, costruita in modo evidente generalizzando la formula (8.1), dà una stima approssimata del guadagno di prestazione g reso possibile dalla presenza di cache:

$$\frac{\text{Tempo senza cache}}{\text{Tempo con cache}} = \frac{130 \times 10\tau}{100(0{,}95\tau + 0{,}05 \times 19\tau) + 30(0{,}9\tau + 0{,}1 \times 19\tau)} = 4{,}7$$

Questo risultato mostra che la cache fa apparire la memoria quasi cinque volte più veloce di quello che realmente è. Il guadagno cresce con l'aumentare della velocità della cache rispetto alla memoria principale. Per esempio, se il tempo di accesso alla memoria principale è 20τ, il guadagno diventa 7,3.

Alti tassi di hit sono essenziali per l'efficacia della cache nel ridurre i tempi di accesso alla memoria. I tassi di hit dipendono dalla dimensione della cache, dalla sua progettazione e dal modello di accesso a istruzioni e dati dei programmi in esecuzione. È anche interessante considerare quanto sia efficace la memoria cache reale (configurata come prima) a confronto dell'unità cache ideale capace di raggiungere tasso di hit pari a 100%. Con il comportamento ideale della cache, tutti i riferimenti alla memoria prendono tempo τ. Quindi una stima dell'aumento del tempo di accesso alla memoria causato da eventi di miss è data da:

$$\frac{\text{Tempo per cache reale}}{\text{Tempo per cache ideale}} = \frac{100(0{,}95\tau + 0{,}05 \times 19\tau) + 30(0{,}9\tau + 0{,}1 \times 19\tau)}{130\tau} = 2{,}1$$

In altre parole, un tasso di hit del 100% nella cache farebbe apparire la memoria due volte più veloce rispetto a quando si usano tassi di hit realistici.

Come si può fare per aumentare il tasso di hit? Una risposta ovvia sarebbe quella di aumentare la capacità della memoria cache, ma ciò ne farebbe senz'altro lievitare il costo e inoltre la velocità diminuirebbe. Un'altra possibilità consisterebbe nell'ingrandire il blocco pur tenendo costante la capacità della cache, con lo scopo di sfruttare meglio la località spaziale. Infatti, se tutte le parole del blocco dovranno prima o poi andare in uso, tanto vale caricarle tutte quante in cache a seguito di un solo evento di miss, piuttosto che caricare una serie di blocchi più piccoli a seguito di una molteplicità di eventi di miss. L'efficienza del trasferimento di una raffica di dati contigui con una sola operazione è la ragione che suggerirebbe di agire nel primo modo. Ma aumentare la dimensione del blocco funziona fino a un certo punto, poi la crescita del tasso di hit rallenta e infine esso satura perché il blocco grande, oltre a elementi destinati ad andare in uso, finisce inevitabilmente per contenere anche elementi inutili che occupano spazio in cache a scapito di elementi utili e fanno pertanto crescere il tasso di miss. Siccome la prestazione del calcolatore aumenta con il tasso di hit e diminuisce con quello di miss, ci deve essere una dimensione ottimale per il blocco. Nella pratica si è visto che la dimensione ottimale è quasi sempre compresa tra 16 e 128 byte (ovviamente la dimensione sarà una potenza di due), intervallo dove si colloca la maggior parte dei calcolatori equipaggiati con cache.

Infine bisogna tenere presente che si può ridurre la penalità di miss gestendo la memoria cache in modo di lettura immediata per caricare il blocco nuovo. Infatti il processore, invece di restare fermo in attesa che il blocco sia caricato per intero in cache, può ripartire non appena è arrivata la parola da leggere, la cui assenza in cache aveva dato luogo all'evento di miss.

8.7.2 Integrazione con il processore

Trasferendo informazioni tra unità funzionali differenti, i circuiti interni dedicati al pilotaggio dei morsetti esterni dei componenti integrati comunicanti danno luogo a ritardi di propagazione dei segnali tutt'altro che trascurabili. Pertanto, se si mira ad avere velocità elevata, la

collocazione più opportuna per la memoria cache è sullo stesso componente ospitante il processore. La maggior parte dei chip di processore includono almeno una cache L1. Spesso ci sono due cache L1 separate, una per le istruzioni e un'altra per i dati.

Nei processori ad alte prestazioni, sono normalmente usati due livelli di cache, cache L1 separate per istruzioni e dati e una cache L2 più grande. Queste cache sono spesso realizzate sul chip del processore. In questo caso, le cache L1 devono essere molto veloci, poiché determinano il tempo di accesso alla memoria visto dal processore. La cache L2 può essere più lenta, ma dovrebbe essere molto più grande della cache L1 per assicurare una frequenza di hit alta. La sua velocità è meno critica poiché influisce solo sulla penalità di miss delle cache L1. Un calcolatore tipico può avere cache L1 con capacità di decine di kilobyte e una cache L2 di centinaia di kilobyte o possibilmente diversi Megabyte.

Equipaggiando il sistema con memoria cache di secondo livello si riduce l'incidenza della velocità della memoria centrale sulla prestazione complessiva del calcolatore. Il tempo medio di accesso $\Delta T_{accesso}$ visto dal processore interagendo con il sistema di memoria complessivo dotato di due livelli di cache (unità L1, unità L2 e memoria centrale) è dato dalla formula seguente:

$$\Delta T_{accesso} = h_1 H_1 + (1 - h_1) h_2 H_2 + (1 - h_1)(1 - h_2) M \qquad (8.2)$$

che generalizza la formula (8.1) già incontrata, perché ponendo $h_2 = 0$ si riduce a quella (equivale a eliminare la cache L2. Il significato dei vari simboli che figurano nella formula (8.2) è il seguente:

- h_1 – tasso di hit nella memoria cache di primo livello L1;
- h_2 – tasso di hit nella memoria cache di secondo livello L2;
- H_1 – tempo di accesso alla memoria cache di primo livello L1;
- H_2 – tempo di accesso alla memoria cache di secondo livello L2;
- M – tempo di accesso alla memoria centrale (penalità di miss).

Il tasso di eventi di miss di unità L2, espresso dal termine composito $(1 - h_1)(1 - h_2)$, dovrebbe essere sempre piccolo. Se i termini h_1 e h_2 non sono mai inferiori a 0,9 (cioè 90%) si ha che il termine composito non è superiore a $0{,}1 \times 0{,}1 = 10^{-1} \times 10^{-1} = 10^{-2} = 0{,}01$ cioè 1%, vale a dire il numero di eventi di miss in L2 è meno di un centesimo del numero di operazioni di memoria effettuate dal processore. Ne viene che l'incidenza del tempo di accesso a memoria centrale (penalità di miss) sulla prestazione complessiva è alquanto attenuata (rispetto a non avere unità L2 ma solo unità L1). Si veda il Problema proposto 8.14 nell'Eserciziario per un esame quantitativo di tale questione.

8.7.3 Altre migliorie

Oltre alle metodologie fondamentali di progetto di memoria cache esposte sopra ci sono svariate tecniche ausiliarie per migliorare la prestazione del calcolatore. Si discutono tre di esse in questo paragrafo.

Buffer di scrittura

Se per gestire la memoria cache si adotta il modo di scrittura immediata, ogni operazione di scrittura va a lavorare con la memoria centrale. Se il processore deve attendere il completamento dell'operazione di trasferimento, come si è sempre supposto finora, a ogni scrittura viene ritardato. Tuttavia di solito il processore non dipende immediatamente dall'esito dell'operazione di scrittura, vale a dire non avrebbe necessità di attenderne il completamento.

Per aumentare la prestazione si può dotare il sistema di *buffer* (o *tampone*) *di scrittura* (*write buffer*), verso cui ridirigere temporaneamente l'operazione di scrittura. Il processore annota l'operazione nel buffer e così è subito libero di proseguire con la prossima istruzione macchina. L'operazione viene applicata alla memoria centrale in seguito, quando essa è libera e non sta rispondendo a comandi di lettura. È importante però che l'operazione di lettura venga svolta subito, perché di norma il processore non può proseguire senza il dato che ha chiesto di leggere da memoria, pertanto l'operazione di lettura ha precedenza su quella di scrittura.

Il buffer di scrittura potrebbe contenere un certo numero di operazioni annotate. Può dunque succedere che una richiesta di lettura successiva faccia riferimento al dato scritto da un'operazione precedente che però stia ancora nel buffer. Per assicurare la correttezza logica del flusso di risultati intermedi prodotti dal programma e in definitiva del risultato finale, l'indirizzo della parola da leggere da memoria viene prima confrontato (in parallelo) con gli indirizzi delle parole da scrivere ancora annotate nel buffer e se si trova corrispondenza l'operazione di lettura acquisisce il dato relativo dal buffer, non dalla memoria centrale.

La situazione muta se la memoria cache è gestita in modo di scrittura differita. In tale caso l'operazione di scrittura agisce solo su cache, non su memoria centrale. Ma si provi a considerare che cosa accadrebbe quando il blocco nuovo venisse caricato in cache a seguito di miss di lettura, andando a sostituire un blocco esistente il cui bit di modifica risultasse attivato (o uno dei bit, se ne avesse più di uno): il blocco esistente andrebbe ricopiato in memoria sovrascrivendolo all'originale. Se lo si fa, il processore, prima di proseguire, deve attendere la fine della ricopiatura del blocco vecchio e del caricamento successivo del blocco nuovo, dunque a lungo. Sembra pertanto consigliabile invertire le due operazioni caricando per primo il blocco nuovo. Ma per farlo occorre prevedere un buffer di scrittura dove trasferire al volo il blocco vecchio (quello che sarebbe da sovrascrivere all'originale in memoria centrale perché ha bit di modifica attivato) in parallelo al caricamento del blocco nuovo. In seguito, quando ormai il processore sarà già ripartito, il contenuto del buffer di scrittura andrà ricopiato in memoria centrale. Si vede dunque come a prezzo di qualche complicazione nel controllo il buffer di scrittura funzioni bene anche in modo di scrittura differita.

Caricamento anticipato

In tutta la discussione precedente si è sempre supposto che istruzioni e dati venissero caricati in memoria cache quando fossero richiesti per la prima volta. Si può riassumere dicendo che il contenuto di cache viene modificato solo a seguito di evento di miss: il miss (per esempio di lettura) si verifica e di conseguenza il blocco nuovo viene caricato; se non ci fossero eventi di miss il contenuto di cache sarebbe costante. Il processore deve attendere che il caricamento del blocco nuovo sia completo, ciò che costituisce l'effetto della penalità di miss: nessun miss, nessuna attesa. Insomma la cache si comporta in modo passivo, reagendo all'evento di miss ma non anticipandolo.

Per evitare il fermo del processore si può pensare di *caricare in anticipo* (*prefetch*) o *precaricare* blocchi in memoria cache prima che vadano effettivamente in uso. Il modo più semplice per farlo è tramite programma: si aggiunge al linguaggio macchina del processore l'istruzione speciale di precarico; quando tale istruzione viene eseguita, il dato da essa indirizzato (vale a dire il blocco che lo contiene) viene caricato in cache. Si noti come l'istruzione di precarico simuli il verificarsi di evento di miss di lettura (ciò ricorda da vicino il concetto e il meccanismo di trappola o interruzione software, benché la finalità sia poi assai diversa). Tuttavia il processore non si mette in attesa di completamento del caricamento. Bisogna dunque che l'istante di tempo di esecuzione dell'istruzione di precarico preceda l'istante di uso del dato e che l'intervallo di tempo duri abbastanza a lungo per completare il caricamento. Bisogna anche sperare che il caricamento anticipato avvenga mentre il processore è indaffarato a eseguire istruzioni che non incorrano in miss di lettura, dando così modo all'operazione di caricamento da memoria in cache di svolgersi in parallelo alle attività normali del processore (giacché se non c'è miss esso lavora solo in cache).

In quali punti del programma inserire le istruzioni macchina di precarico? Questo può essere deciso dal programmatore o dal compilatore. Sarebbe preferibile lasciare l'incombenza al secondo, che del resto riesce spesso a farlo con successo per numerosi tipi di programma. Si noti però che il caricamento anticipato pilotato da programma dà luogo a costo aggiuntivo perché disseminare il programma di istruzioni di precarico lo allunga. Inoltre talvolta l'istruzione di precarico sbaglia e manda in cache dati che poi il programma, sviluppandosi in modo imprevedibile, non userà. Ciò accade, per esempio, quando il blocco caricato in anticipo non fa in tempo a entrare in uso perché viene estromesso dalla cache a seguito di miss di lettura seguito da sostituzione. Nondimeno l'effetto complessivo del caricamento anticipato da programma sulla prestazione del calcolatore è positivo e non pochi processori dispongono di istruzioni macchina dedicate a tale funzione. Per una discussione approfondita del precarico da programma si veda [1].

Il caricamento anticipato è gestibile anche direttamente dall'unità di controllo della memoria cache. Essa deve contenere un qualche tipo di circuito capace di tenere traccia dell'andamento delle operazioni di caricamento di blocco in cache e di scoprire la presenza di regolarità ripetitive sfruttabili per individuare i blocchi da caricare in anticipo. Sono stati ideati numerosi schemi adatti allo scopo, ma esulano dallo scopo del libro e pertanto si rimanda alla letteratura scientifica di settore [2, 3].

Cache non bloccante

Lo schema di caricamento anticipato in memoria cache appena discusso non funziona bene se interferisce con il flusso di esecuzione normale del programma. Tale sarebbe il caso dove l'operazione di caricamento anticipato tenesse bloccati altri accessi a cache finché non si fosse completata. L'unità di cache che si comporta così si dice di tipo *bloccante* (*blocking cache*). La soluzione al problema consiste nel modificare il comportamento dell'unità di controllo di cache permettendo al processore di continuare a lavorare in cache anche mentre è in corso il servizio di caricamento di blocco a seguito di evento di miss. In effetti si va anche oltre, giacché è desiderabile si possano avere due o più eventi di miss il cui servizio sia contemporaneamente in corso (di nuovo si vede trasparire una certa analogia con il concetto di routine di servizio di interruzione).

L'unità di memoria cache capace di gestire eventi di miss multipli è detta *non bloccante* (*lockup-free cache*). Siccome dal punto di vista fisico essa è in grado di servire un solo evento di miss per volta, deve contenere circuiti che tengano traccia di tutti gli eventi pendenti. Ciò si può ottenere prevedendo un registro apposito contenente tutte le informazioni di stato pertinenti a tali eventi. La cache non bloccante venne introdotta per la prima volta nei calcolatori Cyber prodotti da Control Data Company [4].

La tecnica di caricamento anticipato rappresenta una ragione valida per avere memoria cache non bloccante in caso di miss di lettura. Ma la ragione più importante è che nel processore di tipo pipeline, dove più istruzioni sono in esecuzione in parallelo, l'evento di miss di lettura causato da una determinata istruzione potrebbe avere l'effetto di bloccare anche alcune delle altre che il processore stesse eseguendo (o come si dice di mandarle in *stallo*). La cache non bloccante riduce la probabilità che tale evento abbia luogo.

8.8 Memoria virtuale

Nella maggior parte dei calcolatori odierni la memoria centrale fisicamente installata non è grande quanto lo spazio di indirizzamento gestibile da parte del processore. Per esempio, il

processore che genera indirizzi da 32 bit ha 4 Gbyte come spazio di indirizzamento. Tuttavia nel calcolatore la capacità tipica di memoria può variare da 1 a 4 Gbyte. Se il programma non ci sta per intero in *memoria primaria* (= insieme di memoria centrale e memoria cache) la parte correntemente non in uso viene lasciata o scritta temporaneamente in *memoria secondaria* o di massa, come il disco rigido magnetico. Naturalmente prima o poi ogni parte del programma è destinata ad andare in uso e occorrerà caricarla in memoria primaria. Quando occorre caricare il nuovo segmento di programma in genere bisogna estrometterne un altro per fare spazio, perciò i sistemi operativi attuali sono capaci di trasferire codice e dati del programma tra memoria primaria e secondaria in modo automatico. In tale modo il programmatore e l'utente non devono neppure essere consapevoli dell'esistenza di eventuali limitazioni di capacità di memoria centrale. Essi preparano i programmi usando l'intero spazio degli indirizzi del processore.

L'indirizzo binario generato dal processore per prelevare l'istruzione o per lavorare con il dato si chiama *indirizzo virtuale* o *indirizzo logico* (*virtual address* o *logical address*). L'indirizzo logico viene tradotto in *indirizzo fisico* (*physical address*) tramite un processo misto diretto in parte dal sistema operativo (SO) e in parte da unità funzionali preposte. Se l'indirizzo logico fa riferimento a una parte dello spazio di codice o di dato che si trova correntemente in memoria primaria, il processore ottiene accesso immediato alla parola di interesse, altrimenti il contenuto di tale parte di spazio di indirizzamento va caricato in memoria primaria trovandogli lì collocazione opportuna.

La Figura 8.24 mostra la struttura tipica di memoria virtuale. Tutto si incentra su un'unità funzionale apposita chiamata *unità di gestione di memoria* o MMU (*memory management unit*), che traduce l'indirizzo logico in fisico. Se accade che la parola (istruzione o dato) su cui lavorare si trovi già in memoria primaria, essa viene trasferita come già ampiamente spiegato parlando di memoria cache, altrimenti la MMU forza il sistema operativo a caricare da disco in memoria centrale il blocco (pagina) di codice o dati contenente la parola di interesse.[10] Il trasferimento di blocchi (pagine) tra memoria centrale e disco viene generalmente svolto tramite tecnica di DMA, una delle tecniche di I/O fondamentali già spiegate nel Paragrafo 8.4. L'intervento del SO è forzato dal segnale di *fault di pagina* (o *manco di pagina*), che si attiva quando manca la pagina e funziona come un'interruzione (in effetti è un'eccezione generata internamente al calcolatore).

8.8.1 Traduzione di indirizzo

Un modo semplice per tradurre l'indirizzo logico in indirizzo fisico consiste nello strutturare il programma (segmento di codice e segmenti di dato) in unità di dimensione fissa chiamate *pagine* (*page*), ciascuna costituita da una successione di parole contigue in memoria centrale e dunque aventi indirizzi consecutivi. Comunemente la dimensione di pagina va da 2 K a 16 K parole (o byte) ed è pertanto alquanto superiore alla dimensione tipica del blocco caricabile in memoria cache (cioè della posizione di cache); ciò motiva in parte la differenza di nome. La pagina è l'unità elementare di informazione trasferibile tra memoria centrale e disco che il meccanismo di memoria virtuale sia in grado di movimentare. La pagina non dovrebbe avere dimensione troppo piccola perché il tempo di accesso del disco magnetico (parecchi millisecondi) è alquanto maggiore di quello della memoria elettronica (al massimo poche centinaia di nanosecondi). La ragione principale di tale lunga durata è data dal fatto che localizzare dati sul disco è un'operazione parzialmente meccanica e pertanto consuma tempo;

[10] Si è detto parlando di memoria cache che il controllore di DMA non trasferisce dati direttamente in cache, per motivi di costo ed efficienza.

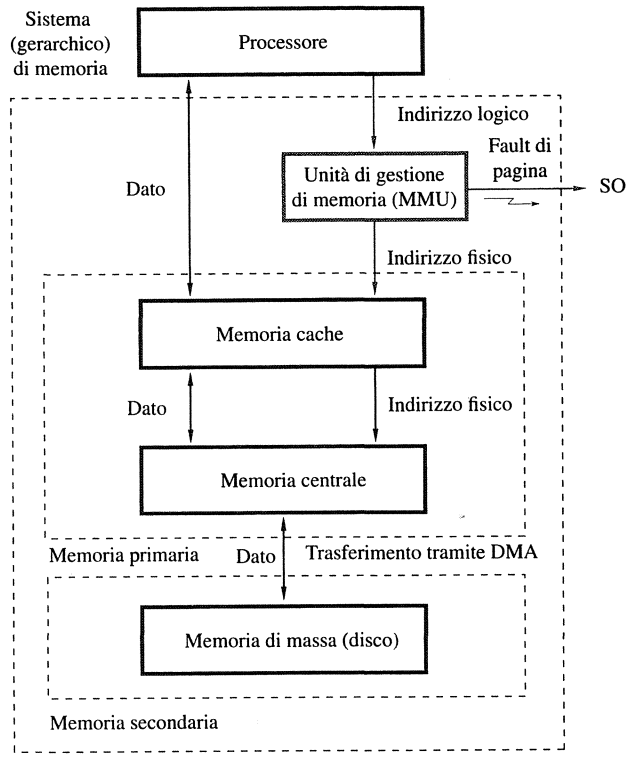


Figura 8.24 Struttura di memoria virtuale e relazione tra MMU e cache.

tuttavia una volta trovati sul disco, i dati sono trasferibili al tasso di parecchi Mbyte al secondo. D'altro canto se la pagina fosse di dimensione eccessiva buona parte del contenuto non sarebbe mai in uso, benché occuperebbe spazio di memoria prezioso e altrimenti usabile in modo più produttivo.

La discussione appena condotta ha un chiaro parallelismo con i concetti introdotti nel Paragrafo 8.6 in merito alla memoria cache. Il meccanismo di cache fa da ponte per superare il divario di *velocità* tra processore e memoria centrale ed è costituito da unità funzionali apposite. Il meccanismo di memoria virtuale fa da ponte per superare il divario di *capacità* tra memoria secondaria e primaria ed è realizzato in parte dal sistema operativo, dunque in modo programmato, in parte da unità funzionali apposite, dunque in modo circuitale. Concettualmente i due meccanismi, di memoria cache e virtuale, sono molto simili, ma differiscono parecchio nei particolari realizzativi.

La funzione di traduzione di indirizzo logico in fisico, fondata sul concetto di pagina con dimensione fissa, è mostrata in modo schematico in Figura 8.25. L'indirizzo logico generato dal processore, non importa se riferito a istruzione da prelevare oppure a dato da trasferire, viene suddiviso in due campi: il *numero di pagina logico* (*virtual page number*), costituito dalla parte più significativa, e lo *spiazzamento* (*offset*), costituito da quella meno significativa.

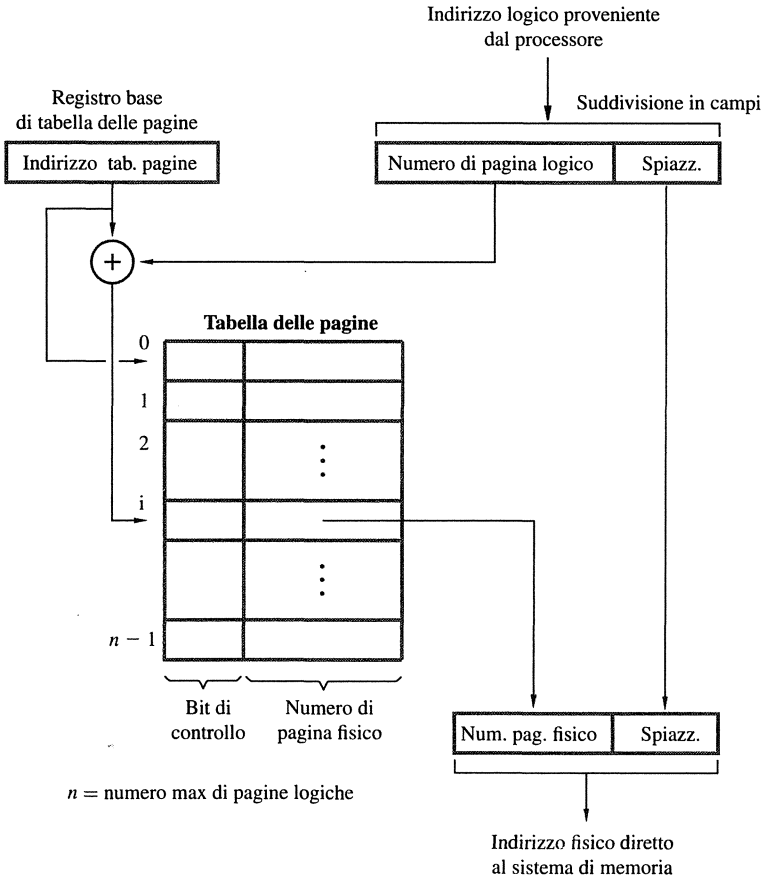


Figura 8.25 Schema di traduzione di indirizzo di memoria virtuale tramite tabella delle pagine.

Il campo spiazzamento indica semplicemente la posizione della parola di interesse all'interno della pagina (non diversamente da quanto succede con la posizione di memoria cache). La *tabella delle pagine* (*page table*) contiene invece la corrispondenza tra numero di pagina logico e collocazione effettiva della pagina in memoria primaria. In realtà per ogni pagina la tabella deve contenere un certo insieme di informazioni comprendenti l'indirizzo fisico iniziale della pagina e alcuni bit aggiuntivi per qualificarne lo stato. La regione di memoria capace di contenere esattamente una pagina si chiama *area di pagina* (*page frame*), o semplicemente pagina se non c'è pericolo di confusione.[11]

La tabella delle pagine sta anch'essa in memoria e l'indirizzo iniziale è conservato nel *registro base di tabella delle pagine* (*page table base register*). Addizionando il campo numero

[11] Dunque pagina sta a blocco (di cache) come area di pagina sta a posizione (di cache).

di pagina logico estratto dall'indirizzo della parola di interesse, generato dal processore, al contenuto del registro base di tabella delle pagine si ottiene l'indirizzo effettivo della riga della tabella dove stanno le informazioni relative alla pagina che si vorrebbe usare, quella cui appartiene la parola di interesse; in particolare, l'elemento essenziale contenuto nella riga è l'indirizzo fisico dell'area dove correntemente si trova caricata la pagina, ovvero il *numero di pagina fisico* (*physical page number*). Concatenandolo al campo spiazzamento estratto dell'indirizzo logico si ottiene l'indirizzo fisico della parola di interesse. Si riesamini la Figura 8.25 per ripercorrere tutto il processo di traduzione.

Ogni riga della tabella delle pagine contiene anche alcuni bit di controllo che qualificano lo stato della pagina. Uno di essi segnala se la pagina sia valida, cioè se l'area di pagina corrispondente abbia contenuto significativo. Disattivando tale bit, il sistema operativo di fatto estromette la pagina da memoria primaria, pur senza bisogno di cancellare fisicamente il contenuto dell'area. Un altro segnala se il contenuto dell'area sia stato modificato durante il soggiorno in memoria e diverga dalla pagina originale (cioè dal contenuto originariamente caricato nell'area), che sta altrove nella memoria secondaria. Come succedeva per la memoria cache, il bit serve per stabilire se al momento dell'estromissione da memoria primaria, per esempio quando occorre fare spazio per altre pagine, la pagina espulsa vada ricopiata in memoria secondaria su disco sovrascrivendone l'originale. Altri bit ancora possono segnalare restrizioni di vario tipo imposte all'uso concesso per la pagina; per esempio dare al programma il permesso completo di leggerla o scriverla oppure imporre qualche vincolo come sola lettura, sola scrittura e simili.

Tabella di traduzione

L'unità di gestione della memoria fa uso della tabella delle pagine ogniqualvolta il processore genera l'indirizzo e comanda l'operazione di memoria, non importa se di lettura o scrittura, pertanto in via ideale la tabella dovrebbe stare nella MMU stessa, per ragioni di efficienza. Tuttavia la tabella può raggiungere dimensione piuttosto grande e siccome di solito la MMU è integrata sullo stesso componente ospitante il processore (come già l'unità di cache primaria) è improponibile tenere lì la tabella completa; infatti di norma la tabella delle pagine risiede in memoria centrale. Tuttavia una porzione limitata può anche stare nella MMU e precisamente le righe corrispondenti alle pagine che figurano come utilizzate più di frequente dall'istante presente andando a ritroso nel tempo per una durata determinata (definita dal sistema operativo). La MMU contiene una memoria cache speciale di capacità limitata destinata a tale scopo; essa si chiama *Translation Lookaside Buffer* o TLB, vale a dire *tabella di traduzione*. Il ruolo della tabella TLB nei confronti della tabella delle pagine completa residente in memoria centrale è essenzialmente lo stesso di quello della memoria cache nei confronti della memoria centrale tutta intera. La tabella TLB contiene un sottoinsieme delle righe della tabella delle pagine. La riga di TLB è identica alla riga di tabella delle pagine (come quella, contiene il numero di pagina fisico e i bit di controllo), ma in aggiunta deve anche contenere il numero di pagina logico che nella tabella delle pagine figura invece implicitamente come indice di riga. La Figura 8.26 mostra lo schema di traduzione di indirizzo con tabella TLB. Va da sé che nella tabella TLB la riga di interesse vada ricercata in modo associativo e dunque in parallelo tra tutte le righe ivi presenti. Le unità di gestione di memoria commerciali sono dotate di tali tabelle TLB costituite da memoria veloce di tipo associativo.

È essenziale assicurare che il contenuto di TLB sia sempre lo stesso del contenuto della tabella delle pagine in memoria. Quando il sistema operativo cambia il contenuto nella tabella delle pagine, deve simultaneamente invalidare i corrispondenti valori in TLB. Uno dei bit di controllo nella TLB viene fornito per questo scopo. Quando un valore è invalidato, la TLB

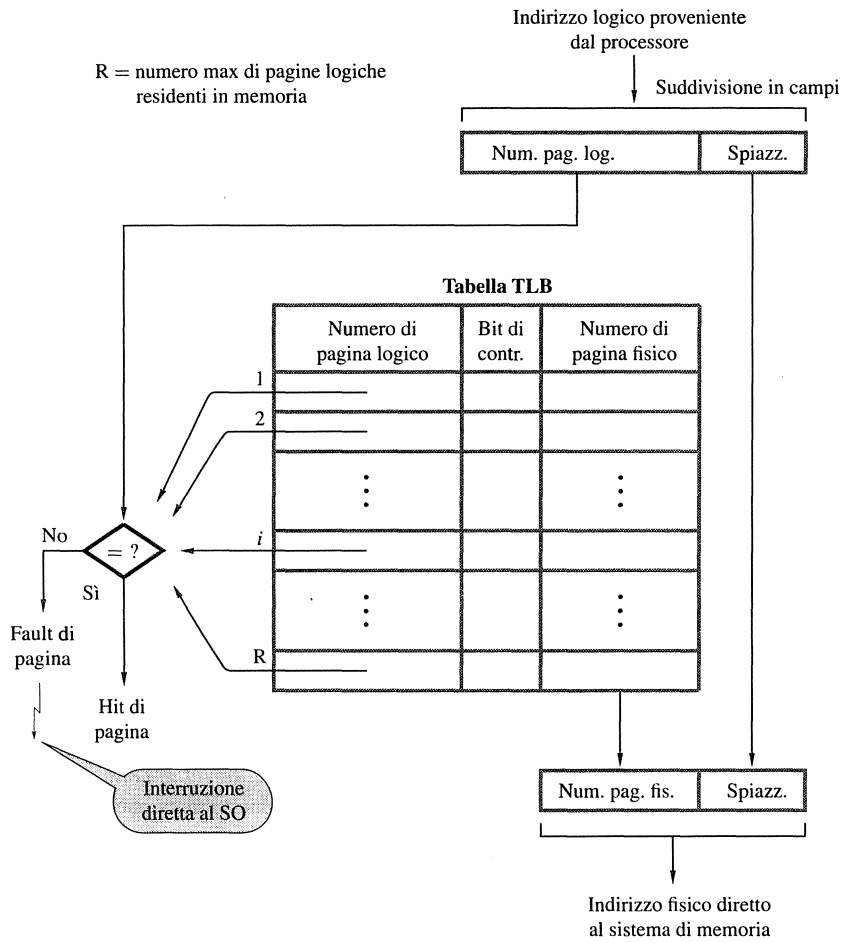


Figura 8.26 Traduzione di indirizzo tramite tabella TLB.

acquisisce la nuova informazione dalla tabella delle pagine in memoria come parte della risposta normale della MMU nel caso di miss di accesso.

Fault di pagina

La traduzione dell'indirizzo logico in fisico procede nel modo seguente. Dato l'indirizzo logico l'unità di gestione di memoria separa il campo numero di pagina logico e lo ricerca in parallelo nella tabella TLB. Se lo trova estrae dalla riga corrispondente il numero di pagina fisico e concatenandolo al campo spiazzamento ottiene l'indirizzo fisico della parola di interesse. Altrimenti si ha evento di *fault di pagina*, la MMU (o per conto di essa il sistema operativo) passa a esaminare la tabella delle pagine residente in memoria centrale (e che essendo completa deve contenere l'informazione cercata) e aggiorna la tabella TLB di conseguenza.

Si ha evento di fault di pagina ogniqualvolta l'istruzione macchina tenta di avere accesso a una parola che non si trova in memoria primaria. Ciò causa interruzione ancora prima che l'istruzione si sia potuta completare; essa del resto non potrebbe affatto completarsi giacché le manca la parola (dato) da elaborare. Quando dunque il processo sospeso torna in esecuzione, o il processore è in grado di riprendere l'esecuzione dell'istruzione dal punto esatto dove essa è stata lasciata in sospeso o l'istruzione va rieseguita daccapo.[12] La prima soluzione è piuttosto raffinata e complicata da gestire, la seconda più rudimentale ma più semplice; processori diversi adottano l'una o l'altra secondo le scelte del progettista.

Se si carica da disco la pagina nuova quando la memoria centrale è piena bisogna sostituire una delle pagine residenti (in memoria primaria). Come fare tale scelta è lo stesso problema che già si è visto per la memoria cache ed è altrettanto cruciale, e anche qui vale la proprietà di località. Siccome però la memoria primaria è ben più capace della memoria cache, è ragionevolmente possibile tenere in memoria primaria una parte relativamente grande del programma intero. In tale modo si riesce a ridurre la frequenza di trasferimento tra memoria centrale e disco. Per scegliere quale pagina sostituire si possono applicare concetti simili a quelli implicati nell'algoritmo LRU (già visto applicato alla cache); i bit di controllo ausiliari tenuti nella tabella delle pagine danno le informazioni necessarie ad alimentare l'algoritmo di sostituzione. Uno schema semplice si basa sul cosiddetto *bit di uso* (*use bit*): esso viene attivato ponendolo a 1 ogniqualvolta il processore lavora con la pagina corrispondente (lettura o scrittura); il sistema operativo lo disattiva (riportandolo a 0) in modo casuale ma comunque abbastanza di frequente (un'interruzione ripetitiva ma casuale si incarica di ciò); le pagine che generalmente hanno bit di uso disattivo sono chiaramente quelle che recentemente *non* hanno goduto di utilizzo rilevante da parte del processore e dunque sono le candidate naturali all'estromissione da memoria primaria; in modo complementare le pagine con bit di uso generalmente attivo formano l'*insieme di lavoro* (*working set*), la collezione di pagine correntemente in forte utilizzo.

La pagina modificata rispetto alla versione originale residente in memoria secondaria va ricopiata su disco prima di estrometterla da memoria primaria. Va chiarito che la gestione in modo di scrittura immediata, possibilissima e anzi utile per la memoria cache, è improponibile per la memoria virtuale: il tempo di accesso del disco è talmente elevato che non ha senso pretendere di andare a ricopiare con frequenza su disco piccoli quantitativi di dati (al limite una sola parola per volta!).

Il processo di traduzione dell'indirizzo logico in fisico richiede un certo intervallo di tempo, che essenzialmente dipende dal tempo di ricerca associativa nella tabella TLB. Ma grazie alla proprietà di località, con probabilità elevata numerose traduzioni consecutive generano indirizzi fisici all'interno della stessa area di pagina. Ciò riesce particolarmente evidente nel prelievo dell'istruzione macchina.[13] Ne viene che si può ridurre il tempo medio di traduzione dotando l'unità MMU di uno o più registri speciali dove mettere i numeri di pagina logico e fisico delle traduzioni effettuate più spesso in tempo recente, e il contenuto di tali registri è accessibile in tempo più breve rispetto a fare la ricerca associativa in tutta la tabella TLB.

---

[12] È bene ricordare (Capitolo 3) che l'interruzione normale da periferica lascia completare l'istruzione corrente prima di intercettare il flusso di esecuzione deviandolo sulla routine di servizio; ma l'interruzione normale non è critica come l'evento di fault di pagina.

[13] Questo equivale a dire che in memoria virtuale la forma temporale della località è meglio verificata di quella spaziale; motivo: il programma trascorre la maggior parte del tempo eseguendo i cicli più interni, che sono quelli più brevi e che dunque possono stare in una sola pagina; è meno comune invece che il programma lavori su quantità di dati piccole; attenzione: il ragionamento non vale anche per la memoria cache, a motivo del grande divario di dimensione tra blocco di memoria cache e pagina di memoria virtuale.

8.9 Gestione della memoria

In tutta la discussione precedente intorno al meccanismo di memoria virtuale si è sempre fatta l'ipotesi tacita che ci fosse un solo programma di dimensione grande in esecuzione. Se il processo (o programma in esecuzione) non sta per intero nella memoria centrale fisicamente installata nel calcolatore, parte di esso risiede in memoria secondaria su disco e quando serve viene caricato in memoria centrale sotto forma di pagine. Sebbene si sia vagamente alluso all'esistenza di routine di sistema che governano tutta la movimentazione di pagine di processo, non si è ancora detto nulla di preciso al riguardo.

Le routine di gestione di memoria sono uno dei componenti principali del sistema operativo. Il sistema operativo stesso è un programma o un complesso di programmi residenti in uno spazio di indirizzamento riservato, chiamato *spazio* (di indirizzamento) *di sistema* (*system addressing space*); esso è indipendente dallo spazio di indirizzamento dove risiede il processo applicativo dell'utente, chiamato *spazio* (di indirizzamento) *di utente* (*user addressing space*). In effetti ogni processo applicativo ha spazio di utente riservato per sé indipendente da quelli degli altri processi. Si gestisce tutto ciò efficacemente prevedendo una tabella delle pagine per ogni processo utente. L'unità di gestione di memoria si serve del registro di base di tabella delle pagine per tenerne l'indirizzo iniziale; modificando il contenuto del registro, il sistema operativo è in grado di passare da una tabella all'altra e dunque di commutare da uno spazio all'altro: è la *commutazione di contesto* (*context switch*). Si vede dunque come la memoria centrale fisicamente installata sia spartita per aree di pagina tra spazio di sistema e numerosi spazi di utente. Ma in ogni istante di tempo solo uno di tali spazi è attivo e solo le rispettive pagine sono accessibili.

In qualsiasi sistema di calcolo dove convivano più processi utente concorrenti e coesistenti nella stessa memoria fisica occorre prevedere e realizzare una qualche forma di *protezione*. Come minimo a nessun processo deve essere consentito di avere accesso a pagine di proprietà di altri processi e magari di distruggerne codice o dati. Tale forma di confinamento tra processi è ottenibile in vari modi. Quello più semplice si basa sul fatto che il processore ha due modi di funzionamento: *modo di sistema* (o *di supervisore*) e *modo di utente*. Come suggerisce il nome, il processore è sempre in modo di sistema quando esegue le routine di sistema operativo e in modo di utente quando esegue il codice ordinario del processo applicativo. In modo di utente alcune istruzioni macchina sono vietate. Tali *istruzioni privilegiate*, che naturalmente comprendono operazioni come la modifica del registro di base di tabella delle pagine, sono eseguibili da parte del processore solo quando funziona in modo di sistema. Così il processo applicativo non ha mezzi per avere accesso alla tabella delle pagine e non riesce a invadere lo spazio di altri processi applicativi o del sistema operativo stesso.

In determinate circostanze sarebbe utile e perfino desiderabile che due o più processi applicativi condividessero alcune pagine: è la funzione di *memoria condivisa* (*shared memory*). Il sistema operativo può gestire la memoria condivisa registrando entro tutte le tabelle delle pagine dei processi interessati le informazioni di accesso relative alle pagine in comune. Ciascun processo può fare uso differenziato della pagina condivisa impostando in modo differente i bit di controllo nella rispettiva tabella; per esempio, a un processo potrebbe essere concessa lettura e scrittura della pagina condivisa, a un altro solo lettura e così via. Per approfondimenti intorno al ruolo del sistema operativo in relazione alla memoria virtuale si rimanda alla bibliografia [5, 6].

8.10 Memoria di massa

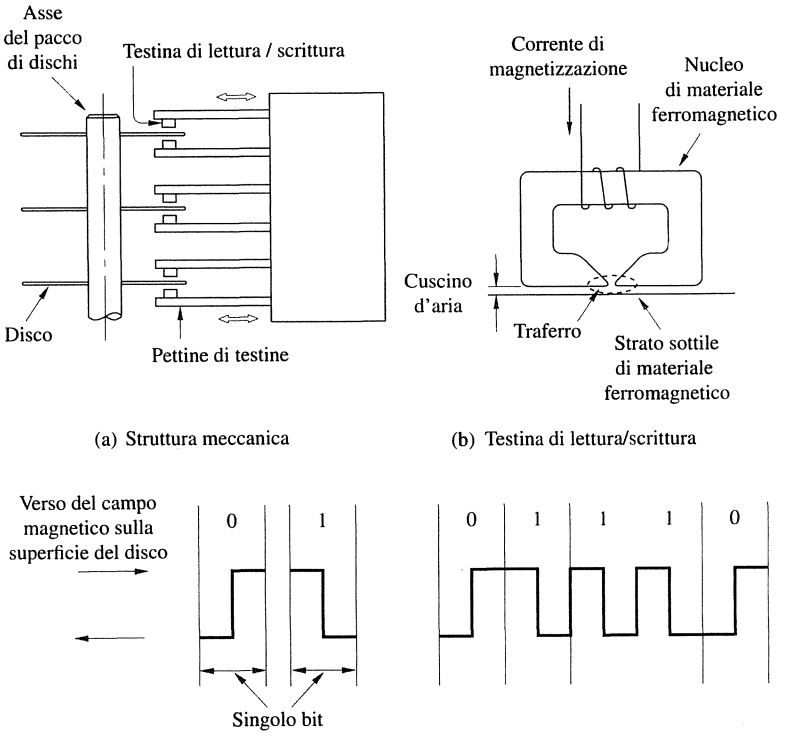
La memoria di massa costituisce lo strato inferiore della gerarchia o piramide di memoria (memoria secondaria), come mostra la Figura 8.24. La funzione della memoria di massa è

quella di ospitare la partizione di memoria aggregata al sistema di memoria virtuale, ma anche di archiviare in modo persistente programmi e dati. Le tecnologie di memoria di massa sono numerose: disco magnetico, nastro magnetico, disco ottico e altre ancora. Qui si daranno un'introduzione a struttura e funzionamento del disco magnetico e del disco ottico, le forme di memoria di massa più diffuse e versatili, e brevi cenni sul nastro magnetico.

8.10.1 Disco magnetico

Il *disco magnetico* (*magnetic disk*) è la tecnologia di memoria di massa più diffusa, perché è relativamente veloce, poco ingombrante (se non ha capacità elevatissima), può raggiungere capacità molto elevata ed è relativamente affidabile.

La Figura 8.27 mostra il principio tecnologico di funzionamento del disco magnetico. In Figura 8.27a si vede che il disco è un piatto o una schiera di piatti rotanti, rivestito da uno straterello di materiale ferromagnetico. Una testina di lettura e scrittura sorvola a breve distanza la superficie del disco percorrendo un cerchio relativamente a essa. Un motore sposta radialmente la testina per cambiare il raggio del cerchio quando serve. La velocità di rotazione



(a) Struttura meccanica

(b) Testina di lettura/scrittura

(c) Codifica dei bit nella traccia

Figura 8.27 Aspetti della tecnologia di disco magnetico.

va da pochi giri al minuto fino a circa diecimila o anche oltre. Con vari sistemi si assicura che la testina non tocchi la superficie per non danneggiarla (a bassa velocità può anche essere a contatto).

Il cerchio descritto dalla testina mentre il disco ruota si chiama *traccia*. La traccia memorizza una sequenza di bit in base agli stati di magnetizzazione della superficie del disco. Mentre il disco ruota, la traccia scorre in corrispondenza della testina e può essere letta o scritta. Il motore annesso alla schiera di testine le può spostare tra le tracce, secondo dove occorra leggere o scrivere. Comunque per leggere o scrivere un segmento di traccia, bisogna attendere che questo, ruotando, scorra in corrispondenza della testina. Alcuni tipi di dischi ad alta prestazione prevedono due o più schiere di testine intervallate ad angoli regolari e capaci di lavorare in parallelo, per ridurre tale tempo di attesa.

La Figura 8.27b mostra come è fatta la testina di lettura e scrittura: si tratta di una piccola bobina elettrica, costituita da nucleo metallico con traferro (dove si sviluppa il campo magnetico) e avvolgimento elettrico. Se l'avvolgimento è percorso da corrente, il nucleo magnetizza (a seguito della legge di Oersted) la traccia sottostante il traferro secondo come la corrente stessa è modulata per rappresentare la sequenza di bit che si desidera scrivere. Viceversa, mentre il nucleo percorre la traccia magnetizzata nell'avvolgimento, viene indotta (a seguito della legge di Faraday-Neumann) tensione elettrica e di conseguenza corrente che, opportunamente amplificata, corrisponde alla sequenza di bit memorizzati sulla traccia che si desidera leggere. La testina è governata da un circuito apposito, in parte analogico e in parte digitale, che rileva o amplifica la corrente nell'avvolgimento e la demodula o modula secondo la sequenza di bit da leggere o scrivere, rispettivamente. Solo i cambiamenti del campo magnetico sotto testina possono essere rilevati durante un'operazione di lettura. Pertanto, se i valori logici 0 e 1 sono rappresentati da due stati di magnetizzazione opposti, viene indotta una tensione nell'avvolgimento solo alle transizioni 0-1 e 1-0 nella sequenza binaria. Una lunga sequenza di valori uguali induce tensione solo all'inizio e alla fine della sequenza. È pertanto necessario un segnale periodico di sincronizzazione per determinare il numero di bit uguali consecutivi immagazzinati.

L'approccio moderno combina l'informazione di sincronizzazione con i dati. Una tecnica semplice, nota come *codifica di fase* (*phase encoding*) o *codifica di Manchester* (*Manchester encoding*), è illustrata in Figura 8.27c, che mostra come le transizioni di stato di magnetizzazione (ovvero le transizioni di segnale elettrico indotto) siano usate per codificare la sequenza di bit: i fronti di salita e discesa codificano il bit 0 e 1, rispettivamente; fra bit uguali adiacenti va inserito un fronte addizionale. Qui sta lo svantaggio di questa tecnica: la rappresentazione di un bit richiede spazio per due variazioni di stato di magnetizzazione, il che limita la densità di informazione sul disco. La codifica di Manchester è un semplice esempio di codifica autosincronizzante (*self-clocking*). Sono state ideate tecniche autosincronizzanti più efficienti, che realizzano codifiche più compatte usando circuiti più complessi. La discussione di tali tecniche esula dagli scopi di questo libro.

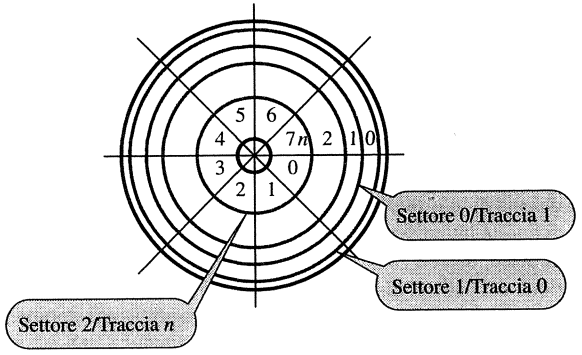
Nella maggior parte delle unità a dischi moderne, i dischi e le testine di lettura e scrittura sono sigillati in un involucro privo di aria. In questo approccio, noto come *tecnologia Winchester*, le testine possono operare a distanza minore dalla superficie magnetizzata, in virtù dell'assenza di particelle di polvere. Ciò permette di ridurre sia lo spazio necessario a rappresentare un bit in una traccia sia la distanza fra tracce adiacenti. A parità di ingombro, la capacità dei dischi Winchester supera quella dei dischi non sigillati. Un altro vantaggio della tecnologia Winchester è la migliore garanzia di integrità dei dati assicurata dalla non esposizione del supporto di memorizzazione a elementi contaminanti.

Un sistema a dischi consta essenzialmente di tre parti. Una è il piatto o schiera di piatti, genericamente indicata come il *disco*. La seconda parte, detta *pilota del disco* (*disk drive*) è costituita dal dispositivo elettromeccanico che fa ruotare il disco e sposta le testine di lettura

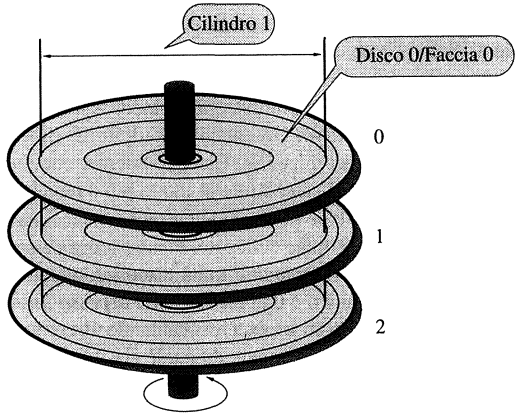
e scrittura. La terza parte è il *controllore del disco* (*disk controller*), ovvero il circuito elettronico che controlla il funzionamento del sistema. Il controllore del disco può essere realizzato separatamente oppure incorporato nell'involucro che contiene l'intero sistema a dischi. Con il termine "disco" spesso ci si riferisce alla combinazione del pilota del disco e del disco ivi contenuto. Si adotta questa convenzione anche qui nel seguito, laddove ciò non sia fonte di ambiguità.

Organizzazione e accesso ai dati su un disco

L'organizzazione dei dati su un disco è illustrata in Figura 8.28. Ciascuna faccia del disco è suddivisa in un certo numero di *tracce* concentriche, numerate a partire dall'interno, e ogni traccia (sequenza di bit) è suddivisa in un certo numero di *settori*, come mostra la Figura 8.28a.



(a) disco a faccia singola



(b) pacco di dischi

Figura 8.28 Struttura del disco magnetico

Ogni settore è a sua volta una sequenza di bit (tipicamente qualche centinaio o migliaio) ed è numerato all'interno della traccia di appartenenza. Dando il numero di traccia e di settore si identifica univocamente il settore sul disco. Il settore è la *minima unità di dati leggibile o scrivibile* (non si può leggere o scrivere un solo byte, ma l'intero settore).

Se il disco è costituito da una schiera (o pacco) di piatti numerati, ciascuno con due facce utilizzabili, per identificare il settore bisogna anche dare il numero di faccia, come mostra la Figura 8.28b. L'insieme di tutte le tracce poste alla medesima distanza dal centro del piatto rispettivo (e dunque numerate in modo identico) va sotto nome di *cilindro*. Per identificare il settore nell'intera schiera di piatti si dà il numero di cilindro, il numero di piatto (o di faccia) e il numero di settore nella traccia.

I bit dei dati sono memorizzati sequenzialmente su ciascuna traccia. Ogni settore può contenere 512 o più byte. I dati sono preceduti da una *intestazione di settore* (*sector header*) che contiene l'informazione di identificazione (indirizzamento) utile a reperire il settore desiderato nella traccia selezionata. In coda ai dati del settore sono posti dei bit addizionali che ne costituiscono il *codice di correzione di errore* o ECC (*Error-Correcting Code*), come spiegato nell'Approfondimento applicativo 1.1. Segue infine un breve lasso di separazione intersettoriale (*inter-sector gap*), che permette ai circuiti di controllo di distinguere facilmente settori consecutivi.

La Figura 8.28a indica che ogni traccia consta dello stesso numero di settori, il che implica che tutte le tracce abbiano la stessa capacità di memoria. In tal caso, le tracce più interne hanno maggiore densità di informazione che le tracce più esterne. Si può accrescere la densità di informazione di queste ultime aumentandone il numero di settori. Tale organizzazione dell'informazione sul disco, detta *a zone*, impiega circuiti di controllo più complessi.

L'organizzazione dei dati sul disco prevede che uno o più settori consecutivi sulla stessa traccia vadano considerati come una sola unità di dati, leggibile e scrivibile in modo indivisibile: il *blocco* di disco. Il numero (intero) di settori costituenti un blocco si chiama *fattore di blocco* e in genere è maggiore di uno. Per il sistema operativo, il disco (o l'insieme di tutte le unità disco) è una memoria indirizzabile per blocchi e ciascuno di essi è identificato da un numero di blocco univoco: l'operazione fondamentale consiste nel leggere o scrivere un blocco intero. La strutturazione del disco in blocchi può essere gestita direttamente dal controllore di disco o demandata al sistema operativo, dipende da quanto il controllore stesso è sofisticato. Pertanto il controllore di disco può contenere i circuiti capaci di gestire le due operazioni fondamentali ed eventualmente di convertire il numero di blocco negli indirizzi cilindro/disco(faccia)/settore dei vari settori consecutivi che costituiscono il blocco; spesso però tale conversione è svolta in software dal sistema operativo. Ha anche il compito di rilevare le situazioni di errore, di segnalarle e se possibile di correggerle. Essenzialmente, comunque, il controllore di disco ha il compito di interfacciarsi al calcolatore in modo più o meno standard e di offrire al SO una visione del disco o dell'insieme di dischi abbastanza svincolata dai dettagli tecnologici e operativi delle unità disco specifiche collegate, mentre lettura e scrittura effettive del singolo settore sono svolte da parte del pilota dell'unità disco interessata.

Complessivamente il disco magnetico è una memoria ad *accesso diretto*, cioè si può accedere (in lettura o scrittura) a un blocco a piacere mandando il comando e il numero di blocco al controllore del disco, indipendentemente da quale sia il blocco utilizzato nell'operazione immediatamente precedente. La strutturazione effettiva in blocchi viene stabilita al momento della *formattazione* del disco.

Un disco non formattato non ha informazione nelle sue tracce. Il processo di formattazione scrive i marcatori che suddividono il disco in tracce e settori. Durante questo processo il controllore del disco può rilevare che alcuni settori o persino intere tracce sono difettose. In tal caso il controllore registra i difetti rilevati e li esclude dall'uso. L'informazione di formattazione comprende le intestazioni di settore, i bit ECC e i gap intersettoriali. La capacità di un disco formattato, al netto dell'informazione di formattazione, è la misura appropriata della capacità di un disco.

Dopo la formattazione, un disco magnetico singolo, o un insieme di dischi sufficientemente simili, è strutturabile in una o più *partizioni* (*partition*). Ciascuna partizione contiene una quantità stabilita di blocchi numerati sequenzialmente, in modo indipendente. Il sistema operativo può dedicare una partizione di disco (o anche più d'una) come componente del sistema di memoria virtuale. In tale caso, la partizione di disco funziona come strato inferiore della piramide di memoria; al riguardo si vedano il Paragrafo 8.5 e, in particolare, la Figura 8.14. Dato che la partizione di disco integrata nel sistema di memoria virtuale è già organizzata in blocchi indirizzabili e accessibili individualmente, come lo sono memoria centrale e cache, la gestione è molto simile e uniforme. Se il blocco di disco ha dimensione diversa da quella del blocco di memoria centrale e cache, il SO si incarica di gestire l'adattamento a livello software. Si ricorda comunque che la funzione di memoria virtuale è facoltativa e volendo la si può anche disattivare.

La maggior parte della memoria di massa è usata semplicemente come archivio di dati e programmi. Il sistema operativo può dedicare la partizione di disco come *file system*. Il file system è un insieme strutturato (in cartelle ecc.) di file contenenti dati e programmi. Di per sé il file system non appartiene alla memoria virtuale, avendo scopo diverso. Talvolta però, invece di avere una partizione specifica integrata nella memoria virtuale, il SO utilizza per lo stesso scopo un file apposito, di dimensioni prefissate e riservato per tale uso. Vi sono vari modi di organizzare il file system, secondo il tipo di SO Tale argomento è di pertinenza della teoria dei SO e qui non interessa; al riguardo si veda la bibliografia [5, 6].

Tempo di accesso

Il ritardo che intercorre fra la ricezione di un indirizzo da parte del pilota del disco e l'inizio del trasferimento dei dati ha due componenti. Il primo, detto *tempo di posizionamento* o di ricerca (*seek time*), è il tempo necessario per spostare la testina di lettura e scrittura sulla traccia selezionata. Questo tempo dipende dalla posizione iniziale della testina relativa alla traccia specificata nell'indirizzo. I suoi valori medi cadono tipicamente nell'intervallo [5, 8] ms. Il secondo componente è la *latenza rotazionale*, o più brevemente latenza (*latency time*), cioè il tempo necessario a raggiungere il settore indirizzato dopo che la testina di lettura e scrittura sia stata posizionata sulla traccia desiderata. In media, la latenza è la metà del tempo di una rotazione del disco. La somma di questi due ritardi è detta *tempo di accesso* al disco (*access time*). Se si accede a pochi settori in una data operazione, il tempo di accesso supera quello di trasferimento dei dati per almeno un ordine di grandezza. I modelli di disco magnetico sono numerosissimi: flessibile (capacità limitata) e rigido (capacità elevata); fisso o rimovibile; lento (velocità di rotazione bassa, testina a contatto) o veloce (velocità di rotazione alta, testina sospesa). La capacità del disco magnetico (eventualmente a schiera) arriva a parecchie centinaia o migliaia di Gbyte. Il tempo di accesso per leggere o scrivere un blocco va da qualche secondo per i dischi flessibili e piccoli a circa un millisecondo (o anche un po' meno) per i dischi a prestazioni elevatissime. Come tempo caratteristico di un disco rigido di media qualità si possono considerare circa 10 millisecondi o poco meno.

Buffer dei dati e cache

Il controllore di disco ad alte prestazioni può essere dotato di un *buffer dei dati*, ovvero una certa quantità di memoria locale a semiconduttori (tipicamente qualche Mbyte). Le unità disco più veloci possono trasferire dati con continuità a un tasso di decine o centinaia di Mbyte al secondo, e spesso per raggiungere tale prestazione sono collegate al calcolatore tramite bus di periferia appositi, quali SCSI o SATA, o via rete ad alta velocità, e impiegano il buffer dei dati come cache, contenente copie dei blocchi di disco cui si accede con frequenza maggiore.

Si tratta dello stesso principio su cui si basa la cache del processore, e anche la gestione è simile pur con gli adattamenti opportuni. Lo scopo è quello di diminuire il tempo di accesso apparente al disco. Simili tecnologie di disco magnetico ad altissima prestazione (e costo relativamente elevato) si usano in calcolatori di tipo server. Per approfondimenti si rimanda alla bibliografia [5, 6].

Controllore del disco

L'unità disco è governata dal pilota del disco, che riceve comandi e scambia dati con un controllore del disco, il quale a sua volta comunica con il calcolatore. Una o più unità disco sono collegabili al calcolatore tramite un controllore, che di fatto è un'interfaccia di I/O. Il controllore accede al bus del calcolatore, vale a dire il bus di sistema o eventualmente un bus di I/O, se il calcolatore ne fosse dotato, come il bus PCI. Il controllore comunica con il calcolatore tramite una delle tecniche di I/O viste nel Capitolo 3: controllo di programma (solo per dischi piccoli e lenti), interruzione o DMA (accesso diretto alla memoria); di solito i dischi con prestazione elevata abbisognano di controllore funzionante in DMA, e tipicamente con interruzione a fine trasferimento.

Sul lato unità disco, il controllore comunica con il pilota dell'unità stessa tramite una varietà di bus di periferia standard specializzati: IDE, SCSI e altri ancora (si veda il Capitolo 7). I comandi eseguiti dal pilota riguardano la schiera di testine e il disco stesso, azionandone i motori, e il circuito di controllo della testina di lettura e scrittura. Tipicamente il pilota può ricevere (dal controllore) il comando di lettura o scrittura di un settore (o di un blocco) e l'indirizzo cilindro/disco(faccia)/settore del settore (iniziale) interessato, azionare la testina in modo opportuno ed effettuare la lettura o scrittura richiesta. Può anche effettuare un posizionamento della testina e può verificare la presenza di errori codificando in modo opportuno il campo ECC del settore.

Disco flessibile

I dischi discussi sopra sono noti come unità disco rigido (*hard disk*). Il *disco flessibile* (*floppy disk*) è un supporto più piccolo, più semplice e meno costoso, costituito da un dischetto di plastica, flessibile e rimovibile, ricoperto di materiale magnetico. Il dischetto è racchiuso in un involucro di plastica con un'apertura su cui si posiziona la testina di lettura e scrittura. Un foro centrale nel dischetto permette a un alberino nel pilota di posizionare e fare ruotare il dischetto.

Le caratteristiche principali dei dischi flessibili sono il basso costo e la facilità di distribuzione. Tuttavia, rispetto ai dischi rigidi hanno capacità di memoria molto più piccola, tempi di accesso più lunghi e tassi di errore più alti. Negli anni più recenti sono stati largamente rimpiazzati da CD, DVD e schede flash, quali supporti di memoria rimovibili.

Schiere di dischi RAID

Le velocità dei processori sono aumentate notevolmente. Allo stesso tempo, i tempi di accesso alle unità disco sono ancora dell'ordine dei millisecondi, a causa delle limitazioni del movimento meccanico coinvolto. Un modo per ridurre il tempo di accesso è di usare dischi multipli che funzionano in parallelo. Nel 1988, i ricercatori dell'Università di California-Berkeley proposero un tale sistema di immagazzinamento [7]. Lo chiamarono RAID, che sta per *Redundant Array of Inexpensive Disks*, ovvero *schiera ridondante di dischi economici* (poiché adesso tutti i dischi sono economici, l'acronimo fu successivamente reinterpretato come *schiera ridondante di dischi indipendenti*). L'uso di dischi multipli permette di migliorare l'affidabilità del sistema complessivo. Furono proposte diverse configurazioni, e molte altre sono state da allora sviluppate.

La configurazione di base, nota come RAID 0, è semplice. Un singolo file, se grande, è immagazzinato in varie unità disco separate, dividendo il file in frammenti più piccoli e immagazzinando questi frammenti su dischi diversi. Questa tecnica è detta *data striping* (ovvero stoccaggio dei dati a strisce). Quando si accede al file per un'operazione di lettura, tutti i dischi accedono alle loro porzioni dei dati in parallelo. Conseguentemente, il tasso con cui i dati possono essere trasferiti è uguale al tasso di trasferimento individuale dei dischi moltiplicato per il numero di dischi. Tuttavia, il tempo di accesso (tempo di posizionamento + latenza) necessario per localizzare l'inizio dei dati su ciascun disco non viene ridotto. Poiché ciascun disco funziona indipendentemente, i tempi di accesso variano. I diversi frammenti dei dati sono bufferizzati, così che il file completo può essere riassemblato e trasferito alla memoria come un'entità singola.

Varie configurazioni RAID formano una gerarchia, dove ciascun livello della gerarchia fornisce caratteristiche aggiuntive. Per esempio, RAID 1 è inteso fornire maggiore affidabilità immagazzinando copie identiche di dati su due dischi anziché solo uno. Si dice che i due dischi sono ciascuno lo specchio dell'altro. Se un'unità disco si guasta, tutte le operazioni di Read e Write vengono dirette verso l'unità specchio. Altri livelli della gerarchia ottengono maggiore affidabilità attraverso vari schemi di controllo della parità, senza richiedere la completa duplicazione dei dischi. Alcuni hanno anche capacità di recupero da errori.

Il concetto RAID ha guadagnato accettazione commerciale. Sistemi RAID sono disponibili da molti costruttori per essere usati con vari sistemi operativi.

8.10.2 Dischi ottici

Si possono realizzare dispositivi di immagazzinamento di dati anche usando mezzi ottici. Il familiare *Compact Disk* (CD), usato nei sistemi audio, fu la prima applicazione pratica di questa tecnologia. Subito dopo, la tecnologia ottica fu adattata all'ambiente del calcolatore per fornire un mezzo di immagazzinamento in sola lettura ad alta capacità conosciuto come CD-ROM.

La prima generazione di CD fu sviluppata alla metà degli anni '80 da Sony e Philips. La tecnologia si basava sulla possibilità di usare una rappresentazione digitale per segnali sonori analogici. Per ottenere registrazione e riproduzione del suono ad alta qualità, si prendono campioni da 16 bit del segnale analogico alla frequenza di 44.100 campioni al secondo. Inizialmente i CD furono progettati per contenere fino a 75 minuti di registrazione, richiedendo un totale di circa 3×10^9 bit (3 Gigabit) di spazio. Da allora sono stati sviluppati dispositivi a più alta capacità.

Tecnologia dei CD

La tecnologia ottica che è usata per i sistemi CD si basa sul fatto che la luce del laser può essere concentrata su un'area molto piccola. Un raggio laser è diretto su un disco rotante, che ha rientranze piccolissime organizzate a formare una lunga traccia a spirale sulla sua superficie. Le rientranze riflettono il raggio verso un fotorivelatore, che rileva le sequenze binarie immagazzinate.

Il laser emette un raggio di luce coerente concentrata con alta precisione sulla superficie del disco. La luce coerente consiste di onde sincronizzate che hanno la stessa lunghezza d'onda. Se un raggio di luce coerente è combinato con un altro raggio dello stesso tipo, e i due raggi sono in fase, il risultato è un raggio più luminoso. Ma se le onde dei due raggi sono in opposizione di fase, cioè sfasate di 180 gradi, si cancellano a vicenda. Quindi si può usare un fotorivelatore, che rileverà un punto luminoso nel primo caso e un punto oscuro nel secondo caso.

In Figura 8.29a è mostrata una sezione trasversale di una piccola porzione di un CD. Lo strato più in basso è fatto di materiale plastico policarbonato trasparente, che serve come base di vetro trasparente. La superficie di questa plastica è programmata per immagazzinare i dati praticandovi delle rientranze o *buche* (*pit*). Le parti non rientranti sono chiamate *terre* (*land*). Sul disco programmato viene disteso un sottile strato di alluminio riflettente, che è poi ricoperto da acrilico protettivo. Infine, si deposita lo strato più in alto e vi si stampa un'etichetta.

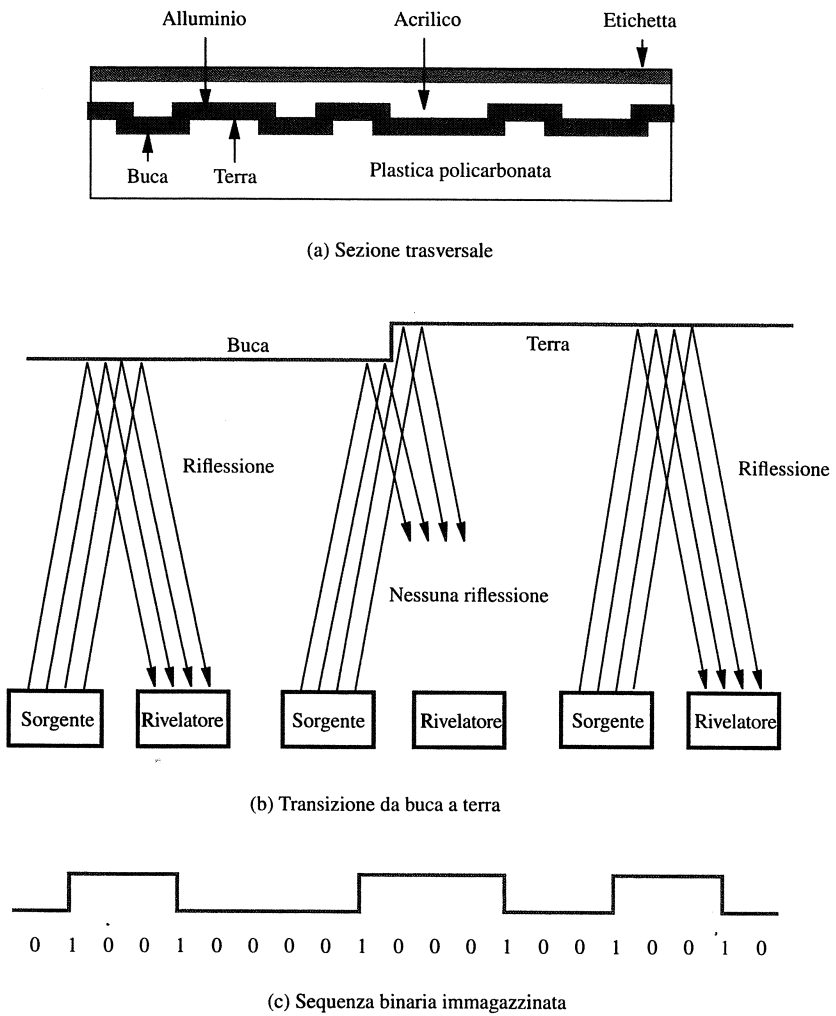


Figura 8.29 Disco ottico.

Lo spessore complessivo del disco è 1,2 mm, quasi tutto dovuto al policarbonato plastico. Gli altri strati sono molto sottili.

La sorgente laser e il fotorivelatore sono posizionati sotto il policarbonato plastico. I raggi emessi attraversano lo strato di plastica, vengono riflessi sullo strato di alluminio e viaggiano in ritorno al fotorivelatore. Si noti che dal lato del laser le buche appaiono in realtà come rilievi sulle terre.

La Figura 8.29b mostra cosa accade quando il raggio laser effettua la scansione del disco e incontra una transizione da buca a terra. Sono mostrate tre differenti posizioni della sorgente

laser e del rivelatore, come avviene quando il disco ruota. Quando la luce è riflessa solamente da una buca, o da una terra, il rivelatore vede il raggio riflesso come un punto luminoso. Ma una situazione diversa ha luogo quando il raggio passa sul bordo fra una buca e la terra adiacente. La buca è un quarto di lunghezza d'onda più vicina alla sorgente laser. Quindi i raggi riflessi dalla buca e dalla terra adiacente saranno in opposizione di fase, cancellandosi l'un l'altro. Quindi il rivelatore non vedrà un raggio riflesso durante le transizioni buca-terra e terra-buca, ma rileverà un punto oscuro.

La Figura 8.29c illustra varie transizioni fra terre e buche. Se si conviene che ciascuna transizione, rilevata come un punto oscuro, denoti il valore binario 1 e che le porzioni piatte rappresentino 0, allora la sequenza binaria rilevata sarà come mostrato in figura. Questa sequenza non è una rappresentazione diretta dei dati immagazzinati. I CD usano uno schema di codifica complesso per rappresentare i dati. Ciascun byte di dati è rappresentato da un codice a 14 bit, che permette una considerevole capacità di rivelazione di errori (si veda l'Approfondimento applicativo 1.1). Non si approfondiranno i dettagli di questo codice.

Le buche sono disposte su una lunga traccia spiraliforme sulla superficie del disco, che va dal centro del disco verso il bordo esterno. È però consuetudine riferirsi a ciascun percorso circolare di ampiezza 360 gradi come a una traccia separata, con terminologia analoga a quella usata per i dischi magnetici. Il CD ha un diametro di 120 mm, con un foro di 15 mm al centro. Le tracce coprono un'area da un raggio di 25 mm a un raggio di 58 mm. Lo spazio fra le tracce è 1,6 micron. Le buche sono larghe 0,5 micron e lunghe da 0,8 a 3 micron. Su un disco vi sono più di 15.000 tracce. Se l'intera traccia a spirale fosse srotolata, sarebbe lunga più di 5 km!

CD-ROM

Poiché i CD immagazzinano informazioni in forma binaria, risultano adatti a essere usati per l'immagazzinamento di dati nei calcolatori. La sfida principale è assicurare l'integrità dei dati immagazzinati. Poiché le buche sono molto piccole, è difficile realizzare tutte le buche in modo perfetto. Per l'uso in applicazioni audio e video, qualche errore nei dati è tollerabile, poiché è improbabile che il suono o l'immagine riprodotti ne risentano in modo percettibile. Tuttavia, tali errori non sono accettabili in applicazioni di calcolo. Poiché non si possono evitare le imperfezioni fisiche, è necessario usare bit aggiuntivi per fornire la capacità di rivelazione e correzione degli errori. I CD usati per immagazzinare dati dei calcolatori sono detti *CD-ROM*, poiché, come i chip ROM a semiconduttori, il loro contenuto può solo essere letto.

I dati immagazzinati sono organizzati su tracce del CD-ROM in forma di blocchi detti *settori*. Ci sono diversi formati di settore. Un formato, conosciuto come Modo 1, usa settori da 2.352 byte. Vi è un'intestazione di 16 byte che contiene un campo di sincronizzazione usato per rilevare l'inizio del settore e dell'informazione di indirizzamento che identifica il settore. Questa è seguita da 2.048 byte di dati immagazzinati. Alla fine del settore, vi sono 288 byte usati per realizzare lo schema di correzione di errore. Il numero di settori per traccia è variabile; vi sono più settori sulle tracce esterne più lunghe. Con il formato del Modo 1, un CD-ROM ha una capacità di immagazzinamento di circa 650 Mbyte.

La rivelazione e correzione di errore si effettua a più di un livello. Come detto prima, ciascun byte di informazione immagazzinato su un CD è codificato usando un codice da 14 bit che ha una certa capacità di correzione di errore. Questo codice può correggere errori su bit singolo. Errori che avvengono in piccole serie che coinvolgono più bit sono rivelati e corretti usando i bit di controllo di errore alla fine del settore.

I lettori di CD-ROM possono operare a differenti velocità di rotazione. La velocità base, nota come 1X, è 75 settori al secondo. Questa dà un tasso di trasferimento di dati di 153.600 byte/s (150 Kbyte/s), usando il formato Modo 1. I lettori di CD-ROM a velocità più alte sono identificati in rapporto alla velocità di base. Quindi un CD-ROM 56X ha un tasso di trasferimento di dati che è 56 volte quello del CD-ROM 1X, ovvero circa 8 Mbyte/s. Questo tasso

di trasferimento è considerevolmente più basso rispetto ai tassi di trasferimento dei dischi fissi magnetici, che sono dell'ordine di decine di Mbyte al secondo. Un'altra significativa differenza di prestazione è il tempo di accesso, che per i CD-ROM può essere di centinaia di millisecondi. Quindi in termini di prestazione, i CD-ROM sono nettamente inferiori rispetto ai dischi magnetici. La loro attrattiva consiste nella loro piccola dimensione fisica, basso costo e facilità d'uso come supporto di memoria di massa rimovibile e portabile. Di conseguenza, sono ampiamente usati per la distribuzione di software, libri, programmi, video-giochi e così via.

CD-Registrabili

I CD descritti sopra sono dispositivi a sola lettura, in cui le informazioni sono immagazzinate al momento della produzione. All'inizio viene prodotto un disco master usando un laser ad alta potenza per bruciare le buche richieste. Quindi viene prodotto un calco dal disco master, che ha rilievi al posto delle buche. Le copie vengono fatte iniettando nel calco plastica policarbonata liquefatta, così da produrre CD che hanno la stessa sequenza di buche del disco master. Questo processo è chiaramente adatto solo per produzioni in grandi quantità di CD che contengono le stesse informazioni.

Un nuovo tipo di CD, detto CD-Registrabile o CD-R (*CD-Recordable*), fu sviluppato alla fine degli anni '90, sul quale i dati possono essere facilmente registrati dall'utente di un calcolatore. Una traccia a spirale traslucida coperta da un colorante organico viene realizzata su un disco durante il processo di fabbricazione. Dopo, un laser in un'unità CD-R brucia le buche nel colorante organico. Le zone bruciate diventano opache. Esse riflettono meno luce rispetto alle aree traslucide quando il CD viene letto. Questo processo è irreversibile, ovvero i dati scritti sono immagazzinati permanentemente. Porzioni non utilizzate di un disco possono essere successivamente usate per immagazzinare dati aggiuntivi.

CD-Riscrivibili

I CD più flessibili sono quelli che possono essere scritti più volte dall'utente. Essi sono noti come CD-Riscrivibili o CD-RW (*CD-ReWritable*).

La struttura di base dei CD-RW è simile alla struttura dei CD-R. Invece di usare un colorante organico nello strato di registrazione, si impiega una lega di argento, indio, antimonio e tellurio. Questa lega ha un comportamento interessante e utile quando viene riscaldata e raffreddata. Se viene riscaldata sopra il suo punto di fusione (500 °C) e quindi raffreddata, essa va in uno stato amorfo in cui assorbe la luce. Tuttavia se viene riscaldata a soli circa 200 °C e questa temperatura viene mantenuta per un periodo esteso, avviene un processo noto come *ricottura* (*annealing*), che lascia la lega in uno stato cristallino che permette alla luce di passare attraverso. Se lo stato cristallino rappresenta l'area delle terre, le buche possono essere create riscaldando le parti selezionate oltre il punto di fusione. I dati immagazzinati possono essere cancellati usando il processo di ricottura, che fa tornare l'alluminio a uno stato cristallino uniforme. Un materiale riflettente viene posto sopra lo strato di registrazione per riflettere la luce quando il disco viene letto.

Un'unità CD-RW usa tre potenze di laser diverse. La potenza più alta viene usata per registrare le buche. La potenza di mezzo è usata per portare la lega al suo stato cristallino; viene detta "potenza di cancellazione". La potenza più bassa viene usata per leggere le informazioni immagazzinate.

Le unità CD progettate per leggere e scrivere dischi CD-RW possono di solito essere usate per altri supporti compact disk. Esse possono leggere CD-ROM e leggere e scrivere CD-R; sono progettate per soddisfare i requisiti di interfacce di interconnessione standard, come SATA e USB.

I dischi CD-RW sono supporti di memoria di massa a basso costo. Essi sono adatti per archiviare informazioni che possono andare da basi di dati a immagini fotografiche. Possono essere usati per bassi volumi di distribuzione di informazioni, proprio come i CD-R, e per la

creazione di copie di riserva (*back-up*). La tecnologia CD-RW ha ridotto la rilevanza dei CD-R poiché offre funzionalità superiori a un costo poco più alto.

Tecnologia DVD

Il successo della tecnologia dei CD e la continua richiesta per capacità di immagazzinamento più grandi ha portato allo sviluppo della tecnologia del disco versatile digitale o DVD (*Digital Versatile Disk*). Il primo standard DVD fu definito nel 1996 da un consorzio di aziende, con l'obiettivo di riuscire a immagazzinare un film di lunghezza normale su un lato del disco DVD.

La dimensione fisica del disco DVD è la stessa di quella dei CD. Il disco ha lo spessore di 1,2 mm e il diametro di 120 mm. La sua capacità di memoria è resa molto più grande di quella dei CD grazie a vari cambiamenti nella progettazione.

- Uso di un laser a luce rossa con una lunghezza d'onda di 635 nm invece del laser a luce infrarossa usato nei CD, che ha una lunghezza d'onda di 780 nm. La lunghezza d'onda più corta rende possibile collimare la luce su una zona più piccola.
- Le buche sono più piccole, ed hanno una lunghezza minima di 0,4 micron.
- Le tracce sono posizionate più vicine tra loro; la distanza fra le tracce è 0,74 micron.

L'uso di queste migliorie porta a una capacità dei DVD di 4,7 Gbyte.

Ulteriori aumenti di capacità sono stati ottenuti per mezzo di dischi a due strati e a due lati. Il disco a singolo strato e singolo lato, definito nello standard come DVD-5, ha una struttura che è quasi la stessa del CD in Figura 8.29a. Un disco a doppio strato usa due strati su cui le tracce vengono realizzate una sopra l'altra. Il primo strato è la base chiara, come nei dischi CD. Tuttavia, invece di usare alluminio riflettente, le terre e le buche di questo strato sono coperte da un materiale traslucido che agisce come un semi-riflettore. Anche la superficie di questo materiale è quindi programmata con buche rientranti per immagazzinare i dati. Un materiale riflettente viene posto sopra il secondo strato di buche e terre. Il disco viene letto focalizzando il raggio laser sullo strato desiderato. Quando il raggio viene focalizzato sul primo strato, dal materiale traslucido viene riflessa luce sufficiente per rilevare le sequenze binarie immagazzinate. Quando il raggio viene focalizzato sul secondo strato, la luce riflessa dal materiale riflettente corrisponde all'informazione immagazzinata su questo strato. In entrambi i casi, lo strato su cui il raggio non è focalizzato riflette una quantità di luce molto minore, che viene eliminata dal circuito di rilevazione come rumore. La capacità di immagazzinamento totale dei due strati è 8,5 Gbyte. Questo disco viene detto DVD-9 nello standard.

Due dischi a singolo lato possono essere messi insieme per formare una struttura a sandwich dove il disco superiore è rivoltato sottosopra. Ciò può realizzarsi con dischi a singolo strato, come specificato in DVD-10, dando un disco composito con capacità di 9,4 Gbyte, come pure con dischi a doppio strato, come specificato in DVD-18, portando a una capacità di 17 Gbyte.

I tempi di accesso per le unità DVD sono simili a quelli delle unità CD. Tuttavia, a parità di velocità di rotazione, i tassi di trasferimento dei dati sono molto più alti con i DVD poiché la densità delle buche è più alta. Sono anche state sviluppate versioni riscrivibili di dispositivi DVD che forniscono grandi capacità di memoria di massa.

8.10.3 Sistemi a nastri magnetici

I nastri magnetici sono adatti per l'immagazzinamento off-line di grandi quantità di dati. Sono tipicamente usati per copie di riserva e per archiviazione. La registrazione su nastri magnetici usa lo stesso principio dei dischi magnetici. La principale differenza è che il film magnetico è depositato su un nastro di plastica molto sottile di larghezza 0,5 o 0,25 pollici.

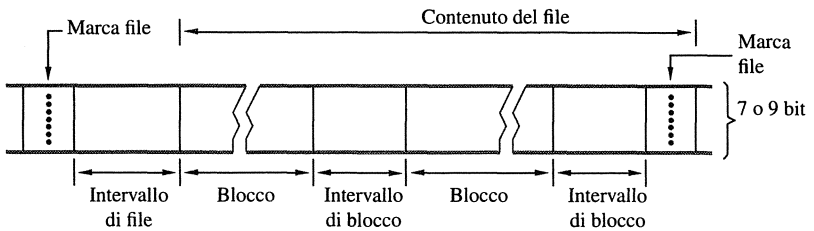


Figura 8.30 Struttura del nastro magnetico.

Sette o nove bit (che corrispondono a un carattere) sono registrati in parallelo sulla larghezza del nastro, perpendicolarmente alla direzione del movimento. Una testina di lettura e scrittura separata viene fornita per ciascuna posizione trasversale di bit sul nastro, così che tutti i bit di un carattere possono essere scritti o letti in parallelo. Uno dei bit del carattere è usato come bit di parità.

I dati sul nastro sono organizzati a formare blocchi (*record*) separati da intervalli (*gap*) non registrati, come mostrato in Figura 8.30. Il movimento del nastro viene fermato solo quando vi è un intervallo sotto le testine di lettura e scrittura. Gli intervalli fra blocchi sono sufficientemente lunghi da permettere al nastro di acquistare la sua velocità normale prima di raggiungere l'inizio del nuovo blocco. Se si usa uno schema di codifica come quello in Figura 8.27c per registrare dati sul nastro, gli intervalli fra blocchi sono identificati come aree in cui non c'è alcun cambiamento nella magnetizzazione. Questo permette di rilevare gli intervalli indipendentemente dai dati registrati. Per aiutare gli utenti a organizzare grandi quantità di dati, un gruppo di blocchi viene detto *file*. L'inizio di un file è identificato da una *marca di file*, come mostrato in Figura 8.30. La marca di file è un blocco speciale costituito da uno o più caratteri, di solito preceduto da un intervallo più lungo di quello fra blocchi. Il primo blocco dopo una marca di file può essere usato come *intestazione* o *identificatore* per il file. Questo permette all'utente di cercare un certo file in un nastro che contiene un gran numero di file.

Sistema di cartucce di nastri

I sistemi a nastri sono stati sviluppati per copie di riserva di dischi on-line. Uno di questi sistemi usa un nastro da 8 mm di formato video alloggiato in una cassetta. Queste unità sono dette cartucce di nastri (*tape cartridge*): hanno capacità che vanno da 2 a 5 Gbyte e gestiscono trasferimenti di dati al tasso di alcune centinaia di Kbyte al secondo. Lettura e scrittura sono effettuate da un sistema di scansione elicoidale che opera trasversalmente rispetto al nastro, analogamente a quello usato nelle unità a nastro per videocassette. Si possono ottenere densità di bit dell'ordine della decina di milioni di bit per centimetro quadrato. Sistemi a più cartucce sono disponibili per automatizzare il caricamento e lo scaricamento delle cassette, così da permettere il back-up di decine di Gbyte di memoria di massa on-line senza supervisione.

Sommario

Il sistema di memoria è uno dei componenti di maggiore importanza del calcolatore. La velocità e la capacità della memoria sono parametri fondamentali nel determinare la prestazione complessiva del calcolatore. Nel capitolo sono stati illustrati e discussi gli aspetti tecnologici e strutturali più rilevanti per il progetto del sistema di memoria.

Il progresso della tecnologia microelettronica a semiconduttori ha dato luogo a un aumento spettacolare di velocità e capacità dei componenti integrati di memoria, insieme a un calo imponente del costo. La prestazione di memorie per calcolatori è ulteriormente migliorata dall'uso di una gerarchia di memoria. Oggigiorno per realizzare la memoria centrale del calcolatore, che deve essere grande e affidabile, si utilizza principalmente la tecnologia di memoria dinamica o DRAM. Uno o più livelli di memoria cache sono sempre presenti. L'introduzione della memoria cache riduce significativamente l'effettivo tempo di accesso alla memoria visto dal processore. La memoria virtuale fa apparire la memoria principale più grande della memoria fisica.

I dischi magnetici continuano a essere la tecnologia principale per la memoria secondaria. Essi forniscono capacità di immagazzinamento enormi, raggiungendo e superando i mille miliardi di byte su una singola unità, con un costo per bit molto basso. Tuttavia, la tecnologia dei semiconduttori flash sta cominciando a competere in modo efficace in alcune applicazioni.

Note bibliografiche

1. Mowry T.C., *Tolerating Latency Through Software-Controlled Data Prefetching*, Ph. D. Thesis, CSL-TR-94-628, Stanford University, March 1994.
2. Baer J.L., Chen T.F., "An effective on-chip preloading scheme to reduce data access penalty", *Proc. ACM/IEEE Conf. Supercomputing'91*, Nov. 1991, pp. 176-186.
3. Fu J.W.C., Patel J.H., Janssens B.L., "Stride directed prefetching in scalar processors", *Proc. MICRO92, 25th Int'l Symp. on Microarchitecture*, Dec. 1992, pp. 102-110.
4. Kroft D., "Lockup-free instruction fetch/prefetch cache organization", *Proc. ISCA 81, 8th Int'l Symp. on Computer Architecture*, May 1981, pp. 81-87.
5. Silberschatz A., Galvin P. B., Gagne G., *Sistemi operativi – Concetti ed esempi*, 8/E., Pearson Paravia Bruno Mondadori, Milano 2009.
6. Tanenbaum A.S., *I moderni sistemi operativi*, 3/E., Pearson Paravia Bruno Mondadori, Milano 2009.
7. Patterson D.A., Gibson G.A., Katz R.H., "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proc. ACM SIGMOD, Int'l Conf. on Management of Data*, June 1988, pp. 109-116.

9 Aritmetica

In questo capitolo

- Circuiti addizionatori e sottrattori
- Addizionatori ad alta velocità basati su circuiti logici di anticipo del riporto (*carry-lookahead*)
- L'algoritmo di Booth per la moltiplicazione di numeri con segno
- Moltiplicatori ad alta velocità basati sull'addizione con salvataggio del riporto (*carry-save*)
- Circuiti logici per la divisione
- Operazioni aritmetiche su numeri in virgola mobile in conformità allo standard IEEE

L'addizione e la sottrazione di due numeri sono operazioni di base al livello delle istruzioni macchina in tutti i calcolatori. Queste operazioni, e anche altre operazioni aritmetiche e logiche, sono realizzate nell'unità aritmetica e logica (o ALU) del processore. In questo capitolo si presentano i circuiti logici usati per realizzare operazioni aritmetiche. Il tempo necessario a effettuare addizioni o sottrazioni influisce sulla prestazione del processore. Pure le operazioni di moltiplicazione e divisione, che richiedono circuiti più complessi rispetto alle operazioni di addizione o sottrazione, influiscono sulla prestazione. Si presentano alcune delle tecniche usate nei calcolatori moderni per effettuare operazioni aritmetiche ad alta velocità. Sono inoltre descritte operazioni sui numeri in virgola mobile.

Nel Paragrafo 1.4 è stata descritta la rappresentazione dei numeri binari con segno ed è stato mostrato che il complemento a due è la migliore rappresentazione per realizzare le operazioni di addizione e sottrazione. Gli esempi in Figura 1.6 mostrano che due numeri con segno a n bit possono essere sommati usando l'addizione binaria a n bit, trattando il bit di segno allo stesso modo degli altri bit. In altre parole, un circuito logico progettato per sommare numeri binari senza segno può anche essere usato per sommare numeri con segno in complemento a due. I primi due paragrafi di questo capitolo presentano i circuiti logici per l'addizione e la sottrazione.

9.1 Addizionatore e sottrattore

La Figura 9.1 mostra la tabella di verità delle funzioni combinatorie di somma e riporto per l'addizione di due numeri da un solo bit x_i e y_i, facenti parte della rappresentazione binaria di due numeri naturali X e Y da $n \geq 1$ bit ciascuno ($0 \leq i \leq n - 1$). Sono mostrate anche le espressioni logiche delle due funzioni e un esempio di calcolo di addizione ponendo $n = 4$ bit e $X = 7$, $Y = 6$. Si noti come occorra propagare il riporto lungo la catena di bit. I simboli

x_i	y_i	Riporto in ingresso c_i	Somma s_i	Riporto in uscita c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \bar{x}_i\bar{y}_i r_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = x_i c_i + y_i c_i + x_i y_i$$

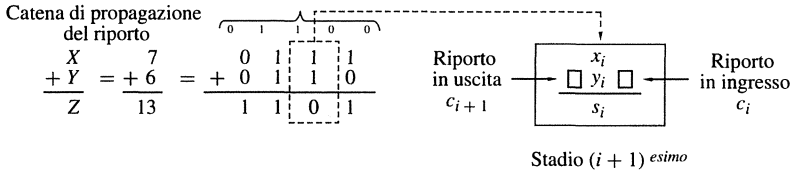


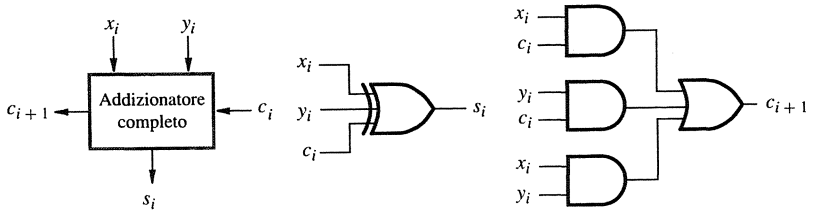
Figura 9.1 Specifica logica di uno stadio di addizionatore binario.

c_i e c_{i+1} indicano il riporto (*carry*) in ingresso e in uscita dello stadio $(i+1)^{esimo}$ di addizione; per $i > 0$, il primo (cioè c_i) coincide con il riporto c_i in uscita dallo stadio i^{esimo}.

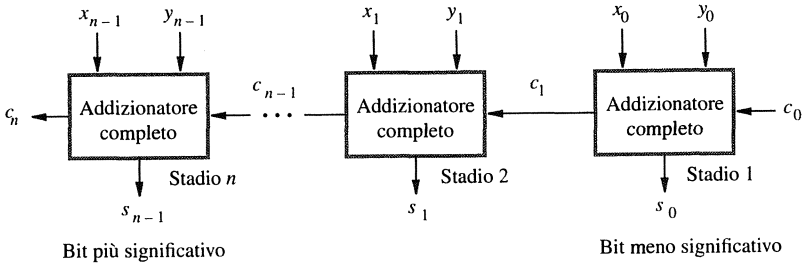
L'espressione logica che dà il bit di somma s_i ammette numerose forme differenti, per esempio si potrebbe esprimere sia come somma di prodotti sia come differenza simmetrica, cioè come XOR (si veda la Figura 9.1); in Figura 9.2a, dove si mostra la rete combinatoria di somma, si preferisce usare quest'ultima forma, più compatta. L'espressione logica del riporto in uscita c_{i+1} è data come somma di prodotti (ammette anch'essa numerose varianti, meno importanti) e la rete combinatoria corrispondente è pure mostrata in Figura 9.2a. L'insieme delle due funzioni di somma e riporto costituisce il circuito *addizionatore completo* (*full adder*), che è il blocco funzionale di base di tutti i circuiti aritmetici del calcolatore.

Componendo la catena di n stadi di tipo addizionatore completo, collegati in modo da propagare il riporto, si ottiene il circuito addizionatore per numeri binari naturali da n bit, come mostra la Figura 9.2b. Lo schema di collegamento è molto noto e di fatto riproduce l'algoritmo di addizione binaria manuale; esso va sotto il nome di *addizionatore a propagazione di riporto* (*ripple carry adder*).

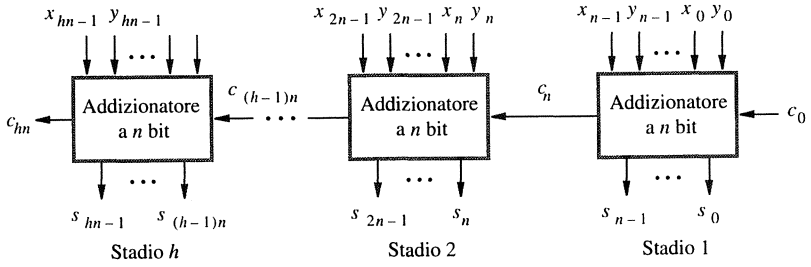
Il riporto c_0 in ingresso allo stadio addizionatore completo meno significativo fornisce un modo semplice e a costo nullo per addizionare 1 alla somma, ovvero per effettuare l'incremento. Ciò torna utile in svariate circostanze, per esempio quando si deve complementare a due un numero (giacché l'algoritmo di complementazione a due prevede di aggiungere 1 in posizione meno significativa). Il circuito addizionatore a n bit è a sua volta componibile in catena. Per esempio, prendendo $h > 1$ stadi da n bit ciascuno e componendoli si ottiene il circuito addizionatore per numeri da hn bit, come mostra la Figura 9.2c.



(a) Addizione di numeri da un bit e circuiti di somma e riporto



(b) Addizionatore da n bit a propagazione di riporto



(c) Catena di h addizionatori da n bit ciascuno

Figura 9.2 Circuiti digitali per addizione di numeri binari naturali.

9.1.1 Unità logica per addizione e sottrazione

L'addizionatore a n bit di Figura 9.2b è utilizzabile anche per calcolare la somma (algebrica) di numeri interi relativi rappresentati in complemento a due. Siano X e Y i due numeri, a $n > 1$ bit, dove x_{n-1}, y_{n-1} sono i rispettivi bit di segno. Se si vuole considerare la somma come numero a n bit, il riporto in uscita c_n va trascurato. Come determinare l'eventuale tra-

bocco è stato spiegato nel Paragrafo 1.4. L'espressione logica seguente esprime il rilevamento di trabocco:

$$\text{trabocco} = x_{n-1}y_{n-1}\bar{s}_{n-1} + \bar{x}_{n-1}\bar{y}_{n-1}s_{n-1} \tag{9.1}$$

Essa si giustifica facilmente osservando che assume valore logico 1 se e solo se i bit di segno concordano tra loro e discordano con il bit più significativo di somma s_{n-1}. Esiste un'espressione logica alternativa per il trabocco (si veda il Problema proposto 9.5 nell'Eserciziario): *si ha trabocco se e solo se i riporti* c_n e c_{n-1} differiscono; pertanto si può anche scrivere:

$$\text{trabocco} = c_n \oplus c_{n-1} \tag{9.2}$$

Il circuito di rilevamento di trabocco si riduce a una sola porta XOR; di solito nella pratica si usa la formula (9.2), meno costosa.

Per calcolare la sottrazione $X - Y$ dei due numeri X e Y in complemento a due a n bit, basta complementare a due il sottraendo Y e poi addizionarlo al minuendo X. La Figura 9.3 mostra il circuito digitale addizionatore e sottrattore corrispondente: esso calcola somma o differenza a seconda che l'ingresso "comando di addizione e sottrazione" assuma valore 0 o 1, rispettivamente. Infatti, se il comando vale 0, l'addendo Y arriva inalterato al circuito addizionatore e anche il riporto in ingresso c_0 è nullo; dunque si calcola la somma $X + Y$. Invece, se il comando vale 1, i bit dell'addendo Y vengono negati (logicamente) e il riporto in ingresso assume valore $c_0 = 1$, pertanto è come se Y fosse stato complementato a due, cioè in ingresso all'addizionatore si ha $Y' = -Y$, dunque si calcola la somma $X + Y' = X - Y$, vale a dire in realtà la differenza. Se si preferisce si può aggiungere al circuito di Figura 9.3 una porta

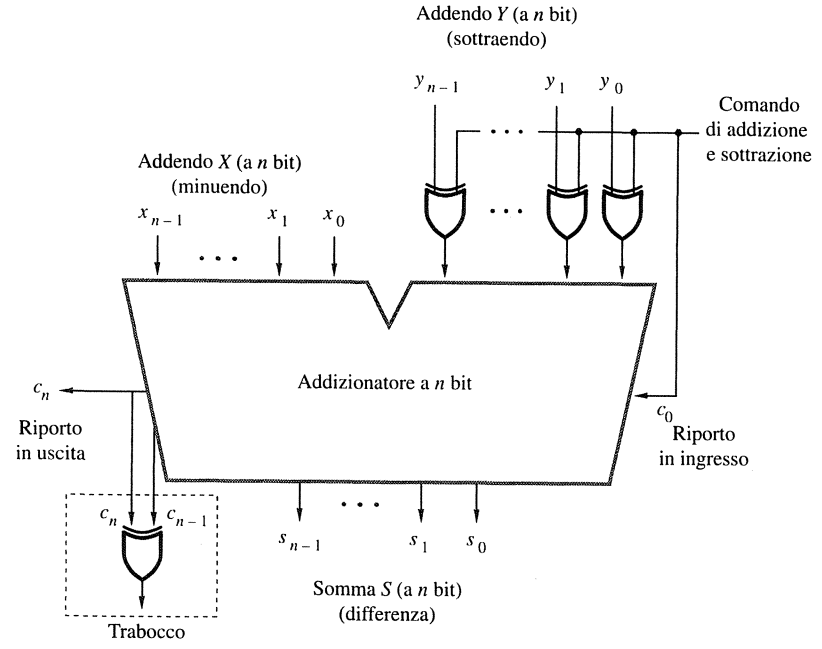


Figura 9.3 Addizionatore algebrico da n bit.

XOR per rilevare l'eventuale trabocco, usando la relazione $c_n \oplus c_{n-1}$ data dalla formula (9.2) ed estraendo il riporto c_{n-1} dall'interno dell'addizionatore (che certamente lo produce).

9.2 Progettazione di addizionatori veloci

Se un addizionatore a n bit a propagazione del riporto (*ripple-carry adder*) venisse usato nel circuito di addizione e sottrazione di Figura 9.3, potrebbe esserci troppo ritardo nella produzione delle sue uscite, da s_0 a s_{n-1} e c_n. Se il ritardo possa essere accettabile o meno può essere deciso solo nel contesto della velocità di altri componenti del processore e dei tempi di trasferimento dei dati di registri e memorie cache. Il ritardo attraverso una rete di porte logiche dipende dalla tecnologia elettronica dei circuiti integrati usati per costruire la rete e dal numero di porte nei percorsi da ingressi a uscite. Il ritardo di attraversamento di qualsiasi circuito combinatorio costruito da porte in una certa tecnologia è determinato sommando il numero di ritardi di propagazione di porta logica nel più lungo percorso di propagazione del segnale attraverso il circuito. Nell'addizionatore da n bit a propagazione di riporto, il cammino più lungo va dagli ingressi x_0, y_0 e c_0 nella posizione meno significativa, o LSB (*Least Significant Bit*), alle uscite c_n e s_{n-1} nella posizione più significativa, o MSB (*Most Significant Bit*).

Usando la realizzazione indicata in Figura 9.2a, c_{n-1} è disponibile dopo $2(n - 1)$ ritardi di porta, e s_{n-1} è corretto dopo un ulteriore ritardo di porta XOR. Il riporto in uscita finale, c_n, è disponibile dopo $2n$ ritardi di porta. Quindi, se si usa un addizionatore a propagazione di riporto per realizzare l'unità di addizione e sottrazione mostrata in Figura 9.3, tutti i bit di somma sono disponibili dopo $2n$ ritardi di porta, includendo il ritardo attraverso le porte XOR sull'ingresso Y. Usando la realizzazione $c_n \oplus c_{n-1}$ per il trabocco, questo indicatore è disponibile dopo $2n + 2$ ritardi di porta.

Si possono seguire due approcci per ridurre i ritardi negli addizionatori. Il primo approccio consiste nell'usare la tecnologia elettronica più veloce possibile. Il secondo approccio consiste nell'usare una rete di porte logiche chiamata rete di anticipo del riporto (*carry-lookahead*), che è descritta nel prossimo paragrafo.

9.2.1 Addizione con anticipo del riporto

Un circuito di addizione veloce deve accelerare la generazione dei segnali di riporto. Le espressioni logiche per s_i (somma) e c_{i+1} (riporto in uscita) dello stadio i (si veda la Figura 9.1) sono:

$$s_i = x_i \oplus y_i \oplus c_i$$

e:

$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

Fattorizzando la seconda equazione in:

$$c_{i+1} = x_iy_i + (x_i + y_i)c_i$$

si può scrivere:

$$c_{i+1} = G_i + P_ic_i$$

dove:

$$G_i = x_iy_i \qquad \text{e} \qquad P_i = x_i + y_i$$

Le espressioni G_i e P_i sono dette funzioni di *generazione* e *propagazione* per lo stadio i. Se la funzione di generazione per lo stadio i è uguale a 1, allora $c_{i+1} = 1$, indipendentemente

dal riporto in ingresso, c_i. Questo avviene quando entrambi x_i e y_i sono 1. La funzione di propagazione fa sì che un riporto in ingresso produrrà un riporto in uscita quando x_i è 1 o y_i è 1. Tutte le funzioni G_i e P_i possono essere formate indipendentemente e in parallelo dopo un ritardo di porta logica dopo che gli operandi X e Y siano stati applicati agli ingressi di un addizionatore a n bit. Ciascuno stadio da un bit contiene una porta AND per formare G_i, una porta OR per formare P_i, e una porta XOR a tre ingressi per formare s_i. Si può derivare un circuito più semplice osservando che una funzione di propagazione adeguata può essere realizzata da $P_i = x_i \oplus y_i$, che differisce da $P_i = x_i + y_i$ solo quando $x_i = y_i = 1$. Tuttavia in questo caso $G_i = 1$, così non importa se P_i è 0 o 1. Quindi, usando una cascata di due porte XOR a due ingressi per realizzare la funzione XOR a 3 ingressi per s_i, si può usare la cella di base B in Figura 9.4a in ciascuno stadio da un bit.

Espandendo c_i in termini di variabili con pedice $i - 1$ e sostituendo nell'espressione di c_{i+1}, si ottiene:

$$c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}c_{i-1}$$

Continuando questo tipo di espansione, l'espressione finale per qualsiasi variabile di riporto è:

$$c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots + P_iP_{i-1}\ldots P_1G_0 + P_iP_{i-1}\ldots P_0c_0 \qquad (9.3)$$

Quindi si possono ottenere tutti i riporti tre ritardi di porta dopo che gli operandi in ingresso X, Y e c_0 siano stati applicati, poiché solo un ritardo di porta è necessario per generare tutti i segnali P_i e G_i, seguito da due ritardi di porta nel circuito AND-OR per c_{i+1}. Dopo un ulteriore ritardo di porta XOR, tutti i bit di somma saranno disponibili. In totale il processo di addizione da n bit richiede solo quattro ritardi di porta, indipendentemente da n.

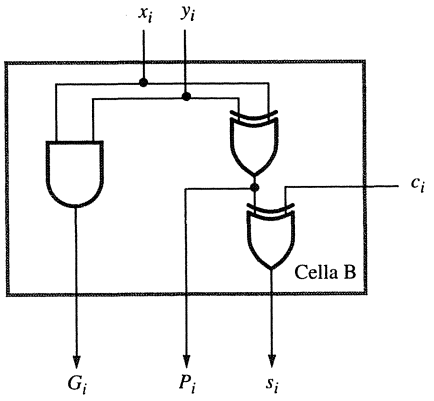
Si consideri la progettazione di un addizionatore a 4 bit. I riporti possono essere realizzati come segue.

$$c_1 = G_0 + P_0c_0$$
$$c_2 = G_1 + P_1G_0 + P_1P_0c_0$$
$$c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$
$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

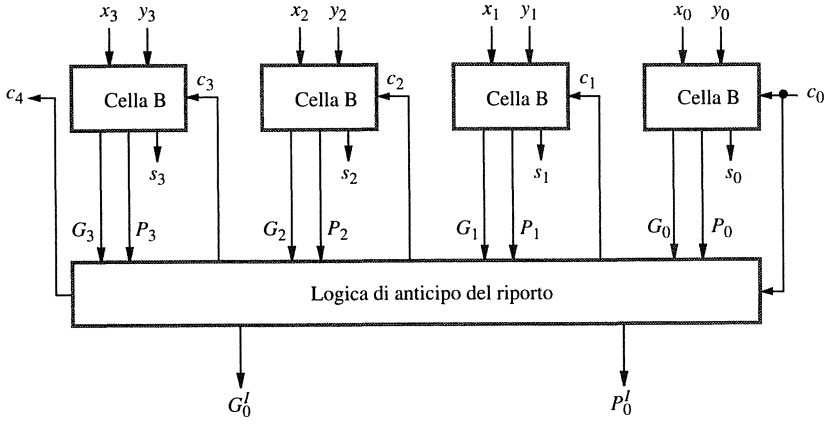
L'addizionatore a 4 bit completo è mostrato in Figura 9.4b. I riporti sono determinati nel blocco denominato logica di anticipo del riporto. Un addizionatore realizzato in tal forma è detto *addizionatore con anticipo del riporto* (*carry-lookahead adder*). Il ritardo di attraversamento dell'addizionatore è pari a 3 ritardi di porta per tutti i bit del riporto e a 4 ritardi di porta per tutti i bit di somma. Per confronto, un addizionatore a propagazione del riporto da 4 bit richiede 7 ritardi di porta per s_3 e 8 ritardi di porta per c_4.

Se si prova a estendere il progetto dell'addizionatore con anticipo del riporto di Figura 9.4b per operandi più lunghi, si incontra il problema dei vincoli di fan-in di porta. Dall'espressione (9.3) si vede che l'ultima porta AND e la porta OR richiedono un fan-in pari a $i + 2$ per la generazione di c_{i+1}. Si richiede un fan-in pari a 5 per c_4 nell'addizionatore a 4 bit. Questo è all'incirca il limite pratico per le porte. Quindi il progetto dell'addizionatore mostrato in Figura 9.4b non può essere facilmente esteso per operandi più lunghi. Tuttavia è possibile costruire addizionatori più lunghi per mezzo di addizionatori a 4 bit in cascata, come mostrato in Figura 9.2c.

Si possono connettere otto addizionatori a 4 bit con anticipo del riporto come in Figura 9.2c per formare un addizionatore a 32 bit. I ritardi di generazione dei bit di somma s_{31}, s_{30}, s_{29}, s_{28}, e del bit di riporto c_{32} nell'addizionatore a 4 bit di ordine alto in questa cascata sono calcolati come segue. Il riporto in uscita c_4 dall'addizionatore di ordine basso è disponibile dopo 3 ritardi di porta dall'applicazione degli operandi in ingresso X, Y e c_0 all'addizionatore a 32 bit. Quindi, c_8 è disponibile all'uscita del secondo addizionatore dopo 2 ulteriori



(a) Cella di stadio da un bit



(b) Addizionatore da 4 bit

Figura 9.4 Un addizionatore con anticipo del riporto da 4 bit.

ritardi di porta, c_{12} è disponibile dopo altri 2 ritardi di porta, e così via. Infine, c_{28}, il riporto in ingresso all'addizionatore a 4 bit di ordine alto è disponibile dopo un totale di $(6 \times 2) + 3 = 15$ ritardi di porta. Quindi c_{32} e tutti i riporti interni all'addizionatore di ordine alto sono disponibili dopo altri 2 ritardi di porta, e tutti i 4 bit di somma sono disponibili dopo ancora 1 ritardo di porta, per un totale di 18 ritardi di porta. Questo è da confrontare con ritardi totali di 63 e 64 per s_{31} e c_{32} se si usa un addizionatore a propagazione del riporto.

Nel prossimo paragrafo mostreremo come si può migliorare la struttura a cascata appena discussa, portando a un'ulteriore riduzione nel ritardo dell'addizionatore. L'idea è di generare i riporti c_4, c_8, ... in parallelo, in modo simile alla generazione in parallelo di c_1, c_2, c_3 e c_4 nell'addizionatore a 4 bit con anticipo del riporto.

Funzioni di generazione e propagazione di più alto livello

Nell'addizionatore a 32 bit appena discusso, i riporti c_4, c_8, c_{12}, ... si propagano attraverso i blocchi addizionatori a 4 bit con due ritardi di porta per blocco, in modo analogo a come i riporti individuali si propagano attraverso ciascuno stadio da un bit nell'addizionatore a propagazione del riporto. Si può usare l'approccio di anticipo del riporto per sviluppare c_4, c_8, c_{12}, ... in parallelo usando funzioni di generazione e propagazione per i blocchi di più alto livello.

La Figura 9.5 mostra un addizionatore a 16 bit costruito a partire da blocchi addizionatori a 4 bit. Questi blocchi forniscono nuove funzioni di uscita definite da G_k^I e P_k^I, dove $k = 0$ per il primo blocco da 4 bit, $k = 1$ per il secondo blocco da 4 bit e così via, come mostrato in Figura 9.4b e 9.5. Nel primo blocco:

$$P_k^I = P_3P_2P_1P_0$$

e:

$$G_k^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

Le funzioni di primo livello G_i e P_i determinano se lo stadio del bit i genera o propaga un riporto. Le funzioni di secondo livello G_k^I e P_k^I determinano se il blocco k genera o propaga un riporto. Con queste nuove funzioni disponibili, non occorre aspettare che i riporti si propaghino attraverso i blocchi da 4 bit. Il riporto c_{16} è formato da uno dei circuiti di anticipo del riporto in Figura 9.5 come:

$$c_{16} = G_3^I + P_3^IG_2^I + P_3^IP_2^IG_1^I + P_3^IP_2^IP_1^IG_0^I + P_3^IP_2^IP_1^IP_0^Ic_0$$

I riporti in ingresso ai blocchi da 4 bit sono formati in parallelo da espressioni simili. Le espressioni per c_{16}, c_{12}, c_8 e c_4 sono identiche nella forma rispettivamente alle espressioni per

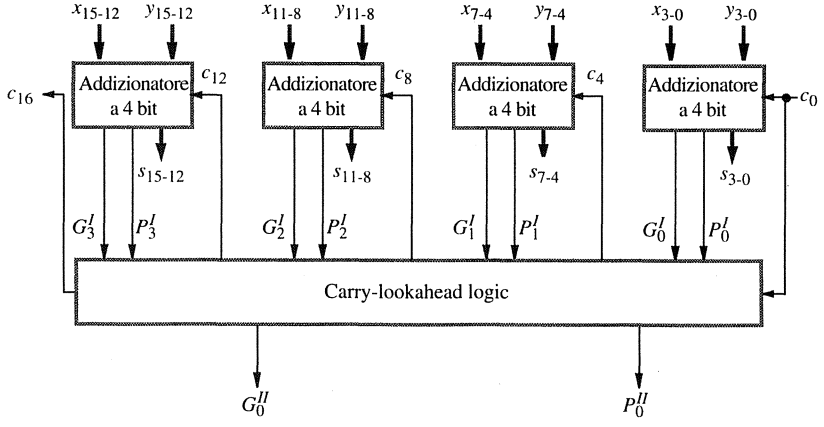


Figura 9.5 Un addizionatore con anticipo del riporto da 16 bit costruito con addizionatori a 4 bit (si veda la Figura 4b).

c_4, c_3, c_2 e c_1, realizzate nei circuiti di anticipo del riporto in Figura 9.4b. Solo i nomi delle variabili sono differenti. Quindi, la struttura dei circuiti di anticipo del riporto in Figura 9.5 è identica a quella dei circuiti di anticipo del riporto in Figura 9.4b. Tuttavia i riporti c_4, c_8, c_{12} e c_{16} generati internamente dai blocchi addizionatori a 4 bit non sono necessari in Figura 9.5 poiché essi sono generati dai circuiti di anticipo del riporto di livello più alto.

Si consideri adesso il ritardo nel produrre le uscite dell'addizionatore a 16 bit con anticipo del riporto. Il ritardo per ottenere i riporti prodotti dai circuiti di anticipo del riporto è di due porte in più rispetto al ritardo necessario per ottenere le funzioni G_k^I e P_k^I. Queste ultime richiedono rispettivamente due ritardi di porta e un ritardo di porta, dopo la generazione di G_i e P_i. Quindi, tutti i riporti prodotti dai circuiti di anticipo del riporto sono disponibili 5 ritardi di porta dopo che X, Y e c_0 siano stati applicati in ingresso. Il riporto c_{15} è generato all'interno del blocco da 4 bit di ordine alto in Figura 9.5 con due ritardi di porta dopo c_{12}, seguito da s_{15} con un ulteriore ritardo di porta. Quindi s_{15} è disponibile dopo 8 ritardi di porta. Se un addizionatore a 16 bit è costruito tramite una cascata di blocchi addizionatori da 4 bit ad anticipo del riporto, i ritardi per produrre c_{16} e s_{15} sono rispettivamente 9 e 10 ritardi di porta, in confronto ai 5 e 8 ritardi di porta per la configurazione in Figura 9.5.

Due blocchi addizionatori a 16 bit possono essere usati in cascata per realizzare un addizionatore a 32 bit. In questa configurazione, l'uscita c_{16} dal blocco di ordine basso è il riporto in ingresso al blocco di ordine alto. Il ritardo è molto più basso rispetto al ritardo dell'addizionatore a 32 bit discusso precedentemente, che era costruito mettendo in cascata otto addizionatori a 4 bit. In quella configurazione, si ricordi che s_{31} è disponibile dopo 18 ritardi di porta e c_{32} è disponibile dopo 17 ritardi di porta. L'analisi dei ritardi per la cascata di due addizionatori a 16 bit è la seguente. Il riporto c_{16} del blocco di ordine basso è disponibile dopo 5 ritardi di porta, come calcolato sopra. Quindi, c_{28} e c_{32} sono entrambi disponibili nel blocco di ordine alto dopo altri 2 ritardi di porta, e c_{31} è disponibile 2 ritardi di porta dopo c_{28}. Quindi, c_{31} è disponibile dopo un totale di 9 ritardi di porta e s_{31} è disponibile dopo 10 ritardi di porta. Ricapitolando, s_{31} e c_{32} sono disponibili rispettivamente dopo 10 e 7 ritardi di porta, in confronto ai 18 e 17 ritardi di porta per le stesse uscite se l'addizionatore a 32 bit fosse costruito mettendo in cascata otto addizionatori da 4 bit.

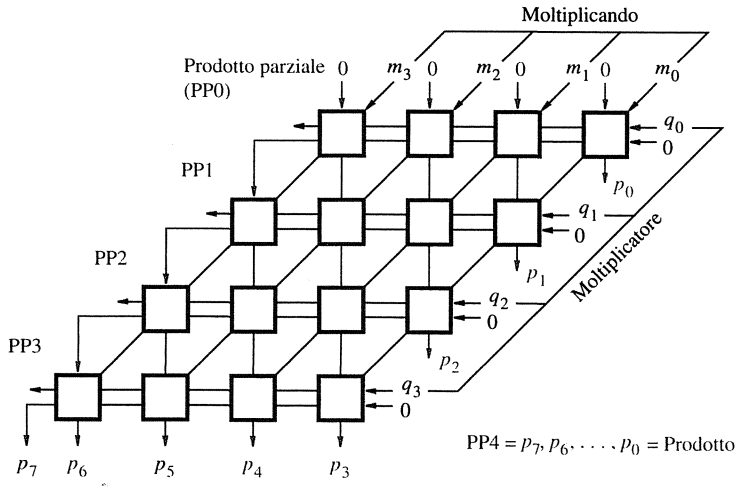
Lo stesso ragionamento usato per sviluppare le funzioni di secondo livello G_k^I e P_k^I a partire dalle funzioni di primo livello G_i e P_i può essere seguito per sviluppare funzioni di terzo livello G_k^{II} e P_k^{II} dalle funzioni G_k^I e P_k^I. Due tali funzioni di terzo livello sono mostrate come uscite dalla logica di anticipo del riporto in Figura 9.5. Un addizionatore a 64 bit può essere costruito usando quattro addizionatori a 16 bit del tipo mostrato in Figura 9.5 e circuiti aggiuntivi di logica di anticipo del riporto che producono i riporti c_{16}, c_{32}, c_{48} e c_{64}. Si può dimostrare che i ritardi attraverso questo addizionatore sono 12 ritardi di porta per s_{63} e 7 ritardi di porta per c_{64}, estendendo il ragionamento usato sopra per l'addizionatore a 16 bit (si veda il Problema proposto 9.7 nell'Eserciziario).

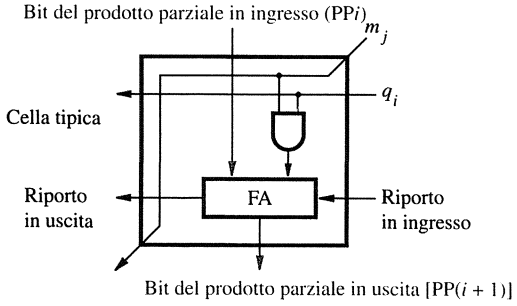
9.3 Moltiplicazione di numeri senza segno

In Figura 9.6a è illustrato l'algoritmo usuale di moltiplicazione manuale degli interi, per il sistema binario. Il prodotto di due numeri senza segno di n cifre può essere rappresentato con $2n$ cifre, quindi il prodotto dei due numeri da 4 bit in questo esempio è rappresentato con 8 bit, come mostrato. Nel sistema binario, la moltiplicazione del moltiplicando per un bit del moltiplicatore è facile. Se il bit del moltiplicatore è 1, si copia il moltiplicando facendolo scorrere nella posizione appropriata. Se il bit del moltiplicatore è 0, si inseriscono zeri, come nella terza riga dell'esempio. Il prodotto è calcolato un bit alla volta sommando le colonne di bit da destra a sinistra e propagando i valori del riporto fra le colonne.

```
                1 1 0 1      (13) Moltiplicando M
              × 1 0 1 1      (11) Moltiplicatore Q
              -----------
                1 1 0 1
              1 1 0 1
            0 0 0 0
          1 1 0 1
          -----------------
        1 0 0 0 1 1 1 1      (143) Prodotto P
```

(a) Algoritmo di moltiplicazione manuale





(b) Realizzazione a matrice

Figura 9.6 Moltiplicazione a matrice di operandi binari senza segno.

9.3.1 Moltiplicatore a matrice

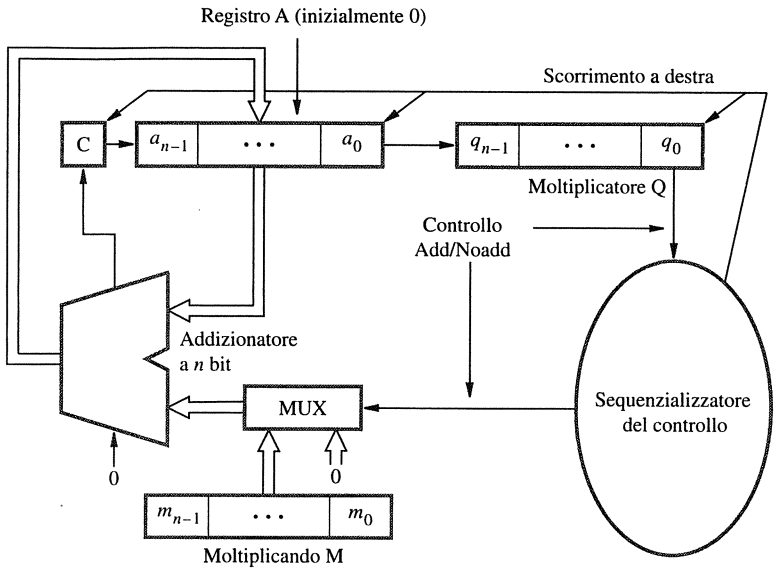
Si può realizzare una moltiplicazione binaria di operandi senza segno per mezzo di una matrice logica combinatoria a due dimensioni, come mostrato in Figura 9.6b per il caso di operandi a 4 bit. Il componente principale in ciascuna cella è un addizionatore completo (*Full Adder*), FA. La porta AND in ciascuna cella determina se un bit del moltiplicando, m_j, è sommato al bit del prodotto parziale in ingresso, in base al valore del bit del moltiplicatore, q_i. Ciascuna riga i, dove $0 \leq i \leq 3$, somma il moltiplicando (dopo opportuno scorrimento) al prodotto parziale in ingresso, PPi, per generare il prodotto parziale uscente, PP(i+1), se $q_i = 1$. Se $q_i = 0$, PPi è inoltrato verticalmente verso il basso senza essere modificato. PP0 è tutti zeri, e PP4 è il prodotto desiderato. Il moltiplicando scorre a sinistra di una posizione per riga grazie al percorso diagonale dei segnali. Si noti che l'addizione riga per riga fatta nel circuito della matrice differisce dalla solita addizione manuale descritta precedentemente, che è fatta colonna per colonna.

Il caso peggiore di ritardo di propagazione del segnale si ha nel percorso dall'angolo in alto a destra della matrice fino al bit di uscita di ordine più alto del prodotto nell'angolo in basso a sinistra della matrice. Questo percorso critico consiste della sequenza a forma di scala che include le due celle alla destra di ciascuna riga, seguita da tutte le celle della riga in basso. Assumendo che ci siano due ritardi di porta dagli ingressi alle uscite per un blocco addizionatore completo, FA, il percorso critico avrebbe un ritardo totale di $6(n - 1) - 1$ ritardi di porta, includendo il ritardo iniziale della porta AND in tutte le celle, per una matrice $n \times n$ (si veda il Problema proposto 9.8 nell'Eserciziario). Nella prima riga della matrice non sono necessari addizionatori completi, poiché il prodotto parziale in ingresso PP0 è zero. Di ciò si è tenuto conto nello sviluppo dell'espressione del ritardo.

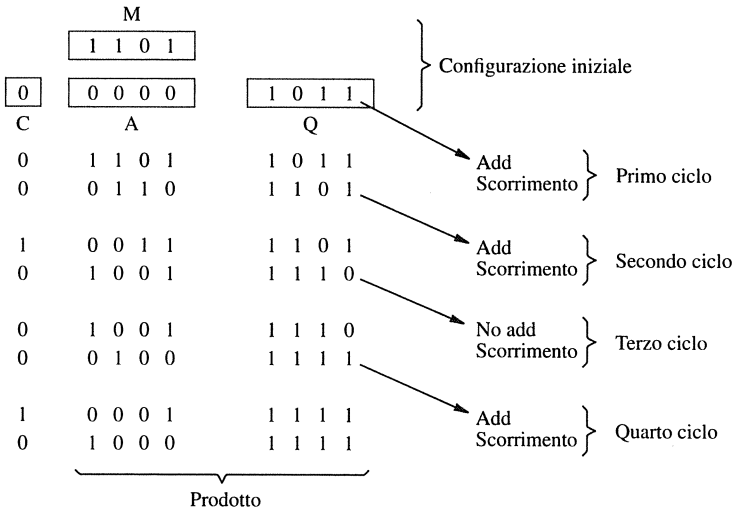
9.3.2 Circuito moltiplicatore sequenziale

Il moltiplicatore combinatorio a matrice appena descritto usa un gran numero di porte logiche per moltiplicare numeri di dimensioni pratiche, ovvero numeri a 32 o 64 bit. La moltiplicazione di due numeri a n bit può anche essere fatta in un circuito sequenziale che usa un solo addizionatore a n bit.

Il diagramma a blocchi in Figura 9.7a mostra la sistemazione dell'hardware per la moltiplicazione sequenziale. Questo circuito effettua la moltiplicazione usando n volte un singolo addizionatore a n bit per realizzare l'addizione effettuata nello spazio dalle n righe di addizionatori a propagazione del riporto in Figura 9.6b. I registri A e Q sono registri a scorrimento, concatenati come mostrato. Insieme, essi contengono il prodotto parziale PPi mentre il bit q_i del moltiplicatore genera il segnale Add/Noadd. Questo segnale fa sì che il multiplatore MUX selezioni 0 quando $q_i = 0$, o selezioni il moltiplicando M quando $q_i = 1$, da sommare a PPi per generare PP(i + 1). Il prodotto è calcolato in n cicli. Il prodotto parziale cresce in lunghezza di un bit per ciclo a partire dal vettore iniziale, PP0, di n zeri nel registro A. Il riporto in uscita dall'addizionatore è memorizzato nel flip-flop C, mostrato all'estrema sinistra del registro A. All'inizio il moltiplicatore è caricato nel registro Q, il moltiplicando nel registro M, e C e A sono azzerati. Alla fine di ciascun ciclo, C, A e Q scorrono a destra di una posizione per permettere al prodotto parziale di crescere man mano che il moltiplicatore scorre fuori dal registro Q. A causa di questo scorrimento, il bit q_i del moltiplicatore appare nella posizione LSB di Q per generare il segnale Add/Noadd al momento giusto, iniziando con q_0 durante il primo ciclo, q_1 durante il secondo ciclo e così via. Dopo che sono stati usati, i bit del moltiplicatore vengono scartati dall'operazione di scorrimento a destra. Si noti che il riporto in uscita dall'addizionatore è il bit più a sinistra di PP(i + 1), e che deve essere tenuto nel flip-flop C per scorrere a destra assieme ai contenuti di A e Q. Dopo n cicli, la metà di ordine alto del prodotto si trova nel registro A e la metà di ordine basso è nel registro Q. La Figura 9.7b mostra come questa sistemazione dell'hardware realizzerebbe l'esempio di moltiplicazione di Figura 9.6a.



(a) Configurazione dei registri



(b) Esempio di moltiplicazione

Figura 9.7 Moltiplicatore binario a circuito sequenziale.

9.4 Moltiplicazione di numeri con segno

Si discute adesso la moltiplicazione di operandi in complemento a due, che genera un prodotto di lunghezza doppia. La strategia generale è ancora di accumulare i prodotti parziali sommando versioni del moltiplicando selezionate dai bit del moltiplicatore.

Si consideri inizialmente il caso di un moltiplicatore positivo e un moltiplicando negativo. Quando si somma un moltiplicando negativo a un prodotto parziale, bisogna estendere il valore del bit di segno del moltiplicando a sinistra per tutta l'estensione del prodotto. La Figura 9.8 mostra un esempio in cui un operando con segno a 5 bit, −13, è il moltiplicando. Esso è moltiplicato per +11, ottenendo il prodotto a 10 bit, −143.

È evidenziata l'estensione del segno del moltiplicando. L'hardware discusso precedentemente può essere usato per numeri negativi se vi si aggiunge il supporto dell'estensione del segno dei prodotti parziali.

Per un moltiplicatore negativo, la soluzione immediata è formare il complemento a due di entrambi moltiplicatore e moltiplicando e procedere come nel caso di un moltiplicatore positivo. Questo è possibile poiché complementando entrambi gli operandi non cambiano né valore né segno del prodotto. Una tecnica che funziona ugualmente bene per moltiplicatori sia positivi sia negativi, chiamata algoritmo di Booth, è descritta nel seguito.

9.4.1 Algoritmo di Booth

L'algoritmo di Booth [1] genera un prodotto di $2n$ bit e tratta uniformemente gli operandi a n bit sia positivi sia negativi in complemento a due. Per comprendere il fondamento di questo algoritmo, si consideri un'operazione di moltiplicazione in cui il moltiplicatore sia positivo e abbia un singolo blocco di 1, per esempio 0011110. Per derivare il prodotto si potrebbero sommare quattro versioni del moltiplicando con scorrimento appropriato, come nella procedura standard. Tuttavia, è possibile ridurre il numero di operazioni richieste trattando questo moltiplicatore come la differenza fra due numeri:

$$\begin{array}{rll} & 0100000 & (32) \\ - & 0000010 & (2) \\ \hline & 0011110 & (30) \end{array}$$

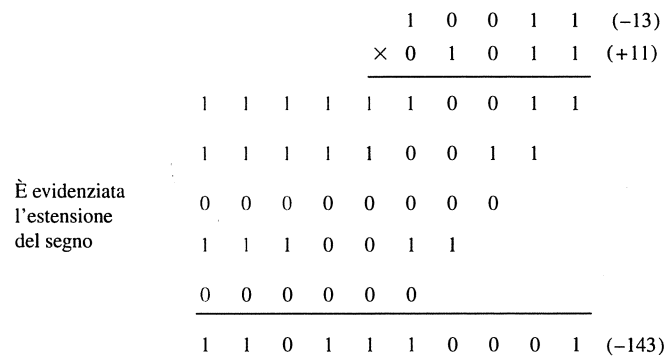


Figura 9.8 Estensione del segno di moltiplicando negativo.

```
                                0  1  0  1  1  0  1
                                0  0 +1 +1 +1 +1  0
                                ------------------
                                0  0  0  0  0  0  0
                             0  1  0  1  1  0  1
                          0  1  0  1  1  0  1
                       0  1  0  1  1  0  1
                    0  1  0  1  1  0  1
                 0  0  0  0  0  0  0
              0  0  0  0  0  0  0
              ------------------------------------
              0  0  0  1  0  1  0  1  0  0  0  1  1  0

                                0  1  0  1  1  0  1
                                0 +1  0  0  0 -1  0
                                ------------------
0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  1  1  1  1  1  1  0  1  0  0  1  1     <---- Complemento a due
0  0  0  0  0  0  0  0  0  0  0  0        del moltiplicando
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  0  1
0  0  0  0  0  0  0  0
------------------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

Figura 9.9 Schemi di moltiplicazione normale e di Booth.

Questo suggerisce che il prodotto può essere generato sommando il moltiplicando per 2^5 al complemento a due del moltiplicando per 2^1. Per convenienza si può descrivere la sequenza delle operazioni richieste ricodificando il suddetto moltiplicatore come 0 +1 0 0 0 −1 0.

In generale, nell'algoritmo di Booth si seleziona −1 per il moltiplicando con scorrimento quando si passa da 0 a 1, mentre si seleziona +1 per il moltiplicando con scorrimento quando si passa da 1 a 0, man mano che si scandisce il moltiplicatore da destra a sinistra. La Figura 9.9 illustra l'algoritmo normale e quello di Booth per l'esempio appena discusso. L'algoritmo di Booth si estende chiaramente a qualsiasi numero di blocchi di 1 in un moltiplicatore, inclusi i blocchi costituiti da un singolo 1. La Figura 9.10 mostra un altro esempio di ricodifica di un moltiplicatore. Il caso in cui il bit meno significativo del moltiplicatore è 1 è trattato assumendo che uno zero implicito sia presente alla sua destra. L'algoritmo di Booth può anche essere usato direttamente per moltiplicatori negativi, come mostrato in Figura 9.11.

```
0  0  1  0  1  1  0  0  1  1  1  0  1  0  1  1  0  0
                          ⇓⇓
0 +1 -1 +1  0 -1  0 +1  0  0 -1 +1 -1 +1  0 -1  0  0
```

Figura 9.10 Ricodifica di Booth di un moltiplicatore.


```
  0  1  1  0  1   (+13)                    0  1  1  0  1
× 1  1  0  1  0   (-6)      ⟹             0 -1 +1 -1  0
-------------                            ------------------
                                0  0  0  0  0  0  0  0  0  0
                                1  1  1  1  1  0  0  1  1
                                0  0  0  0  1  1  0  1
                                1  1  1  0  0  1  1
                                0  0  0  0  0  0
                                ------------------------------
                                1  1  1  0  1  1  0  0  1  0   (-78)
```

Figura 9.11 Moltiplicazione di Booth con moltiplicatore negativo.

Per dimostrare la correttezza dell'algoritmo di Booth per moltiplicatori negativi, si usa la seguente proprietà della rappresentazione dei numeri negativi nel sistema complemento a due.

Si supponga che lo 0 più a sinistra di un numero negativo, X, sia alla posizione k della stringa binaria, ovvero:

$$X = 11 \ldots 10x_{k-1} \ldots x_0$$

Quindi il valore X è dato da:

$$V(X) = -2^{k+1} + x_{k-1} \times 2^{k-1} + \ldots + x_0 \times 2^0$$

La correttezza di questa espressione per $V(X)$ si mostra osservando che se X è formato dalla somma di due numeri, come:

$$\begin{array}{rr} & 11\ldots100000\ldots0 \\ + & 00\ldots00x_{k-1}\ldots x_0 \\ \hline X = & 11\ldots10x_{k-1}\ldots x_0 \end{array}$$

allora il numero in alto è la rappresentazione in complemento a due di -2^{k+1}. La ricodifica del moltiplicatore consiste adesso della parte corrispondente al numero in basso, con −1 aggiunto nella posizione $k + 1$. Per esempio, il moltiplicatore 110110 è ricodificato come 0 −1 +1 0 −1 0. Il caso in cui X non ha zeri è presto trattato: tale X vale −1 nella rappresentazione in complemento a due, e la ricodifica è appunto 0 in tutte le posizioni tranne quella più a destra, dove si ha −1 per via dello 0 implicito a destra.

La tecnica di Booth per la ricodifica dei moltiplicatori è schematizzata in Figura 9.12. La trasformazione 011 ... 110 ⇒ +1 0 0 ... 0 −1 0 è chiamata *salto degli 1* (*skipping over 1s*). Questo termine deriva dal caso in cui il moltiplicatore ha i suoi 1 raggruppati in alcuni blocchi contigui. Solo alcune versioni del moltiplicando con scorrimento (gli addendi) vanno sommate per generare il prodotto, accelerando così l'esecuzione della moltiplicazione. Tuttavia, nel caso peggiore – quello in cui 1 e 0 si alternano nel moltiplicatore – ciascun bit del moltiplicatore seleziona un addendo. In effetti, questo fa sì che ci siano più addendi che senza usare l'algoritmo di Booth. Un moltiplicatore di caso peggiore, un moltiplicatore ordinario e un buon moltiplicatore, tutti da 16 bit, sono mostrati in Figura 9.13.

L'algoritmo di Booth ha due caratteristiche interessanti. La prima è che gestisce moltiplicatori positivi e negativi uniformemente. La seconda è che ottiene una certa efficienza nel numero di addizioni richieste quando il moltiplicatore ha alcuni grandi blocchi di 1.

Moltiplicatore		Versione del moltiplicando selezionata dal bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figura 9.12 Tabella di ricodifica di Booth del moltiplicatore.

Moltiplicatore di caso peggiore

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

⇓

+1 −1 +1 −1 +1 −1 +1 −1 +1 −1 +1 −1 +1 −1 +1 −1

Moltiplicatore ordinario

1 1 0 0 0 1 0 1 1 0 1 1 1 1 0 0

⇓

0 −1 0 0 +1 −1 +1 0 −1 +1 0 0 0 −1 0 0

Moltiplicatore buono

0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1

⇓

0 0 0 +1 0 0 0 0 −1 0 0 0 +1 0 0 −1

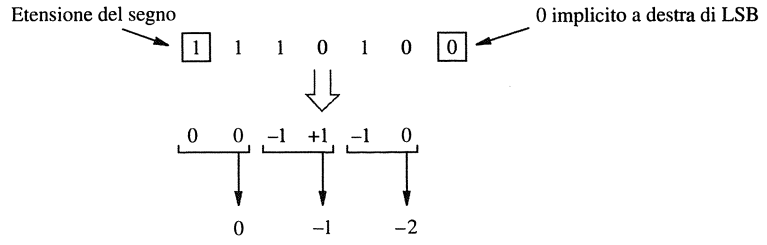
Figura 9.13 Moltiplicatori ricodificati da ricodifica di Booth.

9.5 Moltiplicazione veloce

Si descrivono adesso due tecniche per accelerare l'esecuzione della moltiplicazione. La prima tecnica garantisce che il numero massimo di addendi (versioni del moltiplicando) da sommare sia $n/2$ per operandi a n bit. La seconda tecnica porta a sommare gli addendi in parallelo.

9.5.1 Ricodifica bit-pair dei moltiplicatori

Una tecnica detta *ricodifica bit-pair*, cioè a coppie di bit, del moltiplicatore fa sì che si usi al più un addendo per ciascuna coppia di bit nel moltiplicatore. È derivata direttamente dall'algoritmo di Booth. Si raggruppino in coppie i bit del moltiplicatore ricodificati con l'algoritmo di Booth e si osservi quanto segue. La coppia $(+1\ -1)$ è equivalente alla coppia $(0\ +1)$, ovvero, invece di sommare -1 per il moltiplicando M in posizione di scor-



(a) Esempio di ricodifica bit-pair derivata da ricodifica di Booth

Coppia di bit del moltiplicatore		Bit del moltiplicatore sulla destra	Versione del moltiplicando selezionata in posizione i
$i+1$	i	$i-1$	
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

(b) Tabella delle selezioni di versione del moltiplicando

Figura 9.14 Ricodifica bit-pair del moltiplicatore.

rimento i a $+1 \times M$ in posizione $i+1$, si ottiene lo stesso risultato sommando $+1 \times M$ in posizione i. Altri esempi sono: $(+1\ 0)$ è equivalente a $(0\ +2)$, $(-1\ +1)$ è equivalente a $(0\ -1)$ e così via. Quindi, se il moltiplicatore ricodificato secondo Booth viene esaminato due bit alla volta, partendo da destra, esso può essere riscritto in una forma che richiede al più una versione del moltiplicando da sommare al prodotto parziale per ciascuna coppia di bit del moltiplicatore. La Figura 9.14a mostra un esempio di ricodifica bit-pair del moltiplicatore in Figura 9.11, e la Figura 9.14b mostra una tabella delle decisioni di selezione del moltiplicando per tutte le possibilità. L'operazione di moltiplicazione in Figura 9.11 è mostrata in Figura 9.15 per come sarebbe calcolata usando la ricodifica bit-pair del moltiplicatore.

9.5.2 Addizione di addendi con salvataggio del riporto

La moltiplicazione richiede l'addizione di vari addendi. Per accelerare il processo si può usare una tecnica chiamata *addizione con salvataggio del riporto* o CSA (*carry-save addition*). Si

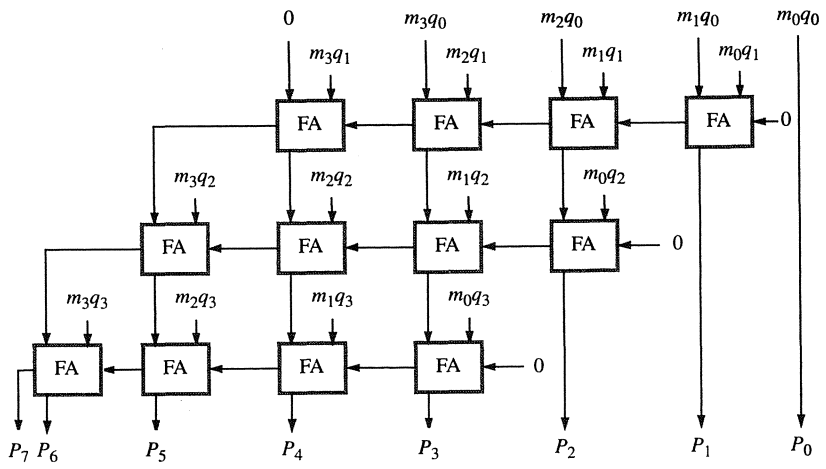
```
                0 1 1 0 1   (+13)
              × 1 1 0 1 0   (-6)
              -----------
                0 1 1 0 1
                0 -1 +1 -1 0
              -----------
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 0 1 1
0 0 0 0 1 1 0 1
1 1 1 0 0 1 1
0 0 0 0 0 0
-------------------------
1 1 1 0 1 1 0 0 1 0   (-78)

                0 1 1 0 1
                0   -1   -2
              -----------
1 1 1 1 1 0 0 1 1 0
1 1 1 1 0 0 1 1
0 0 0 0 0 0
-------------------------
1 1 1 0 1 1 0 0 1 0
```

Figura 9.15 Moltiplicazione che richiede solo $n/2$ addendi.

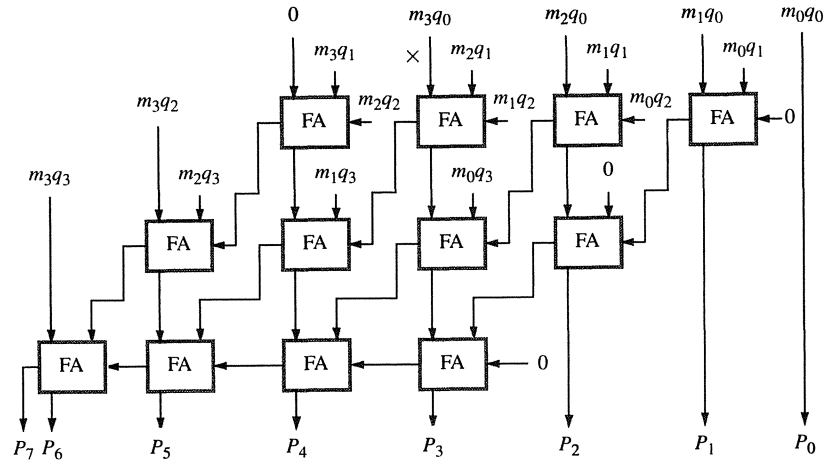
consideri la matrice di moltiplicazione 4×4 mostrata in Figura 9.16a. Questa struttura ha la forma della matrice mostrata in Figura 9.6, in cui la prima riga consiste delle sole porte AND e produce i quattro ingressi m_3q_0, m_2q_0, m_1q_0 e m_0q_0.

Invece di lasciare che i riporti si propaghino lungo le righe, essi possono essere "salvati" e introdotti nella prossima riga, alle posizioni di peso corretto, come mostrato in Figura 9.16b. Questo libera un ingresso per ciascuno dei tre addizionatori completi nella prima riga. Tali ingressi possono essere usati per introdurre i bit del terzo addendo m_2q_2, m_1q_2 e m_0q_2. Adesso due ingressi di ciascuno dei tre addizionatori completi nella seconda riga sono alimentati dalle uscite di somma e riporto dalla prima riga. Il terzo ingresso è usato per introdurre i bit m_2q_3, m_1q_3 e m_0q_3 del quarto addendo. I bit di ordine alto m_3q_2 e m_3q_3 del terzo e quarto addendo sono introdotti nei rimanenti ingressi liberi degli addizionatori completi all'estremità sinistra nella seconda e terza riga. I bit di riporto salvato e i bit di somma dalla seconda riga sono adesso sommati nella terza riga, che è un addizionatore a propagazione del riporto, per produrre i bit finali del prodotto.

Il ritardo attraverso la matrice con salvataggio del riporto è alquanto minore del ritardo attraverso la matrice con propagazione del riporto. Questo poiché i vettori S e C in uscita da ciascuna riga sono prodotti in parallelo, con il ritardo di un addizionatore completo. L'entità della riduzione di ritardo è considerata nel Problema proposto 9.15 nell'Eserciziario.



(a) Matrice a propagazione del riporto



(b) Matrice con salvataggio del riporto

Figura 9.16 Matrici a propagazione del riporto e con salvataggio del riporto per un moltiplicatore 4×4.

9.5.3 Albero di addizione degli addendi con riduttori 3-2

Una riduzione più significativa del ritardo può ottenersi quando si hanno operandi più lunghi di quelli considerati in Figura 9.16. Si possono raggruppare gli addendi in terne ed effettuare l'addizione con salvataggio del riporto su ciascuno di questi gruppi in parallelo per generare un in-

sieme di vettori S e C con un ritardo di addizionatore completo. Qui ci si riferisce al circuito addizionatore completo come a un addizionatore, per brevità. Successivamente si raggruppano tutti i vettori S e C in terne e si effettua la loro addizione con salvataggio del riporto, generando un altro gruppo di vettori S e C con un ulteriore ritardo di addizionatore. Si continua con questo processo finché non restano che due vettori. L'addizionatore in ciascuna posizione di bit dei tre addendi è chiamato *riduttore 3-2*, e la struttura del circuito logico che riduce un numero di addendi a due è chiamata *albero CSA*, come descritto da Wallace [2]. I due ultimi vettori S e C possono essere sommati in un addizionatore con anticipo del riporto per ottenere il prodotto desiderato.

Si consideri l'esempio mostrato in Figura 9.17. Esso richiede l'addizione di sei versioni ottenute da scorrimento del moltiplicando per il caso della moltiplicazione di due numeri senza segno da 6 bit, dove tutti i sei bit del moltiplicatore sono uguali a 1. I sei addendi, A, B, ..., F sono sommati mediante addizione con salvataggio del riporto in Figura 9.18. I rettangoli evidenziati in queste due figure indicano gli stessi bit degli operandi e mostrano come essi siano ridotti a bit di somma e riporto in Figura 9.18 dall'addizione con salvataggio del riporto. Sono effettuati tre livelli di addizione con salvataggio del riporto, come mostrato schematicamente in Figura 9.19. Questa figura mostra che i due vettori finali S_4 e C_4 sono disponibili con tre ritardi di addizionatore dopo che i sei addendi in ingresso sono applicati al livello 1. L'ultima operazione di addizione regolare su S_4 e C_4, che dà il prodotto, può essere compiuta con un addizionatore con anticipo del riporto.

Il ritardo del moltiplicatore è minore quando si usa la struttura ad albero illustrata in Figura 9.19 rispetto a quando si usa la struttura a matrice illustrata in Figura 9.16b. La riduzione del ritardo è tanto più significativa quanto più grande è il numero di addendi. Per esempio, l'addizione di 32 addendi nello schema mostrato in Figura 9.19 richiede solo 8 livelli di riduzione 3-2 prima dell'addizione finale. In generale, si dimostra che occorrono circa $1{,}7\log_2 k - 1{,}7$ livelli di riduzione 3-2 per ridurre k addendi a 2 vettori, che quando sommati danno il prodotto desiderato (si veda il Problema risolto 9.3 nell'Eserciziario).

È opportuno notare che addendi negativi sono in gioco quando si effettua la moltiplicazione di numeri con segno e si usa la ricodifica di Booth dei moltiplicatori. Questo richiede l'estensione del segno degli addendi prima che essi siano immessi nell'albero di riduzione. Inoltre, il numero di addendi da sommare si riduce se si usa la ricodifica bit-pair del moltiplicatore.

						1	0	1	1	0	1	(45)	M
					×	1	1	1	1	1	1	(63)	Q
						1	0	1	1	0	1	A	
					1	0	1	1	0	1		B	
				1	0	1	1	0	1			C	
			1	0	1	1	0	1				D	
		1	0	1	1	0	1					E	
	1	0	1	1	0	1						F	
1	0	1	1	0	0	0	1	0	0	1	1	(2,835)	Prodotto

Figura 9.17 Un esempio di moltiplicazione che illustra l'addizione con salvataggio del riporto mostrata in Figura 9.18.

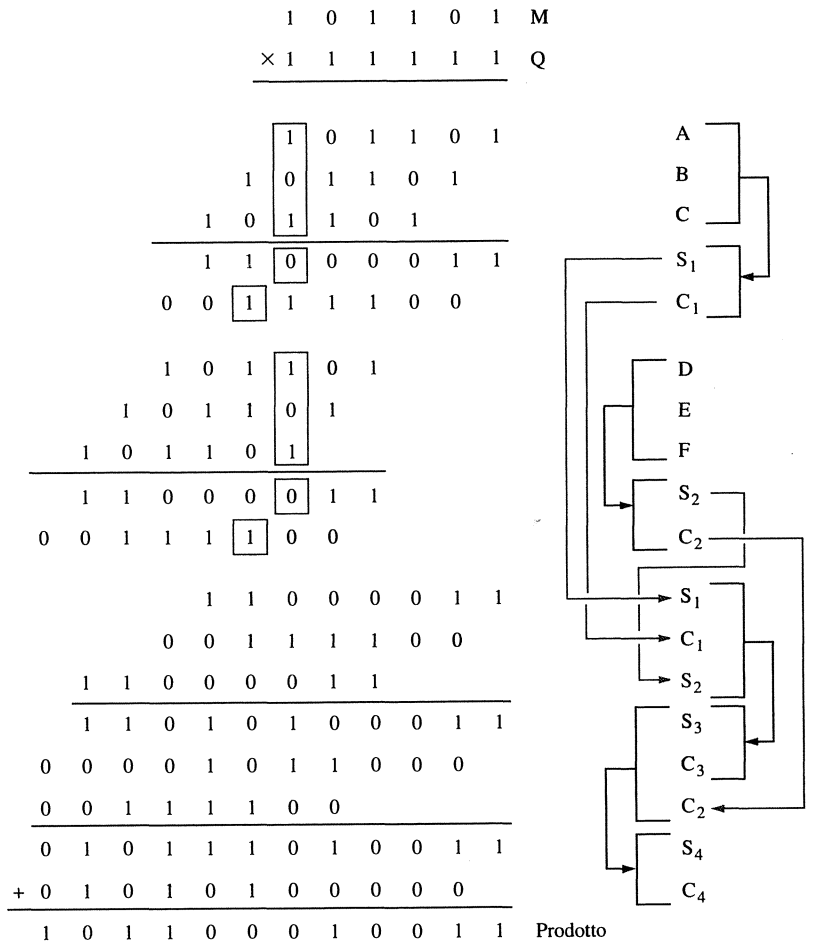


Figura 9.18 L'esempio di moltiplicazione in Figura 9.17 eseguita usando l'addizione con salvataggio del riporto.

Il riduttore 3-2 non è il solo circuito logico che si possa usare per costruire alberi di riduzione. È anche possibile usare riduttori 4-2 e riduttori 7-3. La prima di queste possibilità è descritta nel prossimo paragrafo e la seconda è esplorata nel Problema proposto 9.17 nell'Eserciziario.

9.5.4 Albero di addizione degli addendi con riduttori 4-2

Lo schema di interconnessione fra i livelli in un albero CSA che usa riduttori 3-2 è irregolare, come si può vedere in Figura 9.19. Un albero con struttura più regolare può ottenersi usando riduttori 4-2 [3], specialmente per il caso in cui il numero di addendi da ridurre sia una potenza

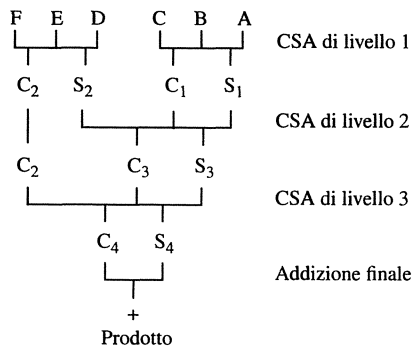


Figura 9.19 Rappresentazione schematica delle operazioni di addizione con salvataggio del riporto in Figura 9.18.

di 2. Questo caso è la norma per la moltiplicazione nella ALU di un processore. Per esempio, se 32 addendi sono ridotti a 2 usando riduttori 4-2 a ogni livello di riduzione, allora sono necessari solo quattro livelli. L'albero ha una struttura regolare, con 16, 8, 4 e 2 addendi alle uscite dei quattro livelli. Se si usano riduttori 3-2, occorrono otto livelli e le connessioni circuitali fra livelli sono piuttosto irregolari. Strutture ad albero regolari facilitano il disegno dei circuiti logici e dei collegamenti per la realizzazione di circuiti VLSI.

Si consideri la progettazione di un riduttore 4-2 come sviluppato in [3]. L'addizione di quattro bit di ugual peso, w, x, y e z, da quattro addendi produce un valore nell'intervallo da 0 a 4. Tale valore non può essere rappresentato da un bit di somma, s, e un singolo bit di riporto, c. Tuttavia, si può usare un secondo bit di riporto, c_{out}, con lo stesso peso di c, assieme a s e c per rappresentare qualsiasi valore da 0 a 5. Tanto basta per gli scopi attuali.

Non si vuole inviare tre bit di uscita giù al successivo livello di riduzione. Ciò realizzerebbe un riduttore 4-3, che riduce meno di un riduttore 3-2. La soluzione è inviare c_{out} lateralmente al riduttore 4-2 nella successiva posizione di peso più alto allo stesso livello di riduzione. Quindi, ciascun riduttore 4-2 deve avere un quinto ingresso, c_{in}, connesso all'uscita c_{out} del riduttore 4-2 nella posizione adiacente di peso più basso allo stesso livello di riduzione.

Un requisito finale sulla progettazione del riduttore 4-2 è che il valore di c_{out} non dipenda dal valore di c_{in}. Questo è un requisito essenziale. Senza di esso i riporti si propagherebbero lateralmente lungo un livello di riduzione, annullando lo scopo della riduzione parallela degli addendi con ritardo basso e fissato. Un blocco riduttore 4-2 è mostrato in Figura 9.20.

Riassumendo, la specifica per un riduttore 4-2 è come segue.

- Le tre uscite, s, c e c_{out}, rappresentano la somma aritmetica di cinque ingressi, ovvero:

$$w + x + y + z + c_{in} = s + 2(c + c_{out})$$

  dove tutti gli operatori sono aritmetici.
- L'uscita s è la solita variabile somma, ovvero s è la funzione XOR delle cinque variabili in ingresso.
- Il riporto laterale, c_{out}, deve essere indipendente da c_{in}. È funzione solo delle quattro variabili in ingresso w, x, y e z.

Vi sono diverse possibilità per specificare le due uscite del riporto in modo da soddisfare le condizioni date. Se ne presenta una facile da descrivere. Primo, si pone a 1 l'uscita del

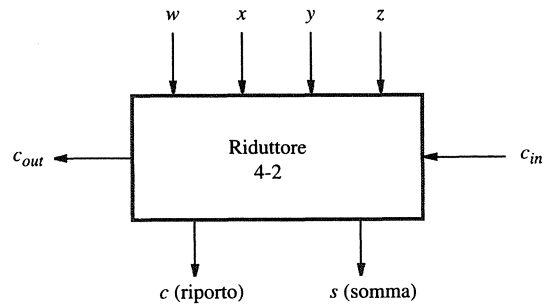


Figura 9.20 Un blocco riduttore 4-2.

riporto laterale, c_{out}, quando due o più delle variabili di ingresso w, x, y e z sono uguali a 1. Quindi, l'altra uscita del riporto, c, è determinata in modo da soddisfare la condizione aritmetica. La tabella di verità completa che soddisfa queste condizioni è mostrata in Figura 9.21, in una forma che differisce da quella usuale, impiegata in Appendice. I quattro ingressi

				$c_{in} = 0$		$c_{in} = 1$		
w	x	y	z	c	s	c	s	c_{out}
0	0	0	0	0	0	0	1	0
0	0	0	1	0	1	1	0	0
0	0	1	0	0	1	1	0	0
0	1	0	0	0	1	1	0	0
1	0	0	0	0	1	1	0	0
0	0	1	1	0	0	0	1	1
0	1	0	1	0	0	0	1	1
0	1	1	0	0	0	0	1	1
1	0	0	1	0	0	0	1	1
1	0	1	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
0	1	1	1	0	1	1	0	1
1	0	1	1	0	1	1	0	1
1	1	0	1	0	1	1	0	1
1	1	1	0	0	1	1	0	1
1	1	1	1	1	0	1	1	1

Figura 9.21 Tabella di verità di un riduttore 4-2.

w, x, y e z non sono elencati in ordine numerico binario, bensì in gruppi che corrispondono al numero di ingressi che hanno il valore 1. Questo rende più facile vedere come le uscite sono specificate per soddisfare le condizioni date. Dalla tabella si può derivare una rete di porte logiche.

9.5.5 Sommario della moltiplicazione veloce

Si riassumono ora le tecniche per la moltiplicazione ad alta velocità. Si può usare la ricodifica bit-pair del moltiplicatore, derivata dall'algoritmo di Booth, per ridurre inizialmente il numero di addendi di un fattore due. Gli addendi risultanti possono quindi essere ridotti a due in un albero di riduzione con un numero relativamente piccolo di livelli di riduzione. Il prodotto finale può essere generato dall'esecuzione di un'addizione che usa un addizionatore con anticipo del riporto. Tutte e tre queste tecniche – ricodifica bit-pair del moltiplicatore, riduzione parallela degli addendi, addizione con anticipo del riporto – sono state usate in varie combinazioni dai progettisti di processori ad alte prestazioni per ridurre il tempo necessario per le moltiplicazioni.

9.6 Divisione di interi

Nel Paragrafo 9.3 si è discusso della moltiplicazione di numeri senza segno confrontando il modo in cui l'operazione di moltiplicazione si esegue manualmente con il modo in cui si esegue in un circuito logico. Qui si segue lo stesso approccio nel discutere la divisione di interi. Si discute la divisione di numeri senza segno in dettaglio, quindi si inserisce qualche commento generale sul caso dei numeri con segno.

La Figura 9.22 mostra esempi di divisione decimale e binaria degli stessi valori. Si consideri prima la versione decimale. Il 2 nel quoziente è determinato dal seguente ragionamento: prima, si prova a dividere 2 per 13, il che non è possibile. Quindi, si prova a dividere 27 per 13. Si procede nel tentativo moltiplicando 13 per 2, che dà 26, e osservando che 27 − 26 = 1 è meno di 13, si pone 2 a quoziente e si svolge la sottrazione richiesta. Si abbassa la prossima cifra del dividendo, 4, e si finisce decidendo che 13 sta in 14 una volta con il resto di 1. È possibile discutere la divisione binaria in modo simile, con la semplificazione che le sole possibilità per i bit del quoziente sono 0 e 1.

Un circuito che realizza la divisione con questo metodo manuale opera come segue: posiziona il divisore in modo appropriato rispetto al dividendo e svolge una sottrazione. Se il resto è zero o positivo, si determina 1 per un bit del quoziente, si estende il resto con un altro

```
        21                    10101
13 ) 274          1101 ) 100010010
     26                    1101
      14                    10000
      13                     1101
       1                      1110
                              1101
                                 1
```

Figura 9.22 Esempi di divisione manuale.

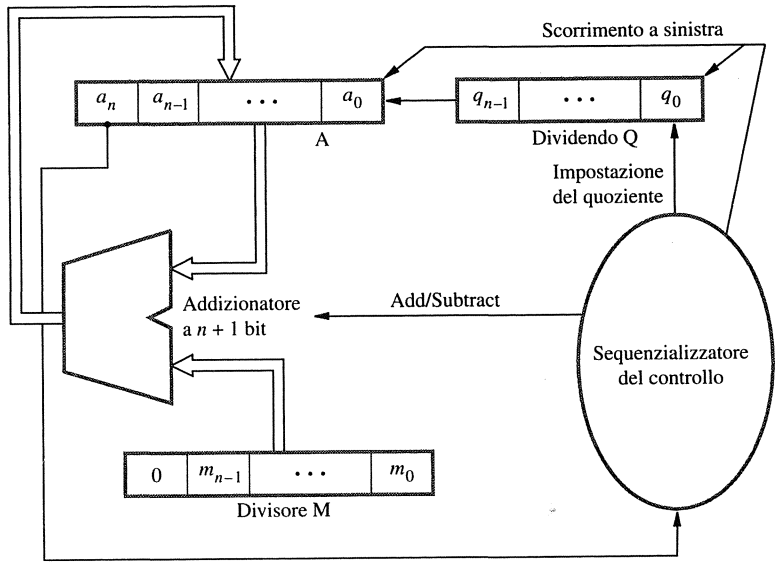


Figura 9.23 Schema di circuito per la divisione binaria.

bit del dividendo, si riposiziona il divisore e si procede svolgendo un'altra sottrazione. Se il resto è negativo, si determina 0 per il bit del quoziente, si ripristina il dividendo sommandogli il divisore appena sottratto e si riposiziona il divisore per un'altra sottrazione. Questo è detto algoritmo di *divisione con ripristino* (*restoring division*).

Divisione con ripristino

La Figura 9.23 mostra una sistemazione del circuito logico che realizza l'algoritmo di divisione con ripristino appena discusso. Si noti la similarità con la struttura per la moltiplicazione mostrata in Figura 9.7. Un divisore positivo da n bit viene caricato nel registro M e un dividendo positivo da n bit è caricato nel registro Q all'inizio dell'operazione. Il registro A è posto a 0. Dopo che la divisione è completa, il quoziente a n bit è nel registro Q e il resto è nel registro A. Le sottrazioni richieste sono facilitate dall'uso dell'aritmetica in complemento a due. La posizione del bit extra all'estremo sinistro di A come di M accoglie il bit di segno durante le sottrazioni. L'algoritmo seguente esegue la divisione con ripristino.

Eseguire n volte i seguenti tre passi:

1. Fare scorrere A e Q a sinistra di una posizione.
2. Sottrarre M da A e porre il risultato in A.
3. Se il segno di A è 1, porre q_0 a 0 e sommare M ad A (ovvero, ripristinare A); altrimenti, porre q_0 a 1.

La Figura 9.24 mostra come un esempio da 4 bit sarebbe elaborato dal circuito in Figura 9.23.

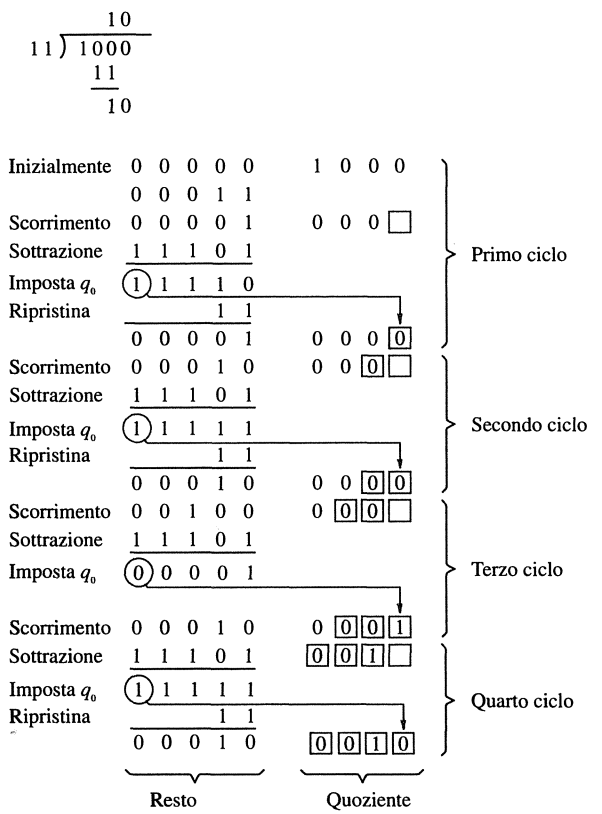


Figura 9.24 Un esempio di divisione con ripristino.

Divisione senza ripristino

L'algoritmo di divisione con ripristino può essere migliorato per evitare la necessità del ripristino di A dopo una sottrazione senza successo. La sottrazione è detta senza successo se il risultato è negativo. Si consideri la sequenza di operazioni svolte dopo l'esecuzione della sottrazione nell'algoritmo precedente. Se A è positivo, lo si fa scorrere a sinistra e se ne sottrae M, ovvero si esegue $2A - M$. Se A è negativo, lo si ripristina eseguendo $A + M$ e quindi lo si fa scorrere a sinistra e se ne sottrae M. Questo equivale a eseguire $2A + M$. Il bit q_0 è posto al valore appropriato dopo l'esecuzione dell'operazione corretta (si veda la Figura 9.24). Si può riassumere tutto questo nel seguente algoritmo di *divisione senza ripristino*.

Stadio 1: Eseguire n volte i seguenti due passi.

1. Se il segno di A è 0, fare scorrere A e Q a sinistra di una posizione e sottrarre M da A; altrimenti, fare scorrere A e Q a sinistra di una posizione e sommare M ad A.
2. Se adesso il segno di A è 0, porre q_0 a 1; altrimenti, porre q_0 a 0.

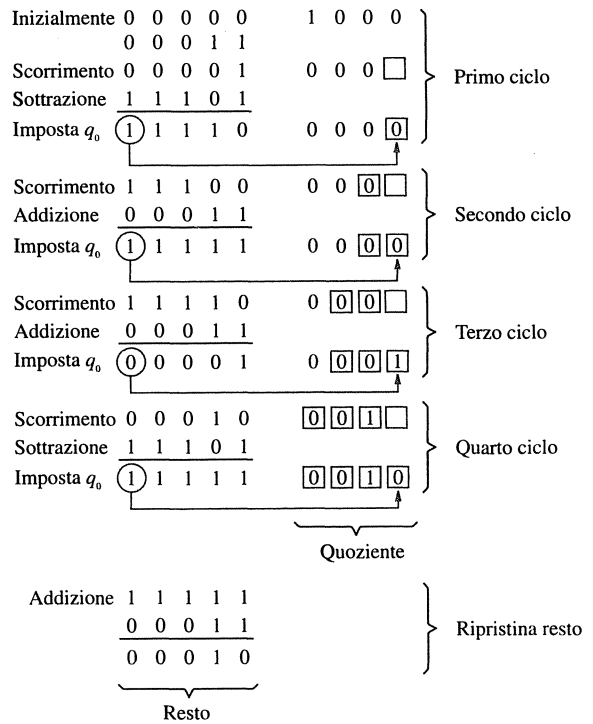


Figura 9.25 Un esempio di divisione senza ripristino.

Stadio 2: Se il segno di A è 1, sommare M ad A.

Lo Stadio 2 è necessario per lasciare il resto positivo appropriato in A dopo gli n cicli dello Stadio 1. Il circuito logico in Figura 9.23 può essere usato per eseguire anche questo algoritmo, eccetto che le operazioni di Ripristino non sono più necessarie. Viene eseguita una delle operazioni Add o Subtract in ciascuno degli n cicli dello Stadio 1, più una possibile addizione finale nello Stadio 2. La Figura 9.25 mostra come si esegue l'esempio di divisione in Figura 9.24 usando l'algoritmo di divisione senza ripristino.

Non vi sono algoritmi semplici per effettuare direttamente la divisione su operandi con segno, che siano confrontabili agli algoritmi per la moltiplicazione con segno. Per la divisione, si possono elaborare inizialmente gli operandi per mutarli in valori positivi. Dopo avere usato uno degli algoritmi appena discussi, i segni di quoziente e resto vanno corretti come necessario.

9.7 Numeri e operazioni in virgola mobile

Il Capitolo 1 ha fornito la motivazione per usare numeri in virgola mobile e ha indicato come li si possa rappresentare in un formato binario da 32 bit. In questo capitolo si danno più

dettagli su formati di rappresentazione e operazioni aritmetiche sui numeri in virgola mobile. Le descrizioni fornite qui sono basate sulla versione del 2008 dello Standard IEEE 754, etichettato 754-2008 [4].

Si richiama dal Capitolo 1 che un numero binario in virgola mobile può essere rappresentato da:

- un segno per il numero;
- alcuni bit significativi;
- un esponente con segno per un fattore di scala con implicita base 2.

Il più semplice formato IEEE 754 di virgola mobile è appunto quello a 32 bit, chiamato *precisione singola* e mostrato in Figura 9.26a. La codifica del numero è formata da tre campi di

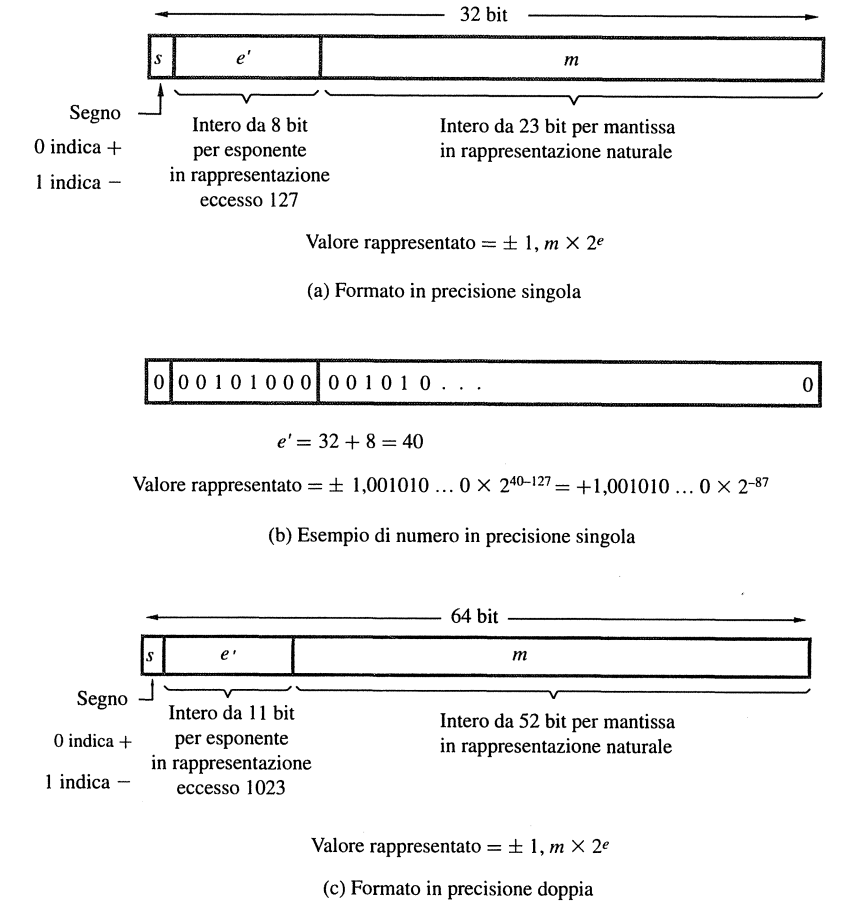


Figura 9.26 Formati di numero in virgola mobile secondo lo standard IEEE 754.

bit concatenati, vale a dire $se'm$, accomodati nella parola da 32 bit. Il valore di tale rappresentazione è dato dalla formula seguente:

$$\mathrm{val}(se'm) = \pm f \times 2^e = \begin{cases} +1{,}m \times 2^{e'-127} & \text{se } s = 0 \\ -1{,}m \times 2^{e'-127} & \text{se } s = 1 \end{cases} \tag{9.4}$$

dove $f = 1{,}m$ ed $e = e' - 127$.

Il bit s rappresenta il segno e si interpreta come indicato nella formula (9.4). Il resto della codifica, vale a dire $e'm$, dà il valore assoluto del numero frazionario. Si noti come in pratica si stia usando la tecnica di segno e valore assoluto.

Gli 8 bit dell'esponente rappresentano un numero intero e', compreso nell'intervallo $0 \leq e' \leq 255$ e dunque sempre positivo o nullo, che non è l'esponente effettivo da usare nella valutazione. I due valori estremi $e' = 0$ ed $e' = 255$ sono usati per scopi speciali, pertanto l'intervallo si restringe a $1 \leq e' \leq 254$. L'esponente effettivo e si ottiene mediante la formula $e = e' - 127$ e, tolti i valori speciali di e', sta nell'intervallo $-126 \leq e \leq 127$. Tale tecnica di codifica si chiama *eccesso 127* e ha il vantaggio di semplificare il confronto dei valori di due numeri in virgola mobile (si veda il Problema proposto 9.23 nell'Eserciziario).

I 23 bit rimanenti servono per codificare il fattore intero, denotato f. Si suppone che il numero sia in forma normale e poiché le cifre sono bit il fattore intero deve avere sempre 1 come bit più significativo, cioè ha la forma seguente:

$$f = \underbrace{1, \underbrace{\textit{stringa di bit } 0 \textit{ e } 1}_{\text{mantissa}}}_{\text{fattore intero}}$$

Ma il bit 1 iniziale non si esprime mai esplicitamente (sarebbe del tutto inutile) e si deve solo immaginare che sia costantemente presente a sinistra della virgola; per tale motivo si chiama *bit nascosto* (*hidden bit*). Di conseguenza i 23 bit rappresentano solo la mantissa m del numero. Il valore del fattore intero è allora $f = 1{,}m$, ottenuto giustapponendo la mantissa m a destra del bit nascosto iniziale 1 seguito dalla virgola. Ciò completa la spiegazione della formula (9.4). La Figura 9.26b mostra un esempio di rappresentazione in virgola mobile in precisione singola, dove si usa la formula (9.4) per valutare il numero frazionario così codificato.

Si noti che nel caso binario in forma normale la mantissa da sola permette di dedurre il fattore intero. Ciò però non sarebbe possibile in ogni altra base diversa da due perché la mantissa da sola non permetterebbe di ritrovare la cifra significativa a sinistra della virgola. La mantissa è un concetto matematico precedente di secoli il calcolatore, essendo stata in origine ideata per dare forma sintetica alle tavole di calcolo del logaritmo decimale. In alcuni testi si estende il nome mantissa a tutto il fattore intero seguendo proprio la struttura di tali tavole, ma qui per maggiore chiarezza si è preferito distinguere.

Il formato di virgola mobile in precisione singola, ovvero a 32 bit, occupa in genere una sola parola di memoria nel calcolatore. Il fattore di scala sta nell'intervallo $[2^{-126}, 2^{127}]$, che in decimale corrisponde approssimativamente a $10^{\pm 38}$, e la mantissa da 23 bit corrisponde a circa 7 cifre decimali di fattore intero. Per avere precisione più elevata lo standard IEEE 754 prevede anche il formato in *precisione doppia*, mostrato in Figura 9.26c. Si usano in totale 64 bit: un bit per il segno, 11 bit per l'esponente e 52 bit per la mantissa. L'esponente è rappresentato in eccesso 1.023. Come prima, i due valori $e' = 0$ ed $e' = 1.023$ sono speciali e dunque l'esponente effettivo e sta nell'intervallo $-1.022 \leq e \leq 1.023$. Il fattore di scala sta nell'intervallo $[2^{-1.022}, 2^{1.023}]$, che in decimale corrisponde approssimativamente a $10^{\pm 308}$ (un intervallo enorme). La mantissa da 52 bit corrisponde a circa 16 cifre decimali di fattore intero.

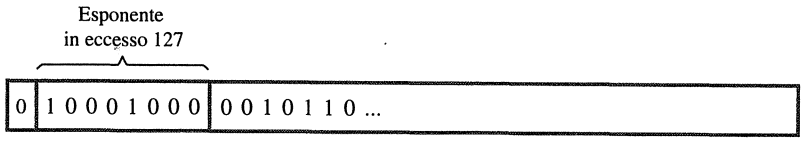
Il calcolatore conforme allo standard IEEE 754 deve disporre almeno del formato in precisione singola, mentre quello in precisione doppia è facoltativo. Lo standard illustra e precisa anche alcune estensioni ai due formati base, pure facoltative. Oltre alle quattro operazioni aritmetiche fondamentali, di prima specie (addizione e sottrazione) e di seconda specie (moltiplicazione e divisione), lo standard impone che ne siano disponibili altre come il calcolo di quoziente e resto di interi, radice quadrata intera e conversioni tra binario e decimale.

Il numero in virgola mobile dovrebbe essere sempre in forma normale. Quando non fosse così lo si può comunque *rimettere in forma normale* o *normalizzare* facendo scorrere verso sinistra la mantissa fino a fare emergere 1 come primo bit (che poi si toglie per lasciarlo sottinteso) e decrementando di conseguenza l'esponente. La Figura 9.27 mostra un numero in forma non normale, $0{,}0010110\ldots \times 2^9$ (a), e la forma normale corrispondente, $1{,}0110\ldots 0 \times 2^6$ (b): si fa scorrere la mantissa di tre posizioni verso sinistra e si decrementa l'esponente di tre unità. In generale si compensa lo scorrimento della mantissa verso destra o sinistra incrementando o decrementando l'esponente, rispettivamente.

Durante i calcoli in virgola mobile può succedere che si producano valori intermedi non rappresentabili nel formato correntemente in uso. In precisione singola varrebbe a dire che comparissero numeri con esponente effettivo e minore di -126 o maggiore di 127. Questo è il fenomeno di *trabocco verso il basso* o *verso l'alto* (*underflow* o *overflow*), rispettivamente. Entrambi gli eventi costituiscono eccezione di tipo aritmetico e vanno trattati come si dirà tra breve.

Valori Speciali

I valori estremi dell'esponente hanno significato speciale, come si è accennato prima. Quando si hanno valori speciali la formula (9.4) non si applica, bensì si dà un'interpretazione *ad hoc*



$e' = 128 + 8 = 136$

(A sinistra della virgola non c'è il bit 1 implicito)
Valore rappresentato $= +\,0{,}0010110\ldots \times 2^{136-127} = +\,0{,}0010110\ldots \times 2^9$

(a) Prima di normalizzare

$e' = 128 + 4 + 1 = 133$

Valore rappresentato $= +\,1{,}0110\ldots\ldots 0 \times 2^{133-127} = +\,1{,}0110\ldots\ldots 0 \times 2^6$

(b) Dopo avere normalizzato

Figura 9.27 Come normalizzare in precisione singola.

dipendente dal valore speciale in questione. Si prenda il caso di precisione singola. Con $e' = 0$ e $m = 0$ si rappresenta il valore 0 esatto. Infatti la forma normale è incapace di esprimere lo zero esatto [si riveda la formula (9.4)] perché il fattore intero è sempre $f = \pm 1, m \neq 0$ e il fattore di scala 2e, essendo una potenza, non si annulla mai; ma non potere rappresentare zero sarebbe evidentemente intollerabile. Con $e' = 255$ e $m = 0$ si rappresenta l'infinito ∞, visto come risultato della divisione per 0; il bit di segno s precisa se sia da intendere $-\infty$ o $+\infty$. Tutto ciò vale anche in precisione doppia aggiustando i valori estremi di e' secondo il caso.

Invece con $e' = 0$ e $m \neq 0$ si rappresentano numeri in *forma non normale* o *denormalizzata*. In precisione singola la rappresentazione si interpreta così: $\pm 0{,}m \times 2^{-126}$, dove $\pm$ è discriminato dal bit s; si noti che qui a e si assegna d'ufficio il valore valido minimo. Si procede similmente per la precisione doppia sostituendo -1.022 a -126. Pertanto in entrambi i casi il fattore di scala è il minimo possibile. Tuttavia l'aspetto importante della forma non normale è l'assenza del bit nascosto 1 implicito a sinistra della virgola. Ne segue che in valore assoluto il numero in forma non normale è sempre inferiore al minimo numero rappresentabile in forma normale, vale a dire che la forma non normale rappresenta valori assoluti piccolissimi. Il numero in forma non normale sembra un oggetto bizzarro, ma assolve alla funzione di *azzeramento graduale* (*gradual underflow*). In pratica la forma non normale amplia l'insieme di valori rappresentabili in virgola mobile e permette di trattare con successo talune situazioni dove compaiono numeri, positivi o negativi, vicinissimi allo zero esatto. Detto altrimenti i numeri in forma non normale costituiscono una sorta di *intorno dello zero*, dove si addensano in modo particolarmente fitto (simmetricamente). Ciò è utile per gestire in modo flessibile problemi di approssimazione senza perdita di informazione, e simili.

Infine con $e' = 255$ e $m \neq 0$, e similmente in precisione doppia con $e' = 2.047$, il valore va interpretato come *entità senza significato numerico* o *NaN* (*Not a Number*). Ciò è utile per segnalare talune operazioni che non producono risultato o che non hanno risultato univoco, oppure il cui risultato non è pensabile come numero reale. Per esempio, lo standard IEEE 754 stabilisce le convenzioni seguenti: $0/0 = NaN$, $\infty/\infty = NaN$, $\sqrt[2]{-1} = NaN$, e altre del genere.

Trattamento di Eccezione

Il processore conforme allo standard IEEE 754 (o meglio la sua unità ALU) deve aggiornare uno o più bit di esito appositi, chiamati *bit di eccezione aritmetica* (*arithmetic exception flag*), ogniqualvolta si verifichi una delle situazioni eccezionali seguenti: *trabocco verso l'alto* o *il basso*, *divisione per zero*, *inesattezza* o *invalidità*. Le prime tre eccezioni sono già state in parte spiegate. Se per ricondurre il numero a valore rappresentabile occorre arrotondare si ha *inesattezza*. Se si tenta un'operazione come $0/0$, ∞/∞ o $\sqrt[2]{-1}$, e simili, si ha *invalidità*. *Quando si verifica eccezione il risultato è sempre un valore speciale.*

Ai bit di eccezione si possono associare *interruzioni* (*interrupt*) (si veda il Capitolo 3). Non appena il bit di eccezione viene aggiornato scatta il meccanismo di interruzione che forza la messa in esecuzione di una routine apposita di utente o anche di sistema operativo, in grado di rimediare in qualche modo alla situazione di eccezione o almeno di impedire che le conseguenze dell'errore si propaghino. In alternativa si può lasciare al programma applicativo il compito di provvedere direttamente correggendo o aggirando l'errore in qualche modo.

9.7.1 Operazioni Fondamentali

Lo standard IEEE 754 precisa con cura gli algoritmi di addizione, sottrazione, moltiplicazione e divisione per numeri in virgola mobile. Qui se ne daranno le versioni per il formato in precisione singola. Gli algoritmi sono descritti in modo sintetico, specificando solo i passi importanti ma omettendo le situazioni di eccezione come le varie forme di trabocco. Tra i vari

dettagli omessi c'è anche il fatto che nelle fasi intermedie dell'algoritmo talvolta occorrono più bit di quanti non ne siano concessi dal formato in uso e se ne ritrovino nel risultato finale. Tali bit aggiuntivi intermedi servono per regolare e limitare l'errore di arrotondamento, nei limiti del possibile. Naturalmente siffatti dettagli vanno considerati quando si progettano circuiti digitali per calcoli in virgola mobile e infatti lo standard IEEE 754 li precisa tutti in modo completo.

Negli algoritmi riportati sotto si presuppone che gli operandi siano dati in forma normale e alla fine se occorre il risultato viene messo in forma normale. Data la mantissa di un operando l'algoritmo ricostruisce il fattore intero del numero, rimettendole in testa il bit nascosto 1, e poi procede lavorando su quest'ultimo.

Algoritmo di addizione algebrica

1. Scegli il numero con esponente più piccolo e fai scorrere la sua mantissa a destra di un numero di passi uguale alla differenza degli esponenti.
2. Poni l'esponente del risultato uguale all'esponente più grande.
3. Addiziona i fattori interi degli addendi ottenendo segno e fattore intero provvisorio della somma.
4. Se il numero risultante non è in forma normale, normalizzalo.

Moltiplicazione e divisione sono alquanto più semplici di addizione e sottrazione, poiché non è necessario allineare le mantisse.

Algoritmo di moltiplicazione

1. Addiziona gli esponenti dei fattori e togli 127 alla somma, ottenendo l'esponente provvisorio del prodotto.
2. Moltiplica i fattori interi dei fattori ottenendo segno e fattore intero provvisorio del prodotto.
3. Se il numero risultante non è in forma normale, normalizzalo.

La divisione di numeri frazionari coincide con quella di numeri reali, con un solo risultato – il rapporto – a differenza della divisione intera che ha quoziente e resto.

Algoritmo di divisione

1. Sottrai l'esponente del divisore dall'esponente del dividendo e aggiungi 127 alla differenza, ottenendo l'esponente provvisorio del rapporto.
2. Calcola il quoziente del fattore intero del dividendo rispetto a quello del divisore, ottenendo segno e fattore intero provvisorio del rapporto.
3. Se il numero risultante non è in forma normale, normalizzalo.

In tutti e tre gli algoritmi occorre la normalizzazione finale del numero per ridare al fattore intero la forma normale aggiustando l'esponente provvisorio, ottenendo infine mantissa ed esponente del risultato.

9.7.2 Bit di guardia e troncamento

Si considerino alcuni aspetti importanti della realizzazione dei passi dei precedenti algoritmi. Sebbene le mantisse degli operandi iniziali e dei risultati finali siano limitate a 24 bit, includendo il bit nascosto, è importante tenere bit aggiuntivi, spesso chiamati bit di *guardia*, durante i passi intermedi. Ciò dà la massima accuratezza nei risultati finali.

La rimozione dei bit di guardia nella generazione di un risultato finale richiede che la mantissa estesa sia *troncata* per creare un numero a 24 bit che approssima la versione più lunga. Questa operazione interviene anche in altre situazioni, per esempio quando si convertono numeri da decimale a binario. Si dovrebbe dire che il termine generale arrotondamento è anche usato per l'operazione di troncamento, ma qui si usa una definizione più restrittiva di arrotondamento come una delle forme di troncamento.

Vi sono vari modi di troncare. Il modo più semplice è rimuovere i bit di guardia e non apportare nessun cambiamento ai bit ritenuti. Questo è detto *spezzamento* (*chopping*). Si supponga di voler troncare una frazione da sei a tre bit con questo metodo. Tutte le frazioni nell'intervallo da $0{,}b_{-1}b_{-2}b_{-3}000$ a $0{,}b_{-1}b_{-2}b_{-3}111$ vengono troncate a $0{,}b_{-1}b_{-2}b_{-3}$. L'errore nel risultato da 3 bit va da 0 a 0,000111. In altre parole, l'errore con lo spezzamento va da 0 a quasi 1 nella posizione meno significativa dei bit ritenuti. Nell'esempio, questa è la posizione b_{-3}. Il risultato dello spezzamento è un'approssimazione *distorta* (*biased*) poiché l'intervallo dell'errore non è simmetrico rispetto a 0.

Il successivo metodo più semplice è l'*arrotondamento di von Neumann*. Se i bit da rimuovere sono tutti 0, questi vengono semplicemente rimossi, senza cambiamenti ai bit ritenuti. Tuttavia, se uno dei bit rimossi è 1, il bit meno significativo dei bit ritenuti è posto a 1. Nell'esempio di troncamento da 6 bit a 3 bit, tutte le frazioni di 6 bit con $b_{-4}b_{-5}b_{-6}$ diverso da 000 sono troncati a $0{,}b_{-1}b_{-2}1$. L'errore in questo metodo di troncamento varia fra -1 e $+1$ nella posizione LSB dei bit ritenuti. Sebbene l'intervallo di errore sia più grande con questa tecnica rispetto a quanto lo è con lo spezzamento, l'ampiezza massima è la stessa, e l'approssimazione è *senza distorsione* poiché l'intervallo di errore è simmetrico rispetto a 0.

Approssimazioni senza distorsione sono vantaggiose se molti operandi e operazioni intervengono nella generazione di un risultato, poiché errori positivi tendono a bilanciare errori negativi man mano che procede il calcolo. Statisticamente possiamo aspettarci che i risultati di un'elaborazione complessa siano più accurati.

Il terzo metodo di troncamento è una procedura di *arrotondamento*. L'arrotondamento ottiene l'approssimazione più vicina al numero soggetto a troncamento ed è una tecnica senza distorsione. La procedura è come segue: si aggiunge 1 alla posizione LSB dei bit da ritenere se c'è 1 nella posizione MSB dei bit che vengono rimossi. Quindi, $0{,}b_{-1}b_{-2}b_{-3}1\ldots$ è arrotondato a $0{,}b_{-1}b_{-2}b_{-3} + 0{,}001$, e $0{,}b_{-1}b_{-2}b_{-3}0\ldots$ è arrotondato a $0{,}b_{-1}b_{-2}b_{-3}$. Questo fornisce l'approssimazione desiderata, eccetto nel caso in cui i bit da rimuovere siano 10 ... 0. Questa è una situazione di pareggio; il valore più lungo è a metà fra le due rappresentazioni troncate più vicine. Per risolvere il pareggio senza distorsioni, una possibilità è scegliere che i bit ritenuti siano il numero pari più vicino. In termini dell'esempio a 6 bit, il valore $0{,}b_{-1}b_{-2}0100$ è troncato al valore $0{,}b_{-1}b_{-2}0$, mentre $0{,}b_{-1}b_{-2}1100$ è troncato a $0{,}b_{-1}b_{-2}1 + 0{,}001$. La frase "arrotondare al numero più vicino o al numero pari più vicino nel caso di pareggio" è usata qualche volta per riferirsi a questa tecnica di troncamento. L'intervallo di errore è approssimativamente da $-1/2$ a $+1/2$ nella posizione LSB dei bit ritenuti. Chiaramente, questo è il metodo migliore. Tuttavia, è anche il più difficile da realizzare poiché richiede un'operazione di somma e una possibile rinormalizzazione. Questa tecnica di arrotondamento è specificata quale modo di default per il troncamento nello standard IEEE per la virgola mobile. Lo standard specifica inoltre altri metodi di troncamento, e si riferisce a essi come modi di arrotondamento.

Questa discussione sugli errori introdotti quando il troncamento rimuove i bit di guardia ha trattato il caso di una singola operazione di troncamento. Per una lunga serie di calcoli con numeri in virgola mobile, l'analisi che determina intervalli o limiti di errore per i risultati finali può essere uno studio complicato. Non si discute ulteriormente questo aspetto del calcolo numerico, tranne che per alcuni commenti sul modo in cui si trattano i bit di guardia e l'arrotondamento nello standard IEEE per la virgola mobile.

Secondo lo standard, i risultati di singole operazioni devono essere calcolati in modo da essere accurati entro la metà di una unità nella posizione LSB. Ciò implica che si deve usare l'arrotondamento come metodo di troncamento. La realizzazione dell'arrotondamento richiede il mantenimento di solo tre bit di guardia durante i passi intermedi nella realizzazione di un'operazione. I primi due di questi bit sono i bit più significativi della sezione della mantissa da rimuovere. Il terzo bit è l'OR logico di tutti i bit al di là di questi primi due bit nella rappresentazione piena della mantissa. Questo bit è relativamente facile da mantenere nei passi intermedi delle operazioni da fare. Esso va inizializzato a 0. Se si fa scorrere un 1 attraverso questa posizione mentre si allineano le mantisse, il bit diventa 1 e mantiene questo valore; quindi è usualmente chiamato *sticky bit* (bit appiccicoso).

9.7.3 Realizzazione di operazioni in virgola mobile

La realizzazione in hardware di operazioni in virgola mobile coinvolge una quantità considerevole di circuiti logici. Queste operazioni possono anche essere realizzate da routine software. In entrambi i casi, il calcolatore deve essere in grado di convertire ingressi e uscite da e verso la rappresentazione decimale dei numeri, più conveniente per l'utente. In molti processori di uso generale, le operazioni in virgola mobile sono disponibili a livello di istruzioni macchina, realizzate in hardware.

Un esempio di realizzazione di operazioni in virgola mobile è mostrato in Figura 9.28. Questo è un diagramma a blocchi di una realizzazione hardware per addizionare e sottrarre operandi in virgola mobile a 32 bit che hanno il formato mostrato in Figura 9.26a. Seguendo l'algoritmo di addizione algebrica dato nel Paragrafo 9.7.1, si vede che il primo passo è confrontare gli esponenti per determinare di quanto far scorrere la mantissa del numero con l'esponente più piccolo. Il valore di conteggio dello scorrimento, n, è determinato dal circuito sottrattore a 8 bit nell'angolo in alto a sinistra della figura. Il valore assoluto n della differenza $E'_A - E'_B$ è inviato all'unità SHIFTER, che attua lo scorrimento. Se n è maggiore del numero di bit significativi degli operandi, allora la risposta è essenzialmente l'operando più grande (a meno della considerazione dei bit di guardia e dello sticky bit ai fini dell'arrotondamento), e si possono prendere delle scorciatoie per derivare il risultato. Non si esplora questo aspetto in dettaglio.

Il segno della differenza che risulta dal confronto degli esponenti determina quale mantissa si deve far scorrere. Quindi, nel passo 1, il segno è inviato alla rete SCAMBIO nell'angolo in alto a destra della Figura 9.28. Se il segno è 0, allora $E'_A \geq E'_B$ e le mantisse M_A e M_B sono inoltrate direttamente dalla rete SCAMBIO. Questo ha come effetto che M_B sia inviata allo SHIFTER, per scorrere di n posizioni a destra: sia M'_B il suo output in questo caso. L'altra mantissa, M_A, è inviata direttamente all'addizionatore/sottrattore di mantissa. Se il segno è 1, allora $E'_A < E'_B$ e le mantisse vengono scambiate prima dell'invio. Sia M'_A l'output dello SHIFTER in questo caso.

Il passo 2 è realizzato dal multiplatore a due vie, MUX, vicino all'angolo in basso a sinistra della figura. L'esponente del risultato, E', è provvisoriamente determinato come E'_A se $E'_A \geq E'_B$, o E'_B se $E'_A < E'_B$, in base al segno della differenza che risulta dal confronto degli esponenti al passo 1.

Il passo 3 mette in gioco il componente principale, l'addizionatore/sottrattore di mantisse al centro della figura. La logica CONTROLLO determina se le mantisse vanno sommate o sottratte. Questo si decide in base ai segni degli operandi (S_A e S_B) e all'operazione (Add o Subtract) da effettuare sugli operandi. La logica CONTROLLO determina anche il segno del risultato, S_R. Per esempio, se A è negativo ($S_A = 1$), B è positivo ($S_B = 0$) e l'operazione è $A - B$, allora le mantisse sono sommate e il segno del risultato è negativo ($S_R = 1$). D'altra parte, se A e B sono entrambi positivi e l'operazione è $A - B$, allora si sottraggono le mantisse. Il segno del risultato, S_R, ora dipende dall'operazione di sottrazione delle mantisse. Per esempio, se $E'_A > E'_B$, allora $M = M'_A - M'_B$ e il numero risultante è positivo, ma se $E'_B > E'_A$, allora $M = M'_B - M'_A$

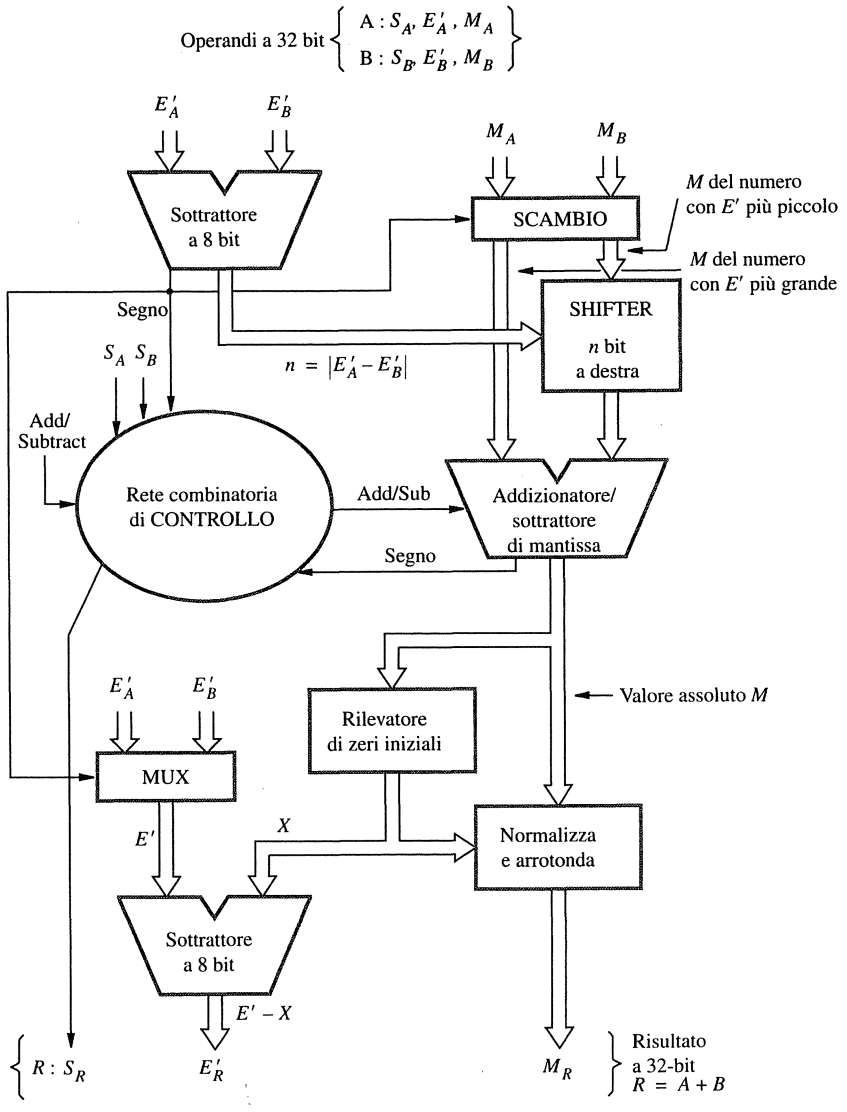


Figura 9.28 Addizionatore algebrico in virgola mobile.

e il risultato è negativo. Questo esempio mostra che è anche richiesto, in ingresso alla rete CONTROLLO, il segno risultante dal confronto degli esponenti. Quando $E'_A = E'_B$ e le mantisse sono sottratte, il segno dell'uscita dell'addizionatore/sottrattore di mantisse determina il segno del risultato. Il lettore dovrebbe ora essere in grado di costruire la tabella di verità completa per la rete CONTROLLO (si veda il Problema propoto 9.26 nell'Eserciziario).

Il passo 4 dell'algoritmo di addizione algebrica consiste nel normalizzare il risultato del passo 3 facendo scorrere M a destra o a sinistra come appropriato. Il numero di zeri a sinistra in M determina il numero X di scorrimenti di bit da applicare a M. Si arrotonda poi il valore normalizzato per generare la mantissa a 24 bit, M_R, del risultato. Il valore X è anche sottratto dall'esponente provvisorio E' per generare il vero esponente del risultato, E'_R. Si noti che non più di uno scorrimento a destra potrebbe essere necessario per normalizzare il risultato. Sarebbe questo il caso se due mantisse della forma $1,xx...$ fossero sommate. Il vettore M avrebbe allora la forma $1x, xx....$

Non è dato alcun dettaglio sui bit di guardia che devono essere elaborati assieme ai valori intermedi della mantissa. Nello standard IEEE, solo alcuni bit sono necessari, come discusso prima, per generare la mantissa a 24 bit normalizzata del risultato.

Si consideri l'hardware effettivamente necessario per realizzare i blocchi in Figura 9.28. I due sottrattori a 8 bit e l'addizionatore/sottrattore di mantissa possono essere realizzati da reti combinatorie, come discusso prima in questo capitolo. Poiché le loro uscite devono essere nella forma di segno e valore assoluto, è necessario modificare alcune discussioni precedenti. Spesso si usa una combinazione di aritmetica in complemento a uno e rappresentazione in segno e valore assoluto. Si permette considerevole flessibilità nella realizzazione di SHIFTER e dell'operazione di normalizzazione dell'uscita. Si possono realizzare tali operazioni con registri a scorrimento, tuttavia le si possono anche costruire come unità logiche combinatorie per ottenere alte prestazioni.

9.8 Conversione da decimale a binario

Nel Capitolo 1 e in questo capitolo, gli esempi che coinvolgono numeri decimali hanno usato valori piccoli. La conversione dalla rappresentazione decimale a quella binaria è stata facile basandosi sui pesi delle posizioni dei bit, ovvero 1, 2, 4, 8, 16, ... Tuttavia è utile disporre di un metodo generale per convertire numeri decimali alla rappresentazione binaria.

Dato il numero binario naturale seguente a $n \geq 2$ bit, in virgola fissa (dove si è posto $p = q = m \geq 1$ e dunque $n = p + q = 2m$):

$$B = b_{m-1} \ldots b_0,b_{-1}b_{-2} \ldots b_{-m}$$

il valore decimale val(B) corrispondente (pure in virgola fissa) è dato dalla formula seguente:

$$\begin{aligned}\text{val}(B) = {} & b_{m-1} \times 2^{m-1} + \ldots + b_0 \times 2^0 + \\ & + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \ldots + b_{-m} \times 2^{-m}\end{aligned}$$

che non necessita di commento. Per convertire in binario il numero decimale in virgola fissa bisogna trattarne separatamente la parte intera e quella frazionaria. Per la conversione della parte intera si usa il noto algoritmo di dimezzamento con resto: si calcola il resto della divisione per due della parte intera, poi si prende il quoziente come nuovo dividendo e si ripete fino a che il quoziente non si sia annullato. Si ottiene così una sequenza di resti, ovvero di bit, che codifica la parte intera in binario naturale, a partire dal bit meno significativo. La conversione della parte frazionaria procede in modo diverso. Si esprime la parte frazionaria come 0, ... e la si raddoppia progressivamente. A ogni passo a sinistra della virgola si ha una cifra 0 o 1, vale a dire un bit, e la successione di tali bit rappresenta la parte frazionaria convertita in binario naturale, a partire dal bit più significativo. Si può procedere indefinitamente, secondo il grado di precisione richiesto.

La Figura 9.29 mostra un esempio di conversione da decimale a binario, in virgola fissa. Il numero decimale di partenza è $927{,}45_{10}$. Si noti come la conversione della parte intera dia

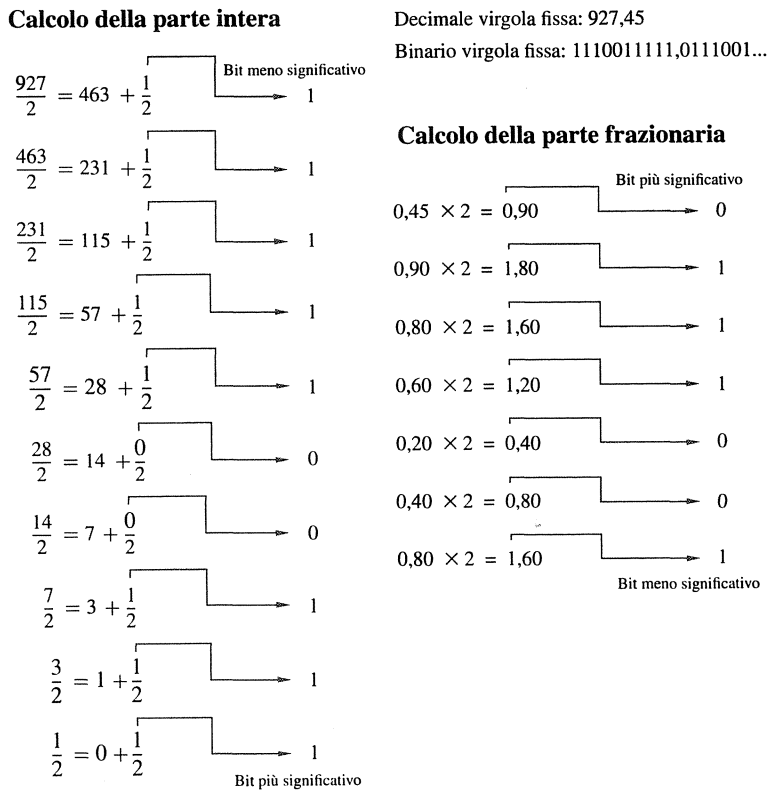


Figura 9.29 Algoritmo di conversione in virgola fissa da decimale a binario.

sempre risultato esatto, mentre per la parte frazionaria si ottiene solo un'approssimazione sempre migliore (salvo casi particolari). Per esempio, il numero frazionario decimale $0{,}45_{10}$ non ammette rappresentazione binaria esatta con un numero finito di bit, come del resto si vede bene esaminando la Figura 9.29. Si può solo procedere fino a ottenere la precisione che si desidera. In generale, se si considera una rappresentazione binaria con parte frazionaria estesa a $k \geq 1$, l'errore di approssimazione non supera 2^{-k} in valore assoluto. Naturalmente in qualche caso l'errore si annulla dopo un numero finito di bit, come succede con il numero decimale $0{,}25_{10}$, che in binario si codifica esattamente come $0{,}01_2$.

Sommario

L'*aritmetica del calcolatore* (*computer arithmetic*) è una parte della scienza dei calcolatori che solleva numerosi problemi di progetto logico interessanti. Il capitolo presenta le nozioni di base necessarie per comprendere il funzionamento delle unità aritmetiche binarie. La tecnica di anticipo del riporto è una delle idee principali nella progettazione di addizionatori ad alte prestazioni.

Nella progettazione di moltiplicatori veloci, la ricodifica bit-pair del moltiplicatore, derivata dall'algoritmo di Booth, riduce il numero di addendi che devono essere sommati per generare il prodotto. L'addizione in parallelo degli addendi che usa alberi di riduzione con salvataggio del riporto riduce in modo sostanziale il tempo necessario a sommare gli addendi. È stato descritto l'importante standard IEEE per la rappresentazione dei numeri in virgola mobile, e sono stati dati algoritmi per effettuare le quattro operazioni standard.

Note bibliografiche

1. Booth A.D., "A Signed Binary Multiplication Technique", *Quarterly Journal of Mechanics and Applied Mathematics* 1951, vol. 2, part 2, pp. 236-240.
2. Wallace C.S., "A Suggestion for a Fast Multiplier", *IEEE Transactions on Electronic Computers* 1964, vol. EC-13, pp. 14-17.
3. Santoro M.R. & Horowitz M.A., "SPIM: A Pipelined 64 × 64-bit Iterative Multiplier", *IEEE Journal of Solid-State Circuits* 1989, vol. 24, n. 2, pp. 487-493.
4. Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-2008, August 2008.

10 Sistemi embedded

In questo capitolo

- Applicazioni embedded
- Microcontrollori per sistemi embedded
- Sensori e attuatori
- Uso del linguaggio C per controllare i dispositivi di I/O
- Questioni di progettazione

Nei capitoli precedenti si sono discussi i concetti usati nei sistemi di calcolatori a uso generale. Adesso si concentrerà la discussione sui sistemi che sono intesi a servire applicazioni specifiche. Un sistema fisico che impiega un calcolatore di controllo per un compito specifico, anziché per un calcolo a uso generale, è detto *sistema embedded*, ovvero sistema incorporato (in un sistema più ampio). Si mostrerà come i concetti generali presentati in precedenza sono applicati in tali sistemi.

Un aspetto importante del software scritto per sistemi embedded è che esso deve interagire da vicino con l'hardware. Il termine *sistema reattivo* è spesso usato per descrivere il fatto che i punti nel tempo in cui si eseguono varie routine sono determinati da eventi esterni al processore, come la chiusura di un interruttore o l'arrivo di nuovi dati su una porta di ingresso. Il progettista del software deve decidere come verrà ottenuta questa interazione. Le tecniche di ingresso/uscita descritte nel Capitolo 3, basate su scansione e interruzioni, sono usate per questo scopo.

Il controllo con microprocessori è adesso comunemente usato in macchine fotografiche, telefoni cellulari, telefoni con display, terminali di vendita tramite bancomat, apparecchi per la cucina, auto e in molti giocattoli. Basso costo e alta affidabilità sono i requisiti essenziali di queste applicazioni. Piccole dimensioni e basso consumo hanno spesso importanza decisiva. Tutto questo può essere ottenuto inserendo su un singolo chip non solo i circuiti del processore, ma anche un po' di memoria, interfacce di ingresso/uscita, circuiti di temporizzazione e altre caratteristiche per rendere facile la realizzazione di un sistema di controllo completo tramite calcolatore usando pochi chip. I chip a microprocessore di questo tipo sono generalmente detti *microcontrollori*. In questo capitolo si esploreranno le caratteristiche principali dei sistemi embedded basati su microcontrollori. Nel Caso di studio C1 si discute l'approccio alla realizzazione di tali sistemi su un solo chip con tecnologia FPGA (si veda inoltre l'Appendice A.11).

10.1 Esempi di sistemi embedded

In questo paragrafo si presentano tre esempi di sistemi embedded per illustrare le capacità di elaborazione e controllo necessarie in una tipica applicazione embedded.

10.1.1 Forno a microonde

Molti apparecchi per la casa usano sistemi di controllo con calcolatore per governare il loro funzionamento. Un tipico esempio è il forno a microonde. Questo apparecchio è basato su un'unità di potenza magnetron che genera le microonde usate per riscaldare il cibo in uno spazio limitato. Quando è acceso, il magnetron genera la sua uscita di potenza massima. Si ottengono livelli di potenza più bassa accendendo e spegnendo il magnetron per intervalli di tempo controllati. Controllando il livello di potenza e il tempo totale di riscaldamento, è possibile realizzare una varietà di opzioni di cottura selezionabili dall'utente.

La specifica di un forno a microonde può includere le seguenti opzioni di cottura:

- selezione manuale del livello di potenza e del tempo di cottura;
- selezione manuale della sequenza di diversi passi di cottura;
- funzionamento automatico, per cui l'utente specifica il tipo di cibo (per esempio, carne, verdure o popcorn) e il peso del cibo; quindi il livello di potenza e il tempo appropriati sono calcolati dal controllore;
- scongelamento automatico del cibo specificando il peso.

Il forno include uno schermo che può mostrare:

- orologio che segna l'orario del giorno;
- temporizzatore che si decrementa durante la cottura;
- messaggi informativi per l'utente.

Si usa un segnale di allarme audio, spesso un bip sonoro, per indicare la fine di un'operazione di cottura. Sono presenti una ventola di aspirazione e una lampadina. Come misura di sicurezza, un sensore di apertura della porta spegne il magnetron se la porta del forno è aperta. Tutte queste funzioni possono essere controllate da un microcontrollore.

Le capacità di ingresso/uscita necessarie per comunicare con l'utente includono:

- tasti per fornire dati in ingresso che comprendono i numeri da 0 a 9 e tasti funzione come Reimposta, Avvio, Stop, Livello potenza, Scongelamento, Cottura automatica, Imposta orologio, Controlla ventola;
- visualizzazione dell'uscita in forma di schermo a cristalli liquidi o LCD (*liquid-crystal display*), simile allo schermo a sette segmenti illustrato in Figura A.37 (si veda anche il Problema risolto 3.3 nell'Eserciziario);
- un piccolo altoparlante che produce il bip sonoro.

I compiti di elaborazione eseguiti da un microcontrollore per controllare un forno a microonde sono abbastanza semplici. Essi includono il mantenimento dell'ora del giorno, la determinazione delle azioni necessarie per le varie opzioni di cottura, la generazione dei segnali di controllo necessari per accendere o spegnere dispositivi come il magnetron e la ventola, la generazione di informazioni da visualizzare. Il programma necessario per realizzare le azioni desiderate è abbastanza piccolo. Esso è immagazzinato in una memoria non volatile a sola lettura, così che non verrà perso quando il forno viene spento. È anche necessario avere una piccola RAM da usare durante l'elaborazione e per tenere i dati inseriti dall'utente. Il requisito più significativo per il microcontrollore è di avere sufficiente capacità di I/O per tutto l'insieme di tasti d'ingresso, LCD e segnali di controllo in uscita. Le porte di I/O parallele forniscono un meccanismo conveniente per gestire i segnali di ingresso e di uscita.

La Figura 10.1 mostra una possibile organizzazione del forno a microonde. Un semplice processore con piccole unità ROM e RAM è sufficiente. Per collegarlo al resto del sistema si usano semplici interfacce di ingresso e uscita. È possibile realizzare la maggior parte di questi circuiti su un piccolo chip microcontrollore.

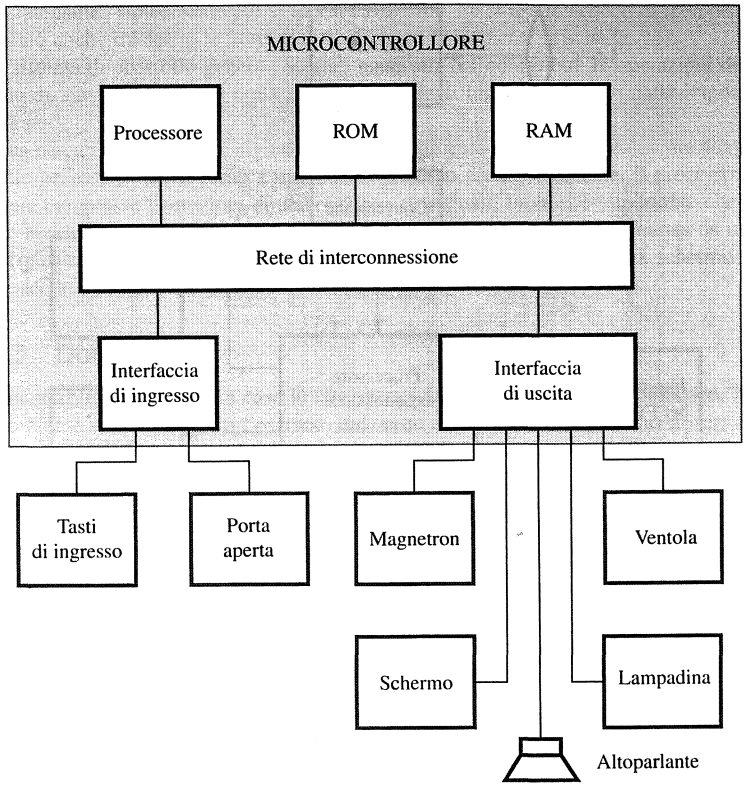


Figura 10.1 Diagramma a blocchi di un forno a microonde.

10.1.2 Macchina fotografica digitale

Le macchine fotografiche digitali forniscono un esempio eccellente di sistema embedded sofisticato in piccole dimensioni. La Figura 10.2 mostra le parti principali di una macchina fotografica.

Le macchine fotografiche tradizionali usano una pellicola per catturare le immagini. In una macchina fotografica digitale, si usa una matrice di sensori ottici per catturare le immagini. Questi sensori convertono la luce in carica elettrica. L'intensità della luce determina l'ammontare di carica generata. Due differenti tipi di sensori sono usati nei prodotti commerciali. Un tipo è detto *Dispositivo ad Accoppiamento di Carica* o DAC (*Charge-Coupled Device*, CCD). È il tipo di dispositivo di rilevamento usato nelle prime macchine fotografiche digitali. Da allora è stato raffinato per fornire immagini di alta qualità. Più recentemente sono stati sviluppati sensori basati sulla tecnologia CMOS. Ciascun sensore genera una carica che corrisponde a un *pixel*, che è un punto di un'immagine grafica. Il numero di pixel determina la qualità delle immagini che possono essere registrate e visualizzate. La carica è una quantità analogica, che è convertita in una rappresentazione digitale usando circuiti di conversione *Analogico-Digitale* (A/D). La conversione A/D produce una rappresentazione digitale del-

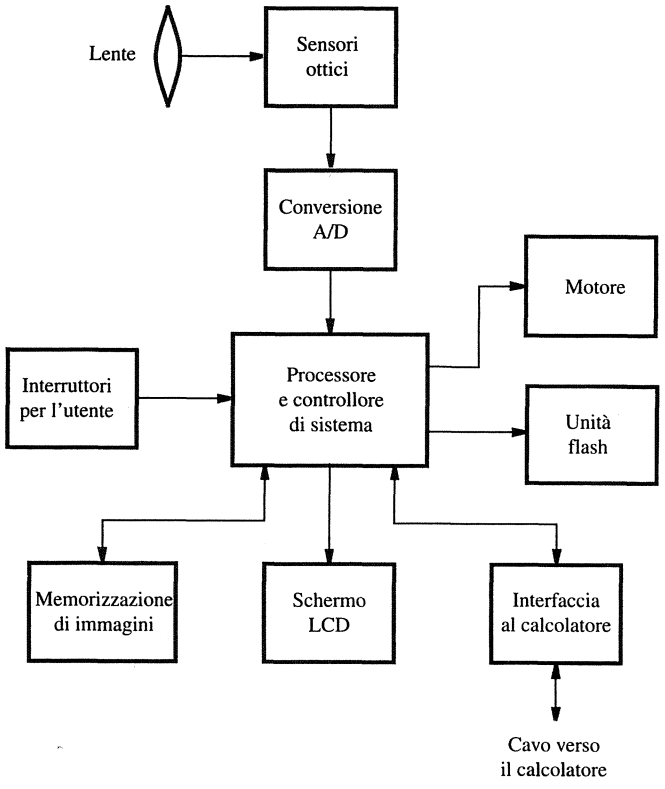


Figura 10.2 Diagramma a blocchi semplificato di una fotocamera digitale.

l'immagine in cui il colore e l'intensità di ciascun pixel sono rappresentati da un certo numero di bit. Il formato digitale dell'immagine può quindi essere trattato come qualsiasi altro dato, manipolabile usando i circuiti standard di un calcolatore.

Il processore e il blocco del sistema di controllo in Figura 10.2 includono una varietà di circuiti di interfaccia necessari a connettere le altri parti del sistema. Il processore governa il funzionamento della macchina fotografica. Esso elabora i dati grezzi dell'immagine ottenuti dai circuiti A/D per generare le immagini rappresentate in formati standard per l'uso nei calcolatori, nelle stampanti e negli schermi. I formati più usati sono TIFF (*Tagged Image File Format*) per immagini non compresse e JPEG (*Joint Photographic Expert Group*) per immagini compresse. Le immagini elaborate sono immagazzinate in un più capace dispositivo di memoria di massa per immagini. Le schede di memoria flash, discusse nel Paragrafo 8.3.5, sono una scelta comune per immagazzinare immagini.

Un'immagine catturata ed elaborata può essere visualizzata in anteprima su uno schermo LCD, incluso nella macchina fotografica. Ciò permette all'utente di decidere se tenere l'immagine. Il numero di immagini che si possono tenere dipende dalla dimensione dell'unità di memoria delle immagini. Dipende anche dalla scelta della qualità delle immagini, ovvero dal numero di pixel per immagine e dal grado di compressione (per il formato JPEG).

Un'interfaccia standard fornisce un meccanismo per trasferire le immagini a un calcolatore o una stampante. Tipicamente ciò si fa via cavo USB. Se si usano schede di memoria flash, si possono anche trasferire le immagini trasferendo fisicamente la scheda.

Il sistema di controllo genera i segnali necessari per controllare le operazioni del meccanismo di messa a fuoco e l'unità flash. Alcuni degli ingressi vengono dalle impostazioni dell'utente.

Una macchina fotografica digitale richiede un processore considerevolmente più potente di quello necessario per il forno a microonde discusso precedentemente. Il processore deve effettuare complesse operazioni di elaborazione di segnali. Inoltre, è importante che il processore non consumi molta potenza poiché la macchina fotografica è alimentata da una batteria. Tipicamente, il processore consuma meno potenza rispetto alle unità schermo e flash della macchina fotografica.

10.1.3 Domotica

I microcontrollori sono usati a casa in una vasta schiera di applicazioni embedded. Nel Paragrafo 10.1.1 si è considerato l'esempio del forno a microonde. Si possono trovare esempi simili in altri dispositivi come lavatrici, asciugatrici, lavastoviglie, piani di cottura, caldaie e condizionatori. Un altro esempio notevole è il telefono con display, in cui un processore embedded abilita una varietà di caratteristiche utili. In aggiunta alle caratteristiche standard del telefono, un telefono con microcontrollore embedded può essere usato per fornire accesso remoto ad altri dispositivi della casa.

Usando il telefono si possono eseguire a distanza funzioni come:

- comunicare con un sistema di sicurezza per la casa controllato da un calcolatore;
- impostare la temperatura desiderata che viene mantenuta da una caldaia o da un condizionatore;
- impostare il tempo di accensione, il tempo di cottura, e la temperatura per il cibo che è stato inserito nel forno in un momento precedente;
- leggere il contatore di elettricità, gas e acqua, risparmiando alle aziende di servizi la necessità di mandare un addetto nelle case per leggere i contatori.

Tutto questo è facilmente realizzabile se ciascuno di questi dispositivi è controllato da un microcontrollore. È necessario solamente fornire un collegamento, con fili o senza, fra il dispositivo microcontrollore e il microprocessore in un telefono. L'uso di segnali da una locazione remota per osservare e controllare lo stato dei dispositivi è spesso detto *telemetria*.

10.2 Chip microcontrollori per applicazioni embedded

Un chip microcontrollore dovrebbe essere sufficientemente versatile da servire per un'ampia varietà di applicazioni. La Figura 10.3 mostra il diagramma a blocchi di un chip tipico. La parte principale è un *nucleo processore* (*processor core*), che può essere una versione di base di un microprocessore disponibile in commercio. La prudenza consiglia di scegliere un'architettura a microprocessore che abbia avuto diffusione in pratica, poiché per tali processori esistono numerosi strumenti CAD, buoni esempi e una gran quantità di esperienza e conoscenza che facilita la progettazione di nuovi prodotti.

È utile che il chip includa memoria sufficiente a soddisfare i requisiti di memoria tipici di piccole applicazioni. Una parte di questa memoria deve essere RAM per tenere i dati che cambiano durante i calcoli. Una parte dovrebbe essere del tipo a sola lettura per tenere il software, poiché in genere un sistema embedded non dispone di un disco magnetico. Per contenere i costi

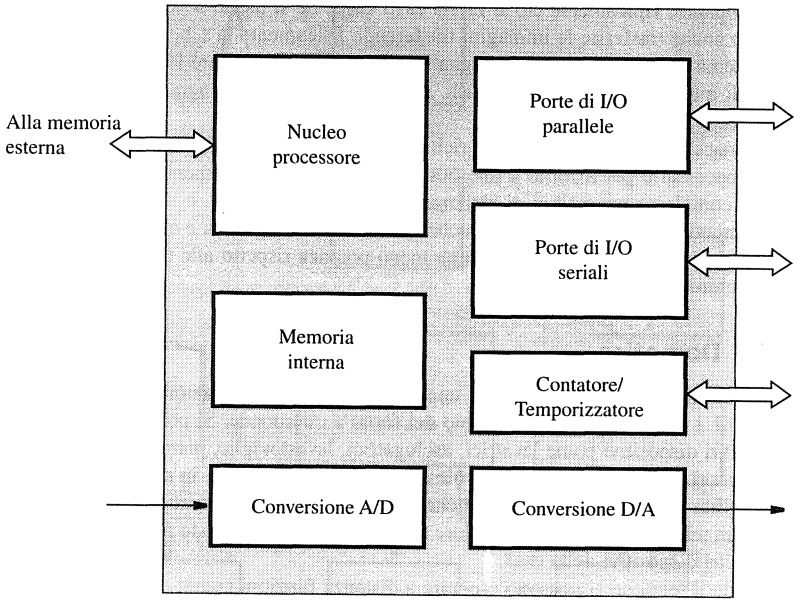


Figura 10.3 Diagramma a blocchi di un microcontrollore.

in applicazioni a bassa diffusione, è necessario avere un tipo di ROM programmabile sul campo. Per la realizzazione di queste unità si scelgono comunemente EEPROM e memorie flash.

Si forniscono diverse porte di I/O sia per interfacce parallele sia seriali, che permettono una facile realizzazione delle connessioni I/O standard. In molte applicazioni è necessario generare segnali di controllo a intervalli di tempo programmati. Ciò si compie facilmente se si include un circuito per il temporizzatore nel chip del microcontrollore. Poiché il temporizzatore è un circuito che conta impulsi di clock, esso può anche essere usato per contare eventi, per esempio per contare il numero di impulsi generati da un braccio meccanico in movimento o da un albero rotante.

Un sistema embedded può includere alcuni dispositivi analogici. Per gestire tali dispositivi è necessario essere in grado di convertire segnali analogici nelle rappresentazioni digitali e viceversa. Questo si ottiene facilmente se il controllore embedded include circuiti di conversione A/D e D/A.

In commercio sono disponibili molti chip di processori embedded. Alcuni dei più noti esempi sono: le famiglie 68HC11 Freescale e 68K/Coldfire, le famiglie 8051 Intel e MCS-96, tutte con processori in stile CISC, e i microcontrollori ARM, che hanno un processore in stile RISC. La natura del nucleo processore non è importante per la discussione in questo capitolo. Si evidenzieranno gli aspetti di sistema delle applicazioni embedded per illustrare come i concetti presentati nei precedenti capitoli si accordino nella progettazione di un calcolatore embedded completo.

10.3 Un microcontrollore semplice

La struttura di ingresso/uscita di un microcontrollore deve essere sufficientemente flessibile per gestire le necessità di diverse applicazioni e per fare buon uso dei piedini disponibili

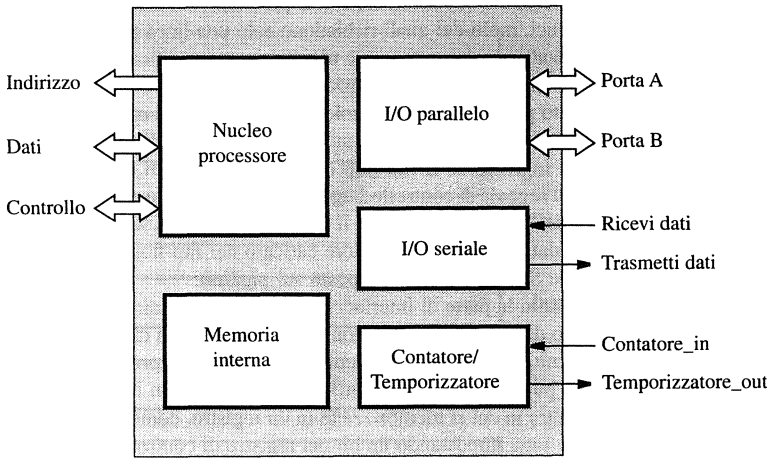


Figura 10.4 Un esempio di microcontrollore.

sul chip. Per esempio, una porta parallela può essere configurabile sia come di ingresso sia come di uscita.

In questa parte si discute una possibile organizzazione di un semplice microcontrollore per illustrare alcune caratteristiche tipiche. La Figura 10.4 ne mostra il diagramma a blocchi. Vi è un processore e alcuni chip di memoria. Poiché i chip di memoria possono essere insufficienti al supporto di tutte le applicazioni potenziali, le connessioni del bus del processore sono pure disponibili sui piedini del chip così da potere aggiungere memoria esterna.

Vi sono due interfacce parallele a 8 bit, chiamate porta A e porta B, e una interfaccia seriale. Il microcontrollore contiene pure un circuito contatore/temporizzatore a 32 bit, che può essere usato per generare interruzioni interne a intervalli di tempo programmati, per fare da temporizzatore di sistema, per contare gli impulsi su una linea di ingresso, per generare segnali di onda quadra in uscita ecc.

10.3.1 Interfacce parallele di I/O

Le applicazioni per sistemi embedded richiedono una notevole flessibilità nelle interfacce di ingresso/uscita. Si può apprezzare la natura dei dispositivi in gioco e come possono essere connessi al microcontrollore considerando alcuni componenti del forno a microonde mostrato in Figura 10.1. Occorre un sensore per generare un segnale con il valore 1 quando la porta è aperta. Questo segnale è mandato al microcontrollore su uno dei piedini di un'interfaccia di ingresso. Lo stesso avviene per i tasti del pannello frontale del forno. Ognuno di questi semplici dispositivi produce un bit di informazione.

I dispositivi di uscita sono controllati in modo simile. Il magnetron è controllato da una sola linea di uscita che lo accende e lo spegne. Lo stesso avviene per la ventola e la lampadina. L'altoparlante può essere pure connesso da una singola linea di uscita sulla quale il processore manda un segnale a onda quadra di frequenza appropriata. Uno schermo LCD, d'altra parte, richiede che vengano inviati diversi bit di dati in parallelo.

Uno degli obiettivi della progettazione di interfacce di ingresso/uscita per un microcontrollore è ridurre il più possibile la necessità di circuiti esterni. È probabile che il microcontrollore sia collegato a dispositivi semplici, molti dei quali richiedono solo una linea per il segnale di ingresso o uscita. Nella maggior parte dei casi, non è necessaria una codifica o decodifica.

Ciascuna porta parallela in Figura 10.4 ha associato un registro da 8 bit di direzione dei dati, che può essere utilizzato per configurare singole linee di dati come ingresso o uscita. La Figura 10.5 illustra il controllo bidirezionale per un bit nella porta A. Il piedino della porta PA_i viene trattato come un ingresso se il flip-flop di direzione dei dati contiene uno 0. In questo caso, l'attivazione del segnale di controllo Leggi_Porta pone il valore logico del piedino di porta sulla linea dati D_i del bus del processore. Il piedino della porta serve come uscita se il flip-flop di direzione dei dati è posto a 1. Il valore caricato nel flip-flop dei dati di uscita, sotto il controllo del segnale Scrivi_Porta, viene posto sul piedino.

La Figura 10.5 mostra solo la parte di interfaccia che controlla la direzione del trasferimento dei dati. Nel percorso di ingresso dei dati non vi è alcun flip-flop che cattura e detiene il valore del segnale dati fornito da un dispositivo collegato al piedino corrispondente. Un'interfaccia parallela versatile può offrire due opzioni: una in cui i dati in ingresso si leggono direttamente dai piedini, l'altra in cui si memorizzano in un registro, come nell'interfaccia in Figura 7.11. La scelta viene fatta impostando un bit nel registro di controllo dell'interfaccia.

La Figura 10.6 mostra tutti i registri dell'interfaccia parallela, così come gli indirizzi assegnati a essi. Si sono scelti arbitrariamente indirizzi nella fascia alta di un intervallo di indirizzi a 32-bit.

Il registro di stato, PSTAT, contiene le variabili binarie di stato. Il bit di stato PASIN è posto a 1 quando ci sono nuovi dati sulla porta A. Viene azzerato quando il processore accetta i dati leggendo il registro PAIN. Il bit PASOUT è posto a 1 quando i dati nel registro PAOUT sono accettati dal dispositivo collegato, per indicare che adesso il processore può caricare

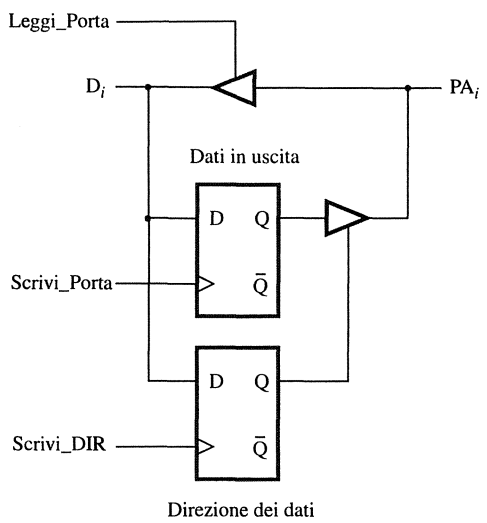


Figura 10.5 Accesso a un bit nella porta A di Figura 10.4.

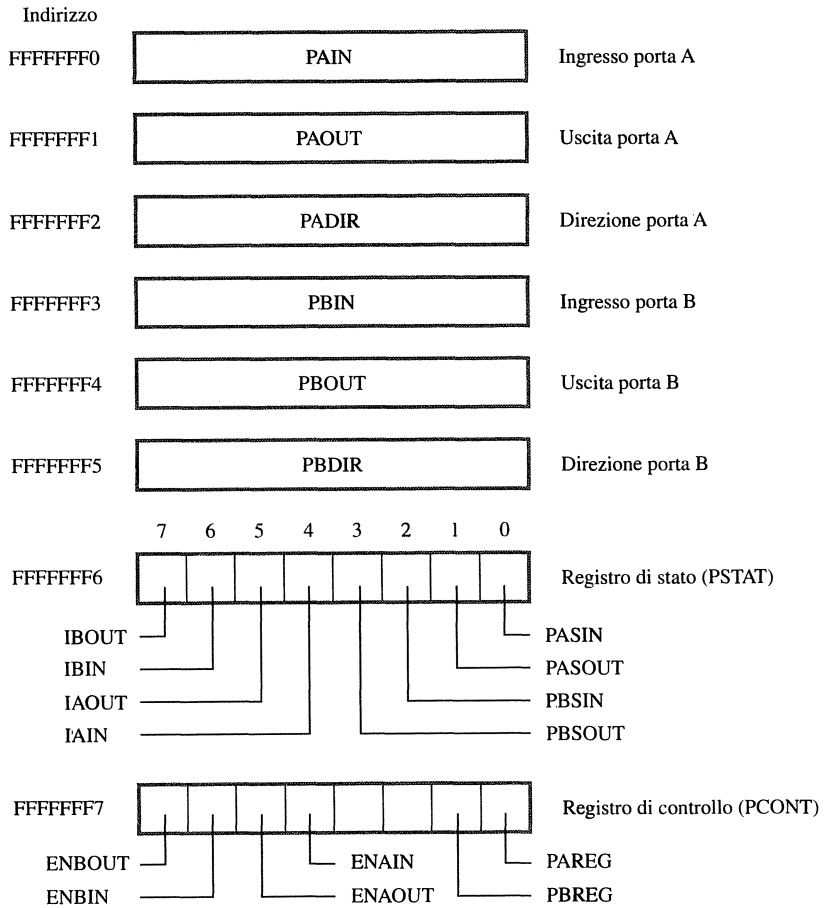


Figura 10.6 Registri dell'interfaccia parallela.

nuovi dati in PAOUT. L'interfaccia usa una linea separata di controllo (descritto di seguito) per segnalare la disponibilità di nuovi dati al dispositivo collegato. Il bit PASOUT viene azzerato quando il processore scrive dati in PAOUT. I bit di stato PBSIN e PBSOUT hanno la stessa funzione per la porta B.

Il registro di stato contiene anche quattro variabili binarie per le interruzioni. Una variabile di interruzione, come IAIN, è posta a 1 quando l'interruzione è attivata e avviene la corrispondente azione di I/O. I bit di abilitazione delle interruzioni sono contenuti nel registro di controllo PCONT. Un bit di abilitazione è posto a 1 per abilitare l'interruzione corrispondente. Per esempio, se ENAIN = 1 e PASIN = 1, allora il bit IAIN è posto a 1 e si genera una richiesta di interruzione. Quindi,

$$IAIN = ENAIN \cdot PASIN$$

Si utilizza un singolo segnale di richiesta di interruzione per tutte le porte nell'interfaccia. In risposta a una richiesta di interruzione, il processore deve esaminare le variabili di interruzione per determinare l'origine della richiesta.

Si utilizzano le informazioni contenute nei registri di stato e di controllo per regolare i trasferimenti di dati da e verso i dispositivi collegati alle porte A e B. La porta A ha due linee di controllo, CAIN e CAOUT, che possono essere usate per realizzare un meccanismo di segnalazione automatica fra l'interfaccia e il dispositivo collegato, per dispositivi che dispongono di questa funzionalità. Per un trasferimento in ingresso, il dispositivo pone i nuovi dati sui piedini della porta e segnala questa azione attivando la linea CAIN per un ciclo di clock. Quando il circuito di interfaccia vede CAIN = 1, il bit di stato PASIN è posto a 1. Successivamente, questo bit viene posto a 0 quando il processore legge i dati di ingresso. Inoltre questa azione fa sì che l'interfaccia invii un impulso sulla linea CAOUT per informare il dispositivo che può inviare nuovi dati all'interfaccia. Per un trasferimento in uscita, il processore scrive i dati nel registro PAOUT. L'interfaccia risponde ponendo il bit PASOUT a 0 e inviando un impulso sulla linea CAOUT per informare il dispositivo che sono disponibili nuovi dati. Quando il dispositivo accetta i dati, invia un impulso sulla linea CAIN, che a sua volta pone PASOUT a 1. Questo meccanismo di segnalazione funziona quando tutti i piedini dati di una porta hanno la stessa direzione, ovvero quando la porta serve come porta di ingresso o come porta di uscita. Se alcuni piedini sono selezionati come ingressi e altri come uscite, il meccanismo automatico non è utilizzato e né le linee di controllo né i registri di stato e di controllo contengono informazione significativa. In questo caso, gli ingressi sono letti direttamente dai piedini.

I bit PAREG e PBREG del registro di controllo sono utilizzati per selezionare la modalità di funzionamento degli ingressi rispettivamente per le porte A e B. Se è posto a 1, un registro memorizza i dati di ingresso, altrimenti si usa un percorso diretto dai piedini come indicato in Figura 10.5. Come esempio d'uso del percorso diretto, si consideri il funzionamento del forno a microonde illustrato nella Figura 10.1. Il microcontrollore accende il magnetron per avviare la cottura, ma può farlo solo se la porta del forno è chiusa. Un semplice sensore binario indica se la porta è aperta, con un segnale che può essere letto come un bit di dati. Il sensore è collegato a un piedino dell'interfaccia del microcontrollore, consentendo al microcontrollore di determinare lo stato della porta leggendo direttamente il valore logico di questo ingresso.

10.3.2 Interfaccia di I/O seriale

L'interfaccia seriale fornisce la funzionalità di trasferimento di dati nota come UART (*Universal Asynchronous Receiver/Transmitter*), in base allo schema descritto nel Paragrafo 7.4.2. Si usa un doppio buffer sia nel percorso di trasmissione sia di ricezione, come mostrato in Figura 10.7. Tali buffer sono necessari per gestire correttamente raffiche di dati nei trasferimenti di I/O.

La Figura 10.8 mostra i registri indirizzabili dell'interfaccia seriale. I dati di ingresso sono letti dal buffer Ricezione a 8 bit, mentre i dati di uscita sono caricati nel buffer Trasmissione a 8 bit. Il registro di stato, SSTAT, fornisce informazioni sullo stato corrente delle unità di ricezione e trasmissione. Il bit $SSTAT_0$ è posto a 1 quando nel buffer di ricezione ci sono dati validi; viene posto automaticamente a 0 al momento di un accesso in lettura del buffer di ricezione. Il bit $SSTAT_1$ è posto a 1 quando il buffer di trasmissione è vuoto e può essere caricato con nuovi dati. Questi bit servono allo stesso scopo delle variabili di stato KIN e DOUT discusse nel Paragrafo 3.1. Il bit $SSTAT_2$ è posto a 1 se si verifica un errore durante il processo di ricezione. Per esempio, si verifica un errore se il carattere nel buffer di ricezione è sovrascritto da un carattere ricevuto successivamente prima che il primo carattere sia letto dal processore. Il registro di stato contiene anche le variabili di interruzione. Il bit $SSTAT_4$ è posto a 1 quando il buffer di ricezione è pieno e l'interruzione è abilitata. Analogamente,

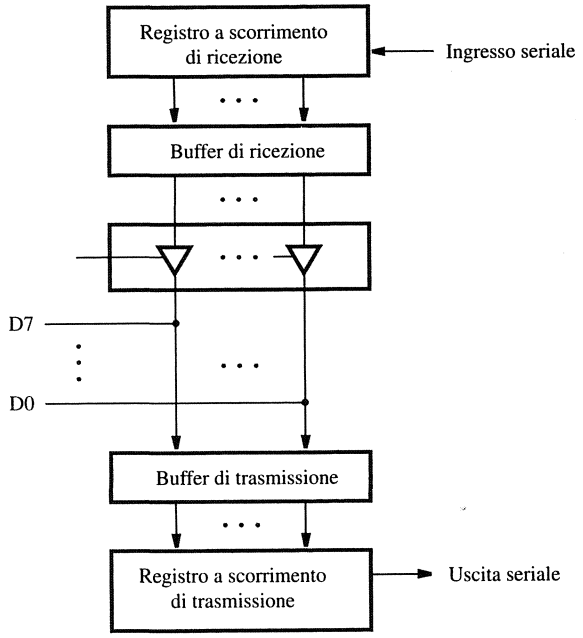


Figura 10.7 Struttura di ricezione e trasmissione dell'interfaccia seriale.

$SSTAT_5$ è posto a 1 quando il buffer di trasmissione diventa vuoto e l'interruzione di trasmissione è abilitata. L'interfaccia seriale genera un'interruzione se $SSTAT_4$ oppure $SSTAT_5$ è 1. Inoltre, si genera un'interruzione se $SSTAT_6 = 1$, che si verifica se $SSTAT_2 = 1$ e se l'interruzione per la condizione di errore è abilitata.

Il registro di controllo, SCONT, contiene i bit di abilitazione delle interruzioni. Ponendo i bit $SCONT_{6-4}$ a 1 o 0 rispettivamente si abilitano o disabilitano le interruzioni corrispondenti. Questo registro indica anche come è generato il clock di trasmissione. Se $SCONT_0 = 0$, allora il clock di trasmissione coincide con quello del sistema (processore). Se $SCONT_0 = 1$, allora un clock di trasmissione a frequenza più bassa viene ottenuto utilizzando un circuito di divisione del clock (si veda la Figura 10.8).

L'ultimo registro dell'interfaccia seriale è il registro divisore del clock, DIV. Questo registro a 32-bit è associato a un circuito contatore che divide il segnale di clock del sistema per generare il clock di trasmissione seriale. Il contatore genera un segnale di clock la cui frequenza è uguale alla frequenza del clock di sistema divisa per il contenuto di questo registro. Il valore caricato in questo registro viene trasferito nel contatore, che quindi conta alla rovescia utilizzando il clock di sistema. Quando il conteggio raggiunge lo zero, il contatore viene ricaricato usando il valore del registro DIV.

10.3.3 Contatore/Temporizzatore

Un circuito contatore verso il basso (conto alla rovescia) da 32 bit è usabile sia come contatore sia come temporizzatore. Il funzionamento di base del circuito consiste nel caricare un valore

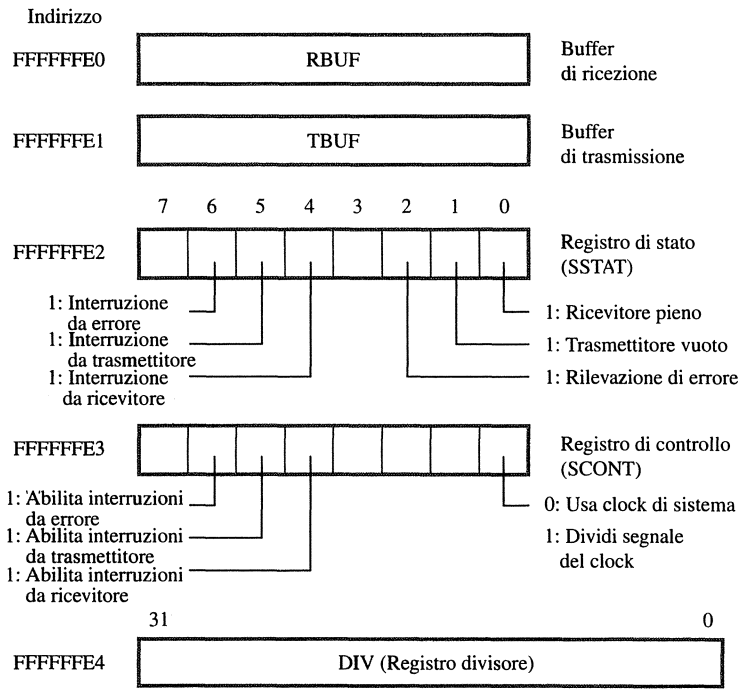


Figura 10.8 Registri dell'interfaccia seriale.

iniziale nel contatore e quindi decrementarne il contenuto utilizzando il clock interno del sistema o un segnale di clock esterno. Si può programmare il circuito per generare un'interruzione quando il contenuto del contatore raggiunge lo zero. La Figura 10.9 mostra i registri associati al circuito contatore/temporizzatore.

Si può caricare il registro contatore/temporizzatore, CNTM, con un valore iniziale, che è poi trasferito al circuito contatore. Il contenuto attuale del contatore può essere letto accedendo all'indirizzo di memoria FFFFFFD4. Si usa il registro di controllo, CTCON, per specificare la modalità operativa del circuito contatore/temporizzatore. Esso fornisce un meccanismo per avviare e arrestare il conteggio e per abilitare le interruzioni quando il contenuto del contatore è decrementato a 0. Il registro di stato, CTSTAT, riflette lo stato del circuito.

Modalità contatore

La modalità contatore è selezionata ponendo il bit $CTCON_7$ a 0. Si carica il valore iniziale nel contatore scrivendolo nel registro CNTM. Il conteggio inizia quando il bit $CTCON_0$ è posto a 1 da un'istruzione di programma. Quando inizia il conteggio, il bit $CTCON_0$ viene automaticamente posto a 0. Il contatore è decrementato dagli impulsi sulla linea Contatore_in in Figura 10.4. Arrivati a zero, il circuito contatore pone il bit di stato $CTSTAT_0$ a 1, e genera un'interruzione se il corrispondente bit di abilitazione delle interruzioni è stato posto a 1. Il successivo impulso di clock fa sì che il contatore carichi nuovamente il valore di partenza,

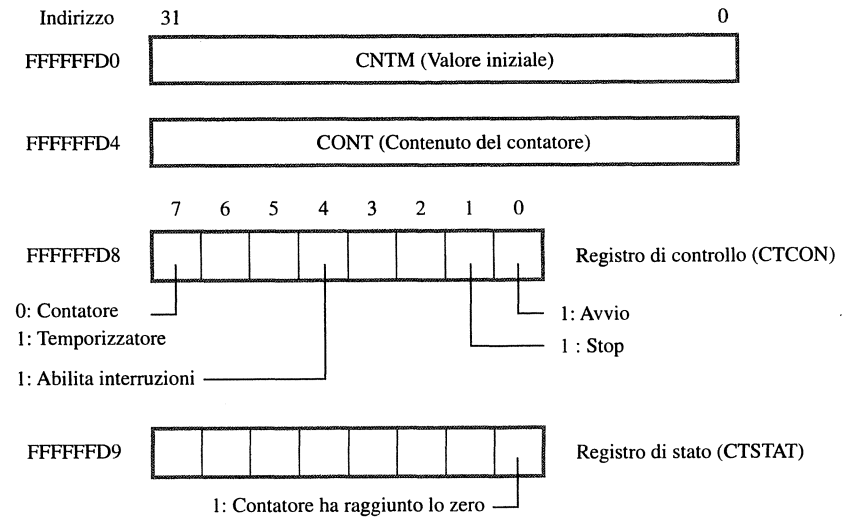


Figura 10.9 Registri del contatore/temporizzatore.

contenuto nel registro CNTM, e il conteggio continua. Il conteggio viene bloccato ponendo il bit $CTCON_1$ a 1.

Modalità temporizzatore

La modalità temporizzatore si seleziona ponendo a 1 il bit $CTCON_7$. Questa modalità può essere utilizzata per generare interruzioni periodiche. È inoltre adatta a generare un segnale a onda quadra sulla linea in uscita Temporizzatore_uscita in Figura 10.4. Il processo inizia come spiegato sopra per la modalità contatore. Quando il contatore si decrementa, il valore sulla linea di uscita è tenuto costante. Arrivato a zero, il contatore viene ricaricato automaticamente con il valore iniziale, e il segnale di uscita sulla linea è invertito. Pertanto, il periodo del segnale di uscita è il doppio del valore iniziale del contatore moltiplicato per il periodo dell'impulso del clock di controllo. Nella modalità temporizzatore, il contatore è decrementato dal clock di sistema.

10.3.4 Meccanismo di controllo delle interruzioni

Il processore del microcontrollore d'esempio ha due ingressi di richiesta d'interruzione, IRQ e XRQ. L'ingresso IRQ è utilizzato per le interruzioni generate dalle interfacce di I/O interne al microcontrollore. L'ingresso XRQ è utilizzato per le interruzioni generate da dispositivi esterni. Se l'ingresso IRQ è asserito e le interruzioni sono abilitate, il processore esegue una routine di servizio delle interruzioni che usa il metodo della scansione per determinare la fonte della richiesta di interruzione. Questo viene fatto esaminando i bit di stato nei registri PSTAT, SSTAT e CTSTAT. Le interruzioni XRQ hanno priorità maggiore rispetto alle interruzioni IRQ.

Il registro di stato del processore, PSR, ha due bit di abilitazione delle interruzioni. Le interruzioni IRQ sono abilitate se $PSR_6 = 1$ e le interruzioni XRQ sono abilitate se $PSR_7 = 1$.

Quando il processore accetta un'interruzione, disabilita ulteriori interruzioni allo stesso livello di priorità azzerando il corrispondente bit PSR prima di eseguire la routine di servizio dell'interruzione. Si usa uno schema di interruzioni vettorizzate, con i vettori per le interruzioni IRQ e XRQ rispettivamente nelle locazioni di memoria 0x20 e 0x24. Ogni vettore contiene l'indirizzo della prima istruzione della routine di servizio dell'interruzione corrispondente. Questo indirizzo viene automaticamente caricato nel contatore di programma, PC.

Il processore ha un registro di collegamento, LR, usato per il collegamento di sottoprogrammi, come spiegato nel Paragrafo 2.7. Un'istruzione di chiamata a sottoprogramma fa sì che il contenuto aggiornato del contatore di programma, che è l'indirizzo di rientro richiesto, sia salvato in LR prima del salto alla prima istruzione del sottoprogramma. Un altro registro, IRA, salva l'indirizzo di rientro quando si accetta una richiesta di interruzione. In questo caso, oltre a salvare l'indirizzo di rientro in IRA, si memorizza il contenuto del registro di stato del processore, PSR, nel registro del processore IPSR.

Il rientro da un sottoprogramma avviene tramite un'istruzione ReturnS, che trasferisce il contenuto di LR in PC. Il rientro da un'interruzione avviene tramite un'istruzione ReturnI, che trasferisce il contenuto di IRA e IPSR rispettivamente in PC e PSR. Poiché c'è solo un registro IRA e un registro IPSR, le interruzioni annidate possono essere realizzate salvando in pila il contenuto di questi registri mediante istruzioni apposite nella routine di servizio dell'interruzione. Si noti che se la routine di servizio dell'interruzione chiama un sottoprogramma, allora deve salvare il contenuto di LR, poiché si potrebbe verificare un'interruzione mentre il processore sta eseguendo un altro sottoprogramma.

10.3.5 Esempi di programmazione

Dopo aver introdotto l'hardware del microcontrollore, si considerino adesso alcune questioni software che sorgono quando si utilizzano le interfacce del microcontrollore per collegare dispositivi di I/O. I programmi possono essere scritti in linguaggio assemblativo o in un linguaggio ad alto livello. Quest'ultima scelta è preferibile nella maggior parte delle applicazioni poiché il codice desiderato è più facile da produrre e mantenere, e il tempo di sviluppo è più breve. Negli esempi di questo capitolo si userà il linguaggio di programmazione C.

Gli esempi forniti nei Problemi risolti 10.1 e 10.2 dell'Eserciziario sono rudimentali, intesi a illustrare i possibili approcci. Nel paragrafo seguente si dà un esempio più elaborato di un'applicazione completa.

10.4 Cronometro di riflessi, un esempio completo

Dopo avere introdotto le caratteristiche fondamentali del microcontrollore, si mostra adesso come può essere utilizzato in un semplice sistema embedded che realizza un compito facile da comprendere, esemplificativo del termine *sistema reattivo*. Si desidera progettare un "cronometro di riflessi", utile a misurare la velocità di risposta di una persona a uno stimolo visivo.

L'idea è quella che il microcontrollore accenda una fonte di luce e poi misuri il tempo di reazione che il soggetto impiega per spegnerla premendo un pulsante. I dettagli del sistema e del suo funzionamento sono elencati di seguito.

- Vi sono due pulsanti manuali, Avvio e Stop, un diodo a emissione di luce o LED (*Light-Emitting Diode*) e uno schermo con tre cifre a sette segmenti.
- Il sistema viene attivato premendo il pulsante Avvio.
- Dopo l'attivazione, lo schermo a sette segmenti è posto a 000 e il LED è spento.

- Dopo un ritardo di tre secondi, si accende il LED e parte il cronometro.
- Quando viene premuto il pulsante Stop, il cronometro si ferma, il LED si spegne e il tempo trascorso viene visualizzato sullo schermo a sette segmenti.
- Si calcola e si visualizza il tempo trascorso in centesimi di secondo. Poiché il display ha solo tre cifre, si presume che il tempo trascorso sarà inferiore a dieci secondi.

La Figura 10.10 mostra l'hardware in grado di realizzare il cronometro di riflessi desiderato. Il microcontrollore offre tutti i componenti hardware tranne i pulsanti di ingresso e gli schermi di uscita. In contrasto con gli esempi proposti nei Problemi risolti 10.1 e 10.2 dell'Eserciziario, si suppone che un circuito di decodifica da BCD a sette segmenti sia associato a ciascuno schermo a sette segmenti, cosicché il microcontrollore debba inviare solo un codice a quattro bit BCD per ogni cifra da visualizzare. Il microcontrollore dell'esempio non dispone di un numero di porte parallele sufficiente per l'invio di segnali decodificati ai tre schermi a sette segmenti.

Si useranno le due porte parallele, A e B, per tutte le funzioni di ingresso/uscita. Le due cifre BCD più significative del tempo da visualizzare sono collegate alla porta A, mentre la cifra meno significativa è collegata ai quattro bit superiori della porta B. I pulsanti e il LED sono collegati ai tre bit inferiori della porta B. Per misurare il tempo trascorso si utilizza il circuito contatore/temporizzatore, pilotato dal clock di sistema che si assume abbia una frequenza di 100 MHz.

Un programma che realizza il compito desiderato può essere basato sul seguente approccio.

- L'intenzione dell'utente di iniziare la prova è monitorata per mezzo di un ciclo di attesa che verifica lo stato del pulsante Avvio.
- Dopo avere osservato l'avvenuta pressione del pulsante Avvio, cioè dopo avere rilevato che $PB_1 = 0$, e dopo un ulteriore ritardo di tre secondi, il LED si accende.
- Il contatore è posto al valore iniziale 0xFFFFFFFF e si avvia il processo di decremento del contatore a ogni impulso di clock.
- Un ciclo di attesa verifica lo stato del pulsante Stop per rilevare quando l'utente reagisce premendolo.
- Quando viene premuto il pulsante Stop, si spegne il LED, si ferma il contatore e si calcola il tempo trascorso.
- Il ritardo misurato viene convertito in un numero BCD e inviato agli schermi a sette segmenti.

Gli indirizzi di vari registri di I/O del microcontrollore sono come indicato nelle Figure da 10.6 a 10.9. Il programma deve configurare le porte A e B come richiesto dalle connessioni mostrate in Figura 10.10. Tutti i bit della porta A e i quattro bit di ordine superiore della porta B sono configurati come uscite. Dei tre bit inferiori della porta B, PB_0 e PB_1 sono usati come ingressi, mentre PB_2 è un'uscita. Non vi è alcuna necessità di usare i segnali di controllo disponibili nelle due porte, poiché il dispositivo di ingresso è costituito da pulsanti che pilotano direttamente le linee di porta, e il dispositivo di uscita è uno schermo che segue direttamente i cambiamenti dei segnali sui piedini di porta che lo guidano.

Si mostrerà come l'applicazione richiesta possa essere realizzata usando il linguaggio C. Il programma esegue le seguenti attività. Dopo che il tasto Avvio viene premuto, si realizza tramite il temporizzatore un ritardo di tre secondi. Poiché il circuito contatore/temporizzatore ha un clock a 100 MHz, il contatore viene inizializzato al valore esadecimale 11E1A300, che corrisponde al valore decimale 300.000.000. Il processo di conteggio alla rovescia è avviato ponendo il bit $CTCONT_0$ a 1. Quando il conteggio arriva a zero, si accende il LED per iniziare la misura del tempo di reazione e si pone il contatore a 0xFFFFFFFF. Quando

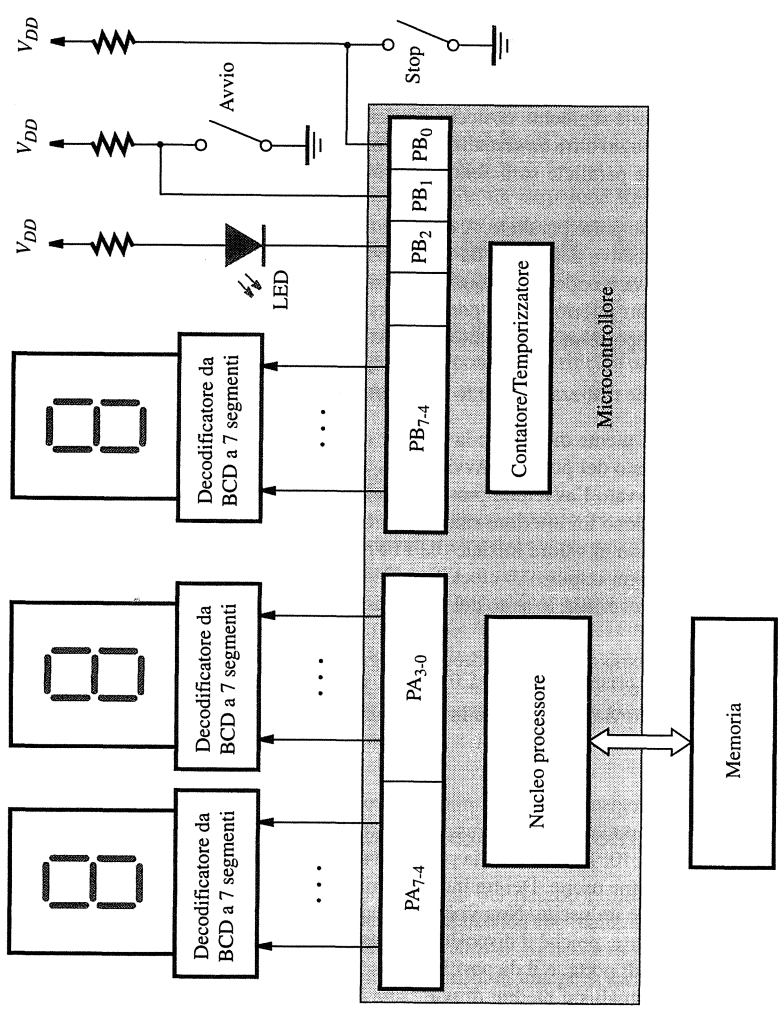


Figura 10.10 Circuito del cronometro di riflessi.

si rileva che il pulsante Stop è stato premuto, il processo di conteggio viene fermato ponendo CTCONT1 = 1. Il conteggio totale è calcolato come:

$$\text{Conteggio totale} = \text{0xFFFFFFFF} - \text{Conteggio presente}$$

Poiché questo è il numero totale di cicli di clock, il tempo effettivo in centesimi di secondo è:

$$\text{Tempo effettivo} = (\text{Conteggio totale})/1.000.000$$

Questo intero binario può essere convertito in un numero decimale dapprima dividendolo per 100 per generare la cifra più significativa. Il resto viene quindi diviso per 10 per generare la cifra successiva. Il resto finale è la cifra meno significativa.

La Figura 10.11 mostra un possibile programma. Dopo avere configurato le porte A e B come richiesto e avere spento lo schermo e il LED, il programma osserva continuamente il valore sul piedino PB_1. Dopo che il pulsante Avvio sia stato premuto e che PB_1 sia diventato uguale a 0, si introduce un ritardo di tre secondi. Quindi, il LED viene acceso e parte il processo di cronometraggio. Si reitera un altro ciclo di scansione, in attesa della pressione del pulsante Stop. Quando questo pulsante viene premuto, si spegne il LED, si arresta il contatore e se ne legge il contenuto. Si eseguono quindi il calcolo del tempo trascorso e la conversione in un numero decimale come spiegato sopra.

Le tre cifre BCD risultanti vengono scritte sui registri dati della porta secondo la disposizione mostrata in Figura 10.10.

10.5 Sensori e attuatori

Un calcolatore embedded interagisce strettamente con il suo ambiente. Finora sono stati usati interruttori e dispositivi di visualizzazione semplici per illustrare tali interazioni. Vi è una varietà di altri dispositivi adoperati in applicazioni pratiche. Per controllare un sistema meccanico, è necessario impiegare dispositivi che forniscano un'interfaccia fra il dominio della meccanica e il dominio dell'elettronica digitale in uso nel calcolatore. Questi dispositivi consentono al calcolatore di rilevare, o monitorare, lo stato del sistema controllato. Inoltre essi fanno sì che le azioni ordinate dal calcolatore siano eseguite. Tali dispositivi sono solitamente detti *sensori* e *attuatori*. Insieme essi sono detti *trasduttori*. In questo paragrafo, si presentano alcuni esempi di trasduttori che illustrano alcuni dei loro principi di base.

10.5.1 Sensori

A seconda della natura del sistema controllato, ci possono essere molti parametri che un calcolatore embedded deve monitorare. Si consideri il sistema di regolazione automatica della velocità (*cruise control*) di un'autovettura. Lo scopo è mantenere la velocità il più vicino possibile a un valore desiderato, sia che la strada sia pianeggiante, in salita o in discesa. Non è sufficiente mantenere semplicemente la valvola a farfalla (*throttle*) in una posizione fissa. Il sistema di controllo deve misurare continuamente la velocità dell'auto e regolare l'apertura della valvola a farfalla per mantenere la velocità desiderata. Pertanto occorre un sensore che misuri la velocità dell'auto e che la indichi in un formato digitale adatto all'uso da parte del calcolatore.

Il calcolatore può quindi determinare se si deve variare l'apertura della valvola a farfalla e può inviare comandi a un attuatore per far sì che ciò avvenga. Nelle autovetture si trovano molti altri sensori. Essi comprendono i dispositivi che misurano i livelli di vari fluidi, compreso il carburante, l'olio e il liquido di raffreddamento del motore. I sensori possono essere utilizzati

```
/* Definisci indirizzi dei registri */
#define   PAOUT   (volatile unsigned char *) 0xFFFFFFF1
#define   PADIR   (volatile unsigned char *) 0xFFFFFFF2
#define   PBIN    (volatile unsigned char *) 0xFFFFFFF3
#define   PBOUT   (volatile unsigned char *) 0xFFFFFFF4
#define   PBDIR   (volatile unsigned char *) 0xFFFFFFF5
#define   CNTM    (volatile unsigned int *) 0xFFFFFFD0
#define   COUNT   (volatile unsigned int *) 0xFFFFFFD4
#define   CTCON   (volatile unsigned char *) 0xFFFFFFD8

void main()
{
    unsigned int counter_value, total_count;
    unsigned int actual_time, seconds, tenths, hundredths;

    /* Inizializza le porte parallele */
    *PADIR = 0xFF;                  /* Configura la porta A. */
    *PBDIR = 0xF4;                  /* Configura la porta B. */
    *PAOUT = 0x0;                   /* Spegni lo schermo */
    *PBOUT = 0x4;                   /* e il LED. */

    /* Avvia il test. */
    while (1)                       /* Ciclo continuo. */
    {
      while ((*PBIN & 0x2) != 0);   /* Attendi che il pulsante Avvio sia premuto. */

      /* Attesa di 3 secondi, quindi accendi il LED */
      *CNTM = 0x11E1A300;           /* Imposta il valore del temporizzatore a 300.000.000 */
      *CTCONT = 0x1;                /* Avvia il temporizzatore. */
      while ((*CTSTAT & 0x1) == 0); /* Attendi finché il temporizzatore non raggiunge zero.*/
      *PBOUT = 0x0;                 /* Accendi il LED. */

      /* Inizializza il processo di conteggio */
      counter_value = 0;
      *CNTM = 0xFFFFFFFF;           /* Imposta il valore iniziale del contatore. */
      *CTCONT = 0x1;                /* Avvia il conteggio. */

      while ((*PBIN & 0x1) != 0);   /* Attendi che il pulsante Stop sia premuto. */

      /* Il pulsante Stop è stato premuto – arresta il conteggio */
      *CTCONT = 0x2;                /* Arresta il contatore. */
      *PBOUT = 0x4;                 /* Spegni il LED. */
      counter_value = *COUNT;       /* Leggi il contenuto del contatore. */

      /* Calcola il conteggio totale */
      total_count = (0xFFFFFFFF – counter_value);

      /* Converti il conteggio in tempo */
      actual_time = total_count /1000000;      /* Tempo in centesimi di secondi */
      seconds = actual_time / 100;
      tenths = (actual_time – seconds * 100)/ 10;
      hundredths = actual_time – (seconds * 100 + tenths * 10);

      /* Visualizza il tempo trascorso */
      *PAOUT = ((seconds < < 4) | tenths);
      *PBOUT = ((hundredths < < 4) | 0x4);   /* Tieni spento il LED */
    }
}
```

Figura 10.11 Programma C per il cronometro di riflessi.

per monitorare la temperatura del liquido di raffreddamento, la pressione degli pneumatici, la tensione della batteria.

Alcuni sensori sono semplici, come quelli che rilevano se una persona o un animale passa sotto una porta di un garage che si sta chiudendo. Altri possono essere molto sofisticati. Si danno nel seguito alcuni esempi di sensori.

Sensori di posizione

Si consideri un calcolatore embedded che controlla il movimento della mano di un robot. La posizione della mano è determinata dalla posizione angolare delle articolazioni di spalla, gomito e polso. Per monitorare la posizione della mano, il calcolatore deve misurare la posizione angolare di ciascuna di queste articolazioni.

Un semplice trasduttore che misura la posizione angolare di un albero rotante rispetto alla sua sede è un potenziometro. Questo consiste di una resistenza avvolta intorno a una base circolare fissata all'alloggiamento e di un contatto scorrevole attaccato all'albero di cui deve essere monitorata la posizione. Man mano che l'albero ruota, il punto di contatto sulla resistenza muta. Quando sono collegate a una sorgente di tensione, come mostrato in Figura 10.12, le due parti della resistenza ai lati del punto di contatto formano un partitore di tensione. La tensione di uscita di questo circuito è data da:

$$V_{out} = \frac{R_2 V_{in}}{R_1 + R_2}$$

Questa tensione alimenta un circuito convertitore A/D per convertire il valore analogico in una rappresentazione digitale. Il microcontrollore in Figura 10.3 comprende un convertitore A/D da usare in tali situazioni.

Alcuni sensori producono direttamente un'uscita digitale. Per esempio, un disco codificato può essere fissato a un albero la cui posizione va monitorata. Il codice sul disco è sotto forma di aree trasparenti e opache organizzate in anelli concentrici, come mostrato nella Figura 10.13. Il disco è posizionato fra coppie di LED e fotorivelatori, una coppia per ciascuna regione anulare. Ogni fotorivelatore è collegato a un circuito che produce un segnale logico 1 quando si rileva la luce e 0 altrimenti. Quindi man mano che il disco ruota, i due fotorivelatori mostrati in figura producono i numeri binari di 2 bit 00, 01, 10 e 11, che rappresentano la posizione angolare dell'albero.

Sensori di temperatura

Molti tipi di sensori sfruttano mutamenti di proprietà dei materiali via via che temperatura o umidità cambiano, o per via di deformazioni provocate da stiramento, compressione o pie-

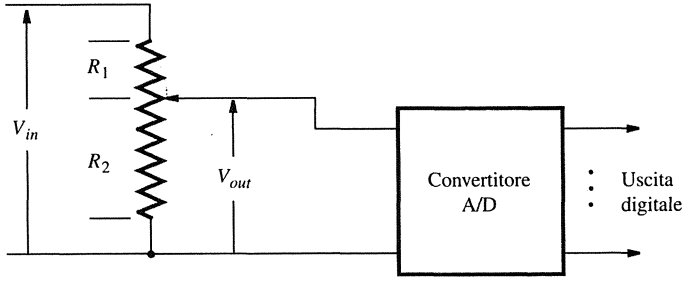


Figura 10.12 Sensore con partitore di tensione.

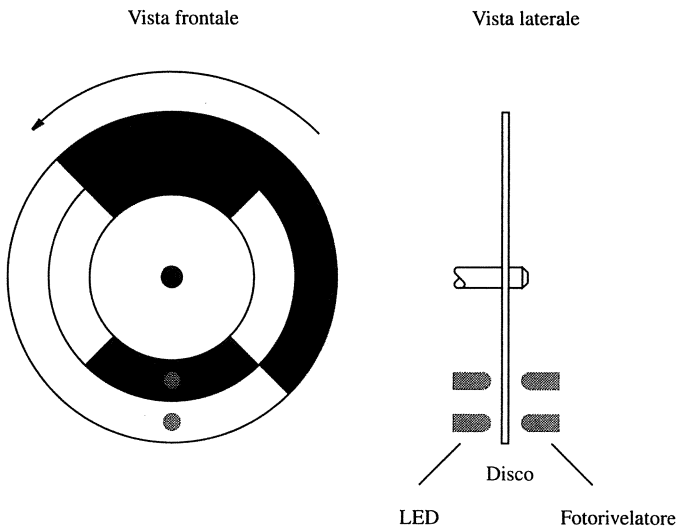


Figura 10.13 Sensore ottico di posizione.

gamento. La resistenza di un conduttore muta con la temperatura, e l'effetto è molto più pronunciato in alcuni materiali rispetto ad altri. Quindi, usando un partitore di tensione simile a quello in Figura 10.12, si può costruire un sensore di temperatura realizzando un resistore R_2 da un materiale con elevata sensibilità alle variazioni di temperatura ed esponendolo all'ambiente di cui si deve misurare la temperatura. Quando la temperatura cambia, la tensione V_{out} cambia. Quindi dopo avere convertito V_{out} in una rappresentazione digitale, il valore può essere letto da un calcolatore come una misura di temperatura. Altri tipi di sensori di temperatura usano un condensatore quale elemento sensibile. La capacità di un condensatore muta al variare delle proprietà del dielettrico (il materiale isolante) usato per la sua costruzione. Alcuni materiali ceramici sono adatti all'uso in sensori di temperatura.

Sensori di pressione

Un sensore, largamente in uso, basato sulla misura di cambiamenti di resistenza è l'estensimetro, che consiste di un resistore disteso su una pellicola flessibile. Il valore della resistenza cambia quando la pellicola viene stirata.

Si supponga di voler monitorare la pressione dell'aria all'interno di un contenitore chiuso, come uno pneumatico di un'auto. Si può costruire un sensore di pressione usando un piccolo cilindro con una membrana attaccata a un'estremità e un estensimetro montato sulla membrana. Il cilindro viene inserito nel contenitore in modo tale che la membrana blocchi il flusso dell'aria dal contenitore. Poiché il contenitore è pressurizzato, la membrana si stira e la resistenza dell'estensimetro cambia. Questa variazione di resistenza può essere misurata usando un circuito simile a quello in Figura 10.12.

Sensori di velocità

Un piccolo generatore elettrico fissato a un albero rotante produce una tensione proporzionale alla velocità di rotazione. Il valore della tensione di uscita del generatore può essere convertito

in formato digitale e letto da un calcolatore per misurare la velocità. Tale dispositivo viene spesso detto tachimetro.

Un altro approccio consiste nel generare impulsi elettrici durante la rotazione dell'albero. Per esempio, un disco dentato può essere solidale con l'albero. Quando il disco ruota, un sensore elettromagnetico o ottico opportunamente posizionato produce un impulso elettrico per ciascun dente che oltrepassa il sensore. La velocità di rotazione viene determinata contando il numero di impulsi generati durante un periodo di tempo fissato o misurando il ritardo di tempo fra impulsi successivi.

10.5.2 Attuatori

Un attuatore è un dispositivo che fa sì che un componente meccanico si muova dopo aver ricevuto un comando. La Figura 10.14 illustra un principio fondamentale usato per costruire una varietà di attuatori. Un filo elettrico è avvolto a spirale attorno a un cilindro che contiene un nucleo mobile, chiamato armatura. Questa struttura è detta solenoide. L'armatura consiste di un materiale magnetico, come il ferro. Essa è tenuta in posizione parzialmente inserita nel cilindro da una molla, come mostrato. Quando la corrente attraversa la bobina, si produce un campo magnetico che tira l'armatura verso l'interno del cilindro. Quando la corrente smette di scorrere, l'armatura viene tirata indietro alla sua posizione di riposo dalla molla. Il movimento dell'armatura può essere utilizzato per aprire una valvola d'acqua, o chiudere un interruttore elettrico che attiva un motore. Questa disposizione può essere, per esempio, usata nel motorino di avviamento di un'auto.

Una combinazione di un motore e di un sensore può essere usata per portare un oggetto a una posizione desiderata. Quando il motore è acceso, l'oggetto inizia a muoversi. Il calcolatore può quindi utilizzare il sensore per verificare ripetutamente se la posizione desiderata sia stata raggiunta. In una macchina fotografica con messa a fuoco automatica il motore fa ruotare il meccanismo di messa a fuoco. Al tempo stesso, il calcolatore utilizza algoritmi di analisi dell'immagine per verificare se l'immagine è correttamente a fuoco. Quando si ha la migliore messa a fuoco, si arresta il motore.

Un altro tipo utile di attuatore è un motore passo-passo. Questo motore fa ruotare il suo asse in risposta a comandi sotto forma di impulsi. Ogni impulso provoca la rotazione dell'asse di una quantità predefinita, per esempio alcuni gradi. Un motore passo-passo è utile in applicazioni che richiedono un controllo preciso della posizione di un oggetto, come un braccio di un robot.

I sensori e gli attuatori di cui sopra costituiscono l'interfaccia fra il calcolatore e l'ambiente fisico in cui è inserito. Essi forniscono al calcolatore la capacità di monitorare lo stato di vari componenti del sistema e fanno sì che le azioni desiderate abbiano luogo. I sensori forniscono informazione sullo stato del sistema al calcolatore. Gli attuatori accettano comandi per attuare la loro funzione. Il compito del calcolatore è di realizzare gli algoritmi di controllo

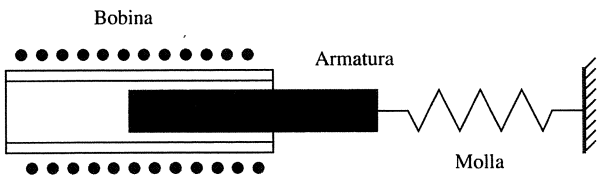


Figura 10.14 Un attuatore con solenoide.

necessari per l'applicazione. Esso inoltre riceve comandi dall'utente e visualizza informazioni per l'utente.

10.5.3 Esempi di applicazione

Si considerano ora brevemente due applicazioni per illustrare l'uso di sensori e attuatori.

Controllo del riscaldamento domestico

Il riscaldamento e il condizionamento in casa sono controllati da un dispositivo noto come termostato. Si consideri il controllo della caldaia. Inserendo un microcontrollore nel termostato, è possibile realizzare funzionalità quali il cambiamento automatico della temperatura desiderata in momenti diversi della giornata o in fine di settimana. Tale termostato comprende:

- un sensore di temperatura;
- un circuito che genera un segnale di uscita per accendere e spegnere la caldaia;
- un temporizzatore per tenere traccia dell'orario del giorno e del giorno della settimana;
- pulsanti che permettono all'utente di inserire le impostazioni desiderate;
- uno schermo.

In questa discussione si esaminerà solo il meccanismo di controllo della temperatura. Si supponga di usare il sensore di temperatura resistivo descritto nel Paragrafo 10.5.1 e che la sua tensione di uscita sia collegata al convertitore A/D nel microcontrollore. Si usa un solenoide per attivare un interruttore che accende o spegne la caldaia. L'algoritmo di controllo è realizzato da un programma nel microcontrollore. Si supponga che la temperatura desiderata sia fissata a T gradi. Si permetterà alla temperatura della casa di deviare da questa impostazione per una piccola quantità ΔT sopra o sotto il valore desiderato. Quindi, l'attività di controllo può essere realizzata eseguendo ripetutamente le seguenti azioni.

1. Campionare il sensore di temperatura per ottenere la tensione di ingresso che rappresenta la temperatura.
2. Se la temperatura è inferiore a $T - \Delta T$, accendere la caldaia.
3. Se la temperatura è superiore a $T + \Delta T$, spegnere la caldaia.

Il resto del programma del calcolatore si occupa di usare il temporizzatore per tenere traccia del trascorrere del tempo, di ricevere i dati in ingresso dall'utente tramite i pulsanti e di inviare informazioni allo schermo.

Controllo di velocità

Il sistema di regolazione automatica della velocità che si trova in molte autovetture richiede i seguenti componenti.

- Un sensore per misurare la velocità dell'albero motore dell'auto.
- Un attuatore per controllare la posizione della valvola a farfalla del carburante.
- Pulsanti di ingresso per il guidatore per attivare il sistema e impostare la velocità desiderata.

A differenza del sistema di riscaldamento domestico, in questa applicazione il parametro da controllare, la posizione della valvola a farfalla, varia con continuità fra due estremi, completamente chiusa e completamente aperta. Il semplice algoritmo di controllo acceso/spento

utilizzato nell'esempio precedente non è adatto. Quando si controlla una quantità variabile come la posizione della valvola a farfalla, sono necessari algoritmi più sofisticati, la cui discussione esula dall'ambito del presente testo. Tuttavia, le azioni desiderate possono essere brevemente descritte come segue. Il calcolatore misura ripetutamente la velocità dell'auto e la confronta con il valore impostato. La differenza è un errore, ϵ, che può essere positivo o negativo. A seconda del valore di ϵ, il calcolatore adegua di un po' la posizione della valvola a farfalla. L'obiettivo è di mantenere un valore di ϵ che sia il più vicino possibile allo zero. In effetti, il calcolatore imita le decisioni e le azioni di un conducente che guida manualmente e mantiene una velocità costante.

10.6 Famiglie di microcontrollori

Dopo aver presentato un esempio di microcontrollore nel Paragrafo 10.3, esamineremo brevemente ora l'ampia gamma di chip microcontrollori disponibili in commercio. Molte applicazioni embedded non richiedono un processore potente. Sicuramente il forno a microonde discusso nel Paragrafo 10.1.1 non ha bisogno di un microcontrollore potente, poiché i suoi requisiti di calcolo sono piuttosto modesti e i tempi di reazione non sono impegnativi. Per tali applicazioni è preferibile usare un chip che abbia un processore semplice, ma che contenga sufficiente memoria e risorse di I/O per realizzare tutte le funzioni del controllore. La macchina fotografica digitale, discussa nel Paragrafo 10.1.2, ha requisiti di calcolo molto più esigenti, e quindi richiede un processore più potente.

I processori possono essere caratterizzati dal numero di bit di dati che trattano in parallelo quando accedono ai dati in memoria. I microcontrollori più potenti sono verosimilmente basati su un processore a 32 bit, con una larghezza del bus di dati di 32 bit. Questo è il caso di alcuni microcontrollori basati sull'architettura ARM. È anche possibile avere un processore che ha una struttura interna a 32 bit, ma una larghezza di 16 bit del bus di dati per l'accesso alla memoria. Ne è un esempio la famiglia di microcontrollori 68K/ColdFire Freescale. Alcuni dei più diffusi microcontrollori sono chip a 8 bit. Essi sono molto più economici, ma sufficientemente potenti per soddisfare le esigenze di un gran numero di applicazioni embedded. Esistono anche chip più piccoli a 4 bit, interessanti per la loro semplicità e per il costo estremamente basso.

10.6.1 Microcontrollori basati su processore Intel 8051

Nei primi anni '80, Intel Corporation ha introdotto un chip microcontrollore chiamato 8051. Questo chip ha l'architettura di base della famiglia di microprocessori Intel 8080, che usava chip a 8 bit per applicazioni di calcolo a uso generale. Il chip 8051 ha guadagnato rapidamente diffusione ed è diventato uno dei chip di più largo uso nella pratica. Ha quattro porte di I/O a 8 bit, una UART, e due circuiti a 16 bit di conteggio/temporizzazione. Contiene anche 4 Kbyte di ROM e 128 byte di memoria RAM. La versione EPROM del chip, che ha 4 Kbyte di EPROM anziché ROM, è disponibile con il nome 8751.

Sono disponibili un certo numero di altri chip basati su architettura 8051, che forniscono miglioramenti vari. Per esempio, il chip 8052 ha 8 Kbyte di ROM e 256 byte di RAM, nonché un contatore/temporizzatore supplementare. La sua versione EPROM è detta 8752.

L'architettura 8051 è stata sviluppata da Intel. Successivamente, un certo numero di altri produttori di semiconduttori hanno prodotto chip che sono identici a quelli della famiglia 8051, o che ne migliorano alcune caratteristiche ma sono peraltro del tutto compatibili con l'8051.

10.6.2 Microcontrollori Freescale

Motorola ha avuto una posizione dominante come produttore di chip microprocessori negli anni '80. I loro microprocessori a 8 bit più diffusi sono diventati la base per i loro microcontrollori. Freescale Semiconductor Inc. è il successore di Motorola in questo settore. Freescale produce una vasta gamma di microcontrollori, basati su differenti nuclei processori.

Microcontrollore 68HC11

I più diffusi microprocessori a 8 bit Motorola erano il 6800 e il 6809. Il 68HC11 è stato successivamente introdotto come chip microcontrollore che realizza un sovrainsieme di istruzioni del 6800. Ha cinque porte di I/O che possono essere utilizzate per una varietà di scopi. La struttura di I/O è dotata di due interfacce seriali. Vi sono anche circuiti contatori/temporizzatori in grado di operare in un certo numero di modalità diverse.

La quantità di memoria inclusa nei chip 68HC11 varia da una ROM di 8 Kbyte, EEPROM di 512 byte e RAM di 256 byte nel chip originale, a una ROM di 12 Kbyte, EEPROM di 512 byte e RAM di 512 byte in chip successivi.

Microcontrollori 68K

Una famiglia di microcontrollori, detta 68K, si basa sul processore a 32 bit 68000. Per ridurre il numero di piedini, il bus dati esterno è di solo 16 bit. Questi chip hanno porte parallele e seriali, contatori/temporizzatori e circuiti di conversione A/D. La quantità di memoria sul chip dipende dal particolare chip. Per esempio il chip 68376 ha 8 Kbyte di EEPROM e 4 Kbyte di RAM.

Microcontrollori ColdFire

L'architettura dell'insieme di istruzioni del 68000 fornisce la base per il processore a 32 bit ColdFire, presentato negli approfondimenti applicativi dei Capitoli 2 e 3, e per i microcontrollori MCF5xxx, detti processori embedded ColdFire. La loro caratteristica distintiva è una struttura a pipeline che porta a migliori prestazioni. La larghezza del bus di dati esterno può essere 16 bit o 32 bit, a seconda del chip microcontrollore. Il nucleo del processore ColdFire è destinato anche per l'uso nell'ambiente dei *sistemi su un chip* (*system-on-a-chip*), di cui si parla nel Caso di studio C1.

Microcontrollori PowerPC

La famiglia di microprocessori Freescale a 32 bit ad alte prestazioni, nota come PowerPC, si basa su un'architettura in stile RISC. Questa architettura del processore è disponibile anche nella forma di microcontrollore, nei chip che compongono la famiglia MPC5xx.

10.6.3 Microcontrollori ARM

L'architettura ARM, presentata negli approfondimenti applicativi dei Capitoli 2 e 3, è interessante per sistemi embedded dove si richiede una notevole capacità di calcolo, e dove il costo e il consumo energetico devono essere relativamente bassi. Un obiettivo fondamentale della progettazione del processore ARM è renderlo adatto per l'uso nell'ambiente dei sistemi su un chip. I microcontrollori ARM sono disponibili anche come chip separati.

Si è avuta una progressione di nuclei processori ARM destinati alle applicazioni embedded, fra cui ARM6, ARM7, ARM9, ARM10 e ARM Cortex. L'architettura ARM di base utilizza un'organizzazione a 32 bit e un insieme di istruzioni in cui tutte le istruzioni sono a 32 bit. Esiste un'altra versione, detta Thumb, che ha istruzioni a 16 bit e trasferimenti di dati a 16 bit. La versione Thumb usa un sottoinsieme delle istruzioni ARM, codificate per stare in un formato a 16 bit. Ha anche un minore numero di registri rispetto all'architettura ARM. Un vantaggio della versione Thumb è che necessita di una memoria significativamente più ridotta

per immagazzinare programmi composti da istruzioni a 16 bit altamente codificate. A tempo di esecuzione, ciascuna istruzione Thumb è estesa a un'istruzione ARM normale da 32 bit. Così, un nucleo processore ARM che riconosce Thumb contiene un *decompressore Thumb* in aggiunta ai suoi normali circuiti.

10.7 Aspetti di progettazione

Il progettista di un sistema embedded deve prendere molte decisioni importanti. La natura dell'applicazione o il prodotto che deve essere progettato presenta determinati requisiti e vincoli. In questo paragrafo si considerano alcuni degli aspetti più importanti che il progettista si trova ad affrontare.

Costo

Il costo dell'elettronica in molte applicazioni embedded deve essere basso. Si ottiene la soluzione meno costosa quando un singolo chip è sufficiente alla realizzazione di tutte le funzioni dovute. Ciò è possibile solo se il chip ha capacità di I/O sufficienti a soddisfare le esigenze dell'applicazione e sufficiente memoria sul chip stesso per contenere i programmi e dati necessari.

Capacità di I/O

I chip microcontrollori forniscono una varietà di risorse di I/O, che vanno da semplici porte parallele e seriali a contatori, temporizzatori e circuiti di conversione A/D e D/A. Il numero di linee di I/O disponibili è importante. Senza sufficienti linee di I/O, è necessario impiegare un circuito esterno per colmare il deficit. Questo è illustrato dall'esempio di cronometro di riflessi in Figura 10.10, nel quale si usano circuiti di decodifica esterni per pilotare gli schermi a 7 segmenti tramite i segnali BCD a 4 bit prodotti dal microcontrollore. Se il microcontrollore avesse 4 porte parallele, anziché 2, sarebbe possibile collegare ciascuno degli schermi a 7 segmenti a una porta. Il programma di controllo genererebbe quindi direttamente i segnali di 7 bit necessari per pilotare i 7 singoli segmenti in ciascuno schermo.

Dimensione

I chip microcontrollori sono disponibili in varie dimensioni. Se un'applicazione può essere gestita in modo adeguato con un microcontrollore a 8 bit non ha senso usare un chip a 16 o 32 bit che sarà probabilmente più costoso, fisicamente più grande, e che consuma più energia. La maggior parte delle applicazioni pratiche si possono gestire con chip relativamente piccoli.

Consumo energetico

Il consumo energetico è una considerazione importante in tutte le applicazioni informatiche. In sistemi ad alte prestazioni il consumo di energia tende a essere elevato e richiede un meccanismo per dissipare il calore generato. In molte applicazioni embedded la potenza consumata è così bassa che la dissipazione del calore non è un problema. Tuttavia, per queste applicazioni sono spesso in gioco prodotti alimentati a batteria, così la vita della batteria, che dipende dal consumo di energia, è un fattore significativo.

Memoria sul chip

L'inclusione di memoria su un chip microcontrollore permette la realizzazione di semplici applicazioni embedded su singolo chip. La dimensione e il tipo di memoria hanno implicazioni significative. Una quantità relativamente piccola di RAM può essere sufficiente a memorizzare dati durante i calcoli. Una memoria a sola lettura più grande è necessaria per

memorizzare i programmi. Tale memoria può essere di tipo ROM, PROM, EPROM, EEPROM o Flash. Per prodotti a grande volume di produzione, le scelte più economiche sono i microcontrollori con ROM. Tuttavia, questa è anche la scelta meno flessibile poiché i contenuti della ROM sono permanentemente fissati al momento in cui il chip è fabbricato. La flessibilità più grande è offerta dalle memorie EEPROM e flash, che si possono programmare più volte.

Per le applicazioni che hanno requisiti di stoccaggio maggiori, è necessario usare una memoria esterna. Alcuni microcontrollori non hanno alcuna memoria su chip. Essi sono tipicamente destinati ad applicazioni più sofisticate per le quali è necessaria una notevole quantità di memoria, che non può essere realizzata all'interno del chip microcontrollore.

Prestazioni

La considerazione delle prestazioni non è importante quando i microcontrollori sono utilizzati in applicazioni quali elettrodomestici e giocattoli. In questi casi si possono scegliere chip piccoli e poco costosi. Tuttavia, in applicazioni quali macchine fotografiche digitali, telefoni cellulari e alcuni videogiochi portatili, è essenziale disporre di prestazioni più elevate. Alte prestazioni richiedono chip più potenti, il che significa un costo più elevato e un maggior consumo di energia. Poiché l'applicazione è spesso alimentata a batteria, è anche importante minimizzare i consumi. Si richiedono vari compromessi per soddisfare questi obiettivi contrastanti.

Software

Si hanno molti vantaggi con l'uso di linguaggi di alto livello per i programmi delle applicazioni. Essi facilitano il processo di sviluppo del programma e rendono il software facile da mantenere e modificare in futuro. Tuttavia, vi sono alcuni casi in cui può essere prudente o necessario ricorrere al linguaggio assemblativo. Un programma in linguaggio assemblativo ben congegnato può generare un codice oggetto dal 10 al 20% più compatto (in termini di quantità di memoria necessaria) rispetto al codice prodotto da un compilatore. Se un'applicazione embedded è basata su un microcontrollore che ha una memoria su chip limitata, un vantaggio importante è che il codice necessario possa essere contenuto nella memoria disponibile sul chip, evitando così la necessità di memoria esterna.

La limitata capacità di RAM disponibile sul chip è una considerazione importante per un progettista di sistemi. Questa memoria viene utilizzata per immagazzinare i dati dinamici, come un buffer temporaneo, e per realizzare una pila. Quando si realizza il programma in un linguaggio di alto livello come C, occorre prestare attenzione a garantire che la quantità totale di codice e dati non superi la memoria disponibile.

Insieme di istruzioni

Un altro aspetto importante è la natura dell'insieme di istruzioni del processore utilizzato. Le istruzioni in stile CISC portano a codice più compatto rispetto alle istruzioni in stile RISC. Quindi, la scelta del processore ha un impatto sulla dimensione del codice. Un esempio interessante di un approccio che risolve questo problema è fornito dalla versione Thumb dell'architettura ARM, nella quale è stato modificato un insieme di istruzioni in stile RISC progettato per processori a 32 bit per avere una codifica migliore che utilizza istruzioni a 16 bit, come discusso nel Paragrafo 10.6.3. I programmi scritti per le versioni Thumb sono fino al 30% più compatti rispetto a quelli scritti per l'architettura ARM completa.

Strumenti di sviluppo

I progettisti di sistemi digitali fanno molto affidamento sugli strumenti di sviluppo. Questi strumenti includono pacchetti software per la progettazione assistita dal calcolatore o CAD (*Computer-Aided Design*), compilatori, assemblatori e simulatori per i processori. La gamma e la disponibilità di strumenti spesso dipende dalla scelta del processore embedded. È inoltre interessante avere un supporto di terzi, quali fonti alternative di strumenti e documentazione. Una buona documentazione e consigli utili dal produttore (se necessario) sono estremamente importanti.

Collaudabilità e affidabilità

I circuiti stampati sono spesso difficili da collaudare, soprattutto se sono densamente riempiti con chip. Il processo di collaudo è notevolmente semplificato se l'intero sistema è progettato per essere facilmente collaudabile. Un chip microcontrollore può includere circuiti che rendono più facile il collaudo di circuiti stampati che contengono il chip. Per esempio, alcuni microcontrollori dispongono di una *porta di accesso per il collaudo* (*test access port*) compatibile con lo standard IEEE 1149.1 per un'architettura collaudabile [1].

Le applicazioni embedded richiedono robustezza e affidabilità. Il ciclo di vita di un prodotto tipico dovrebbe essere di almeno cinque anni. Ciò è in contrasto con i calcolatori personali, che tendono a essere considerati obsoleti dopo un tempo più breve.

Sommario

Questo capitolo ha fornito un'introduzione alla progettazione di sistemi informatici embedded. Non è stato utilizzato un microcontrollore specifico disponibile in commercio per questa discussione, poiché i principi presentati sono di carattere generale e riguardano le questioni significative che il progettista di un sistema embedded deve affrontare.

È particolarmente importante comprendere la stretta interazione di hardware e software. Le scelte progettuali possono riguardare compromessi fra la scansione dell'I/O e le interruzioni, fra differenti insiemi di istruzioni in termini di funzionalità e compattezza del codice, fra il consumo di potenza e le prestazioni e così via.

In questo capitolo si è discusso l'uso di microcontrollori nella realizzazione di sistemi embedded. Nel Caso di studio C1 si mostra come si possano impiegare chip FPGA per costruirli. Si focalizza in particolare l'approccio di sistema su un chip, che mira a realizzare un intero sistema su singolo chip.

Infine, si sono considerate le caratteristiche dei sistemi embedded che li rendono differenti dai calcolatori a uso generale. I principi di base sono gli stessi. Un progettista di sistemi embedded deve avere una buona conoscenza dell'organizzazione del calcolatore, che comprende una conoscenza approfondita di: insiemi di istruzioni, esecuzione di programmi, tecniche di ingresso/uscita, strutture di memoria, schemi di interfacciamento per gestire una varietà di dispositivi che possono essere inclusi nel sistema. Allora, che cosa è diverso?

Un calcolatore a uso generale esegue programmi applicativi arbitrari, inclusi quelli utilizzati per creare o modificare altri programmi. Generalmente ogni programma è eseguito per un periodo limitato di tempo, come effetto dell'algoritmo di elaborazione o del controllo da parte dell'utente. Vi è la necessità di una grande memoria principale che è esterna al chip del processore per soddisfare i requisiti potenzialmente grandi di alcuni programmi applicativi. Sono necessari grandi dispositivi di memorizzazione secondaria per contenere i file di programmi e dati. Si usa un sistema operativo sofisticato per controllare tutte le risorse del sistema informatico.

Al contrario, un sistema embedded esegue tipicamente un singolo programma applicativo che non è da modificare. Il programma va automaticamente in esecuzione quando il sistema viene acceso, e continua fino a quando il sistema è spento. I requisiti di memoria tendono a essere modesti. Spesso è possibile impiegare un microcontrollore con adeguata memoria su chip. Non è necessario un sistema operativo potente. In molti casi, non vi è alcun software di sistema operativo. Il programma singolo è in continua esecuzione e ha accesso diretto a tutte le risorse di elaborazione, memoria e ingresso/uscita.

La discussione è stata focalizzata su sistemi embedded di fascia bassa (*low-end*), soggetti a requisiti di elaborazione relativamente modesti. Tali sistemi incarnano i principi fondamentali

delle applicazioni embedded. Ma vi sono anche molti sistemi embedded di fascia alta (*high-end*), come quelli in uso negli aerei e nei treni ad alta velocità, che richiedono capacità di calcolo molto più elevate e sono spesso realizzati impiegando più processori. Il Capitolo 11 si occupa delle questioni attinenti ai sistemi multiprocessore.

Note bibliografiche

1. Institute of Electrical and Electronics Engineers, *Test Access Port and Boundary-scan Architecture*, IEEE Standard 1149.1, May 1990.

11 Elaborazione parallela e prestazioni

In questo capitolo

- Multithreading
- Elaborazione vettoriale in processori di uso generale e in processori grafici
- Multiprocessori a memoria condivisa
- Coerenza di cache per dati condivisi
- Scambio di messaggi in sistemi a memoria distribuita
- Programmazione parallela
- Modellazione matematica della prestazione

Nel Capitolo 6 si descrivono il pipelining delle istruzioni e il funzionamento superscalare come tecniche di progettazione del processore volte ad aumentare il tasso di esecuzione delle istruzioni, riducendo così il tempo di esecuzione. Nel Capitolo 8 si presentano le cache come un modo per ridurre la latenza media di accesso a istruzioni e dati.

Questo capitolo descrive tre tecniche aggiuntive per migliorare le prestazioni, vale a dire il multithreading, l'elaborazione vettoriale e il multiprocessing. Queste tecniche sono realizzate nei chip di processori multicore impiegati in calcolatori di uso generale. Esse aumentano le prestazioni migliorando l'utilizzazione delle risorse di elaborazione ed effettuando più operazioni in parallelo.

11.1 Multithreading hardware

Il sistema operativo (SO) permette il *multitasking* (multiprocessualità) di programmi diversi nello stesso processore eseguendo commutazioni di contesto fra i programmi. Come spiegato nel Paragrafo 4.9, un programma, insieme a tutte le informazioni che descrivono il suo stato attuale di esecuzione, è considerato dal SO come un'entità detta *processo*. Per ogni processo il SO mantiene l'informazione sulla memoria e altre risorse che gli ha assegnato. I processi possono essere associati con applicazioni, quali navigazione su web, elaborazione di testi e riproduttori di musica, che un utente ha aperto in un calcolatore.

Ogni processo ha un corrispondente thread, che è un percorso di esecuzione indipendente in un programma. Più precisamente, si usa il termine *thread* per riferirsi a un thread di controllo il cui stato consiste dei contenuti del contatore di programma e di altri registri del processore. Come si vedrà nel Paragrafo 11.6, è possibile che più thread eseguano porzioni di un programma in parallelo, come se corrispondessero a programmi separati. Due o più thread possono girare su processori diversi, eseguendo la stessa parte di un programma su dati diversi o eseguendo parti diverse di un programma. Anche thread per programmi diversi possono girare su processori diversi. Tutti i thread che fanno parte di un singolo programma sono eseguiti nello stesso spazio di indirizzamento e associati allo stesso processo.

In questo paragrafo, ci si concentra sul multitasking tramite il quale si eseguono due o più programmi sullo stesso processore e ogni programma ha un solo thread. Il Paragrafo 4.9 descrive la tecnica della divisione in quanti di tempo (*time slicing*), attraverso la quale il SO seleziona un processo fra quelli che non sono attualmente bloccati e ne permette l'esecuzione per un breve quanto di tempo. Solo il thread corrispondente al processo selezionato è attivo durante il quanto di tempo. La commutazione di contesto alla fine del quanto di tempo fa sì che il SO selezioni un altro processo, il cui corrispondente thread diventa attivo durante il quanto successivo. Un'interruzione di un temporizzatore invoca una routine di servizio dell'interruzione nel SO per passare da un processo all'altro.

Per gestire più thread in modo efficiente, si munisce un processore di vari banchi identici di registri, fra cui contatori di programma multipli. Ciascun banco di registri può essere dedicato a un thread diverso. Così non si perde tempo durante una commutazione di contesto per salvare e ripristinare contenuti di registri. Si dice che il processore utilizza una tecnica detta *multithreading hardware*.

Con più banchi di registri, la commutazione di contesto è semplice e veloce. Tutto ciò che occorre fare è cambiare un puntatore hardware nel processore per utilizzare un banco differente di registri per prelevare ed eseguire le istruzioni successive. Il passaggio a un thread diverso può essere completato entro un ciclo di clock. Lo stato del thread precedentemente attivo è salvato nel proprio banco di registri.

Il passaggio a un thread diverso può essere attivato in qualsiasi momento dal verificarsi di un evento specifico, invece che alla fine di un intervallo di tempo fisso. Per esempio, si può verificare una cache miss quando si sta eseguendo un'istruzione Load o Store per il thread attivo. Invece di rimanere in stallo, mentre si accede alla memoria principale più lenta per risolvere la cache miss, un processore può passare rapidamente a un thread diverso e continuare a prelevare ed eseguire altre istruzioni. Questo è chiamato multithreading *coarse-grained* (a grana grossa), poiché si possono eseguire molte istruzioni per un thread prima che un evento come un cache miss provochi il passaggio a un altro thread.

Un'alternativa al passaggio fra thread su eventi specifici è quella di commutare dopo che ciascuna istruzione sia prelevata. Questo è chiamato multithreading *fine-grained* (a grana fine) o *interleaved* (interallacciato). L'intento è di aumentare il throughput del processore. Ogni nuova istruzione è indipendente dalle sue precedenti di altri thread. Questo dovrebbe ridurre la presenza di stalli a causa di dipendenze di dati. Quindi il throughput può essere aumentato tramite l'interleaving di istruzioni provenienti da più thread, tuttavia ci vuole più tempo per un determinato thread per completare tutte le sue istruzioni. Una forma di multithreading di tipo interleaved con due soli thread viene utilizzato nei processori che realizzano l'architettura Intel IA-32 descritta nel Caso di studio C2.

11.2 Elaborazione vettoriale (SIMD)

Molte applicazioni di calcolo impegnative riguardano programmi che usano cicli per eseguire operazioni su vettori di dati, dove un *vettore* è una sequenza indicizzata di elementi dello stesso tipo, quali numeri interi o numeri in virgola mobile. Quando un processore esegue le istruzioni di un tale ciclo, le operazioni sono eseguite una alla volta sui singoli elementi del vettore. Perciò si devono eseguire molte istruzioni per elaborare tutti gli elementi del vettore.

Un processore può essere potenziato con più ALU. In un processore di questo tipo, si può operare su più elementi in parallelo eseguendovi una singola istruzione. Tali istruzioni sono dette *istruzioni singole su dati multipli* o SIMD (*Single-Instruction Multiple-Data*). Sono anche dette *istruzioni vettoriali*. Si possono usare queste istruzioni solo quando le operazioni eseguite in parallelo sono indipendenti. Questo è noto come *parallelismo di dati*.

I dati di istruzioni vettoriali sono tenuti in *registri vettoriali*, ciascuno dei quali può contenere diversi elementi di dati. Il numero di elementi, L, in ciascun registro vettoriale è detto *lunghezza del vettore*. Questa determina il numero di operazioni che possono essere eseguite in parallelo su più ALU. Se le istruzioni vettoriali sono fornite per diverse dimensioni di elementi di dati usando gli stessi registri vettoriali, L può variare. Per esempio, l'architettura Intel IA-32 ha registri vettoriali da 128 bit che sono utilizzati da istruzioni per lunghezze di vettori che vanno da $L = 2$ fino a $L = 16$, corrispondenti a elementi di dati interi con dimensioni che variano da 64 bit fino a 8 bit.

Si riportano nel seguito alcuni esempi tipici di istruzioni vettoriali per illustrare l'uso di registri vettoriali. Si parte dal presupposto che il codice operativo mnemonico comprenda un suffisso S che specifica la dimensione di ogni elemento dei dati. Questo determina il numero di elementi, L, in un vettore. Per le istruzioni che accedono alla memoria, si usa il contenuto di un registro convenzionale nel calcolo dell'indirizzo effettivo.

L'istruzione vettoriale:

VectorAdd.S V*i*, V*j*, V*k*

calcola L somme utilizzando gli elementi nei registri vettoriali V*j* e V*k* e pone le somme risultanti nel registro vettoriale V*i*. Si utilizzano istruzioni simili per eseguire altre operazioni aritmetiche.

Sono necessarie istruzioni speciali per trasferire elementi dati multipli fra un registro vettoriale e la memoria. L'istruzione:

VectorLoad.S V*i*, X(R*j*)

fa sì che L elementi consecutivi a partire dalla locazione di memoria X+[R*j*] siano caricati nel registro vettoriale V*i*. Analogamente, l'istruzione:

VectorStore.S V*i*, X(R*j*)

fa sì che il contenuto del registro vettoriale V*i* sia immagazzinato in L elementi consecutivi nella memoria.

Vettorizzazione

In un programma sorgente scritto in un linguaggio di alto livello, i cicli che operano su matrici di numeri interi o in virgola mobile sono *vettorizzabili* se le operazioni eseguite in ogni passata sono indipendenti dalle altre passate. L'uso di istruzioni vettoriali riduce il numero di istruzioni da eseguire e consente l'esecuzione parallela di operazioni su più ALU. Un *compilatore vettorizzante* può riconoscere tali cicli, se non sono troppo complessi, e può generare istruzioni vettoriali.

La vettorizzazione di un ciclo è illustrata con il semplice esempio di codice in linguaggio C mostrato in Figura 11.1a. Si supponga che le posizioni iniziali in memoria delle matrici A, B e C siano nei registri R2, R3 e R4. Usando istruzioni convenzionali in linguaggio assemblativo, il compilatore può generare il ciclo mostrato in Figura 11.1b. Il corpo del ciclo ha nove istruzioni, quindi per N passi del ciclo si esegue un totale di $9N$ istruzioni.

Per vettorizzare il ciclo, il compilatore deve riconoscere che il calcolo in ogni passata del ciclo sia indipendente dalle altre passate e che la stessa operazione possa essere eseguita simultaneamente su più elementi. Si supponga per semplicità che il numero di passate, N, sia divisibile per la lunghezza del vettore, L. Le istruzioni Load, Add e Store all'inizio del ciclo vengono sostituite con le corrispondenti istruzioni vettoriali che operano su L elementi alla volta. Di conseguenza, il ciclo vettorizzato richiede solo N/L passate per elaborare tutti i dati nelle matrici. Con L elementi elaborati in ogni passata del ciclo, i puntatori a indirizzi nei registri R2, R3 e R4 sono incrementati di $4L$, e il conteggio nel registro R5 viene decrementato

```
for (i = 0; i < N; i++)
  A[i] = B[i] + C[i];
```

(a) Un ciclo in linguaggio C per sommare elementi di vettori

	Move	R5, #N	R5 è il contatore del ciclo
CICLO:	Load	R6, (R3)	R3 punta a un elemento nella matrice B.
	Load	R7, (R4)	R4 punta a un elemento nella matrice C.
	Add	R6, R6, R7	Somma una coppia di elementi delle matrici.
	Store	R6, (R2)	R2 punta a un elemento nella matrice A.
	Add	R2, R2, #4	Incrementa i tre puntatori alle matrici.
	Add	R3, R3, #4	
	Add	R4, R4, #4	
	Subtract	R5, R5, #1	Decrementa il contatore del ciclo.
	Branch_if_[R5]> 0	CICLO	Ripeti il ciclo se non hai finito.

(b) Istruzioni in linguaggio assemblativo per il ciclo

	Move	R5, #N	R5 conta il numero di elementi da elaborare.
CICLO:	VectorLoad.S	V0, (R3)	Carica *L* elementi dalla matrice B.
	VectorLoad.S	V1, (R4)	Carica *L* elementi dalla matrice C.
	VectorAdd.S	V0, V0, V1	Somma *L* coppie di elementi delle matrici.
	VectorStore.S	V0, (R2)	Memorizza *L* elementi nella matrice A.
	Add	R2, R2, #4*L	Incrementa i puntatori alle matrici di *L* parole.
	Add	R3, R3, #4*L	
	Add	R4, R4, #4*L	
	Subtract	R5, R5, #L	Decrementa di *L* il contatore del ciclo.
	Branch_if_[R5]> 0	CICLO	Ripeti il ciclo se non hai finito.

(c) Forma vettorizzata del ciclo

Figura 11.1 Esempio di vettorizzazione di un ciclo.

di L. Il ciclo vettorizzato è mostrato in Figura 11.1c. Si suppone che l'assemblatore calcoli il valore $4L$ per l'espressione fornita come operando immediato nelle istruzioni Add. Vi sono ancora nove istruzioni nel corpo del ciclo, ma poiché il numero di passate è ora N/L, il numero totale di istruzioni da eseguire è solo $9N/L$.

Cicli vettorizzabili esistono in programmi per applicazioni grafiche e di elaborazione digitale di segnali. Tali cicli hanno molti calcoli indipendenti che possono essere eseguiti contemporaneamente su diversi elementi di dati. Se gran parte del tempo di esecuzione per un'applicazione è dedicato all'esecuzione di cicli di questo tipo, allora la vettorizzazione di questi cicli può ridurre significativamente il tempo di esecuzione totale. L'entità del miglioramento delle prestazioni è limitata dalla lunghezza L del vettore, che determina il numero di ALU che possono operare in parallelo. Per ottenere prestazioni più elevate, si può realizzare in un

altro modo il supporto per l'elaborazione vettoriale (SIMD), come descritto nel paragrafo successivo.

11.2.1 Unità di elaborazione grafica (GPU)

Le crescenti esigenze di elaborazione per la grafica hanno portato allo sviluppo di chip specializzati detti *unità di elaborazione grafica* o GPU (*Graphics Processing Unit*). Lo scopo principale delle GPU è accelerare il gran numero di calcoli in virgola mobile necessari nella grafica tridimensionale ad alta risoluzione, come nei videogiochi. Poiché le operazioni coinvolte in questi calcoli sono spesso indipendenti, un grande chip GPU contiene centinaia di nuclei o *core* di processori semplici con ALU in virgola mobile per eseguire tali calcoli in parallelo.

Su una scheda video sono inclusi un chip GPU e una memoria dedicata a esso. Tale scheda è inserita in un alloggiamento (*slot*) di espansione di un calcolatore *host* (ospite) utilizzando uno standard di interconnessione, come lo standard PCIe discusso nel Capitolo 7. Un piccolo programma viene scritto per i core di elaborazione del chip GPU. Un gran numero di core eseguono questo programma in parallelo: eseguono le stesse istruzioni ma operano su dati diversi. Il processore di uso generale dell'host esegue un programma separato di controllo che invoca il programma della GPU quando occorre. Prima di dare inizio al calcolo GPU, il programma dell'host deve anzitutto trasferire i dati necessari al programma GPU dalla memoria principale alla memoria dedicata della GPU. Dopo che il calcolo viene completato, i dati di uscita risultanti nella memoria dedicata sono trasferiti alla memoria principale.

I core di elaborazione in un chip GPU hanno un insieme di istruzioni e un'architettura hardware specializzati, diversi da quelli di un processore di uso generale. Un esempio è l'*architettura di dispositivi di calcolo unificata* o CUDA (*Compute Unified Device Architecture*) che NVIDIA Corporation impiega per i core nei suoi chip GPU. Per facilitare la stesura di programmi che coinvolgono un processore di uso generale e una GPU, NVIDIA ha sviluppato un'estensione del linguaggio di programmazione C, detta CUDA C [1, 2]. Questa estensione consente di scrivere un singolo programma in C, con l'uso di parole chiave speciali per etichettare le funzioni eseguite dai core di elaborazione in un chip GPU. Il compilatore e gli strumenti software correlati partizionano automaticamente il programma oggetto finale nelle porzioni che sono tradotte in istruzioni macchina per l'host e per il chip GPU. Sono fornite routine di libreria per allocare spazio di memoria dedicata in una scheda video basata su GPU e per trasferire dati fra la memoria principale e la memoria dedicata. Uno standard aperto detto OpenCL è stato anche proposto dall'industria come ambiente (*framework*) di programmazione per sistemi che dispongono di chip GPU di qualsiasi fornitore [3].

11.3 Multiprocessori a memoria condivisa

Un sistema multiprocessore consiste di un certo numero di processori in grado di eseguire simultaneamente operazioni indipendenti. La granularità di queste attività può variare considerevolmente. Un'attività può comprendere alcune istruzioni per una sola passata di un ciclo o migliaia di istruzioni eseguite in un sottoprogramma.

In un multiprocessore a memoria condivisa, tutti i processori hanno accesso alla stessa memoria. Le attività in esecuzione in processori diversi accedono a variabili condivise in memoria usando gli stessi indirizzi. Una grande dimensione della memoria condivisa è probabile. La realizzazione di una grande memoria in un singolo modulo creerebbe un collo di bottiglia quando molti processori facessero richieste di accesso alla memoria contemporaneamente. Si

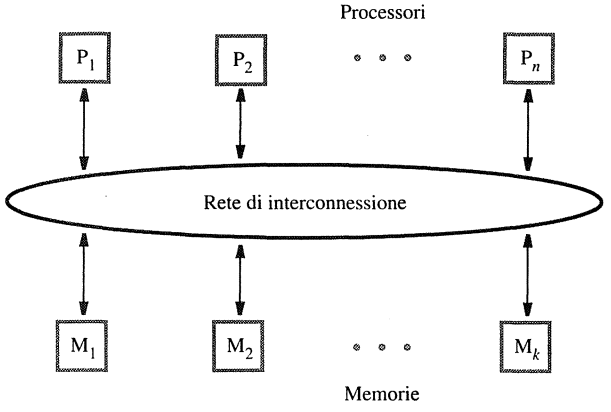


Figura 11.2 Un multiprocessore UMA.

allevia questo problema distribuendo la memoria su più moduli in modo che richieste simultanee da processori diversi accedano con più probabilità a moduli di memoria differenti, a seconda degli indirizzi delle richieste.

Una rete di interconnessione consente a qualsiasi processore di accedere a qualsiasi modulo che è parte della memoria condivisa. Quando i moduli di memoria sono tenuti fisicamente separati dai processori, tutte le richieste di accesso alla memoria devono passare attraverso la rete, che introduce una latenza. La Figura 11.2 mostra tale disposizione. Un sistema che ha la stessa latenza di rete per tutti gli accessi dai processori ai moduli di memoria viene detto multiprocessore *ad accesso uniforme alla memoria* o UMA (*Uniform Memory Access*). Sebbene la latenza sia uniforme, può essere grande per una rete che collega molti processori e moduli di memoria.

Per ottenere prestazioni migliori, conviene posizionare un modulo di memoria vicino a ciascun processore. Il risultato è un insieme di *nodi*, ciascuno costituito da un processore e un modulo di memoria. I nodi sono collegati alla rete, come mostrato in Figura 11.3. Si evita la latenza di rete quando un processore effettua una richiesta per accedere alla sua memoria locale. Tuttavia, una richiesta per accedere a un modulo di memoria remoto deve passare attraverso la rete. A causa della differenza di latenze per l'accesso a porzioni locali e remote della memoria condivisa, un sistema di questo tipo è detto multiprocessore *ad accesso non uniforme alla memoria* o NUMA (*Non-Uniform Memory Access*).

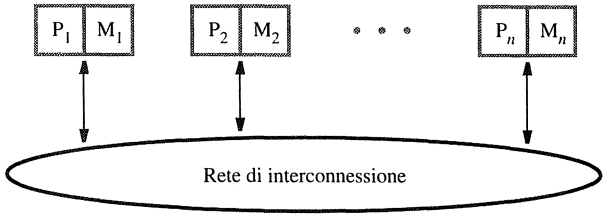


Figura 11.3 Un multiprocessore NUMA.

11.3.1 Reti di interconnessione

La rete di interconnessione deve consentire il trasferimento di informazioni fra qualsiasi coppia di nodi nel sistema. Si può usare la rete anche per inviare informazione in *broadcast*, da un nodo a molti altri nodi. Il traffico della rete è costituito da richieste (quali letture e scritture) e trasferimenti di dati.

L'idoneità di una particolare rete viene giudicata in termini di costi, larghezza di banda, throughput effettivo e facilità di realizzazione. Il termine *larghezza di banda* (*bandwidth*) si riferisce alla capacità di un collegamento di trasferire dati ed è espressa in bit o byte al secondo. Il *throughput effettivo* è il tasso effettivo di trasferimento dati. Questo tasso è inferiore alla larghezza di banda disponibile poiché un determinato collegamento deve anche sostenere informazione di controllo che coordina il trasferimento dei dati.

Il trasferimento delle informazioni attraverso la rete avviene solitamente sotto forma di *pacchetti* di lunghezza fissa e formato specificato. Per esempio, una richiesta di lettura è probabile che sia un singolo pacchetto inviato da un processore a un modulo di memoria. Il pacchetto contiene gli identificatori dei nodi sorgente e destinazione, l'indirizzo della locazione da leggere, e un campo per il comando che indica il tipo di operazione di lettura richiesta. Una richiesta di scrittura che scrive una parola in un modulo di memoria è anche probabile che sia un singolo pacchetto che include i dati da scrivere. D'altra parte, una risposta alla lettura può comportare un blocco intero di cache che richiede diversi pacchetti per il trasferimento dei dati.

Idealmente, un pacchetto completo sarà gestito in parallelo in un ciclo di clock in ogni nodo o commutatore (*switch*) della rete. Ciò implica avere collegamenti larghi, costituiti da molte linee. Tuttavia, per ridurre i costi e la complessità, i collegamenti sono spesso molto stretti. In tali casi, un pacchetto deve essere diviso in pezzi più piccoli, ciascuno dei quali può essere trasmesso in un ciclo di clock.

I paragrafi appresso descrivono alcune delle reti di interconnessione che sono comunemente usate nei multiprocessori.

Topologia a bus

Un *bus* è un insieme di linee (fili) che forniscono un unico percorso condiviso per il trasferimento di informazioni, come discusso nel Capitolo 7. I bus sono più comunemente utilizzati in multiprocessori UMA per collegare i vari processori ai moduli di memoria condivisa. È necessario un arbitraggio per far sì che a uno solo dei tanti possibili richiedenti sia concesso l'uso del bus in qualsiasi momento. Il bus è adatto per un numero relativamente piccolo di processori per via della contesa per l'accesso al bus e per l'aumento del ritardo di propagazione dovuto al carico elettrico quando sono collegati molti processori.

Un bus semplice non permette a una nuova richiesta di apparire sul bus finché non sia stata data risposta alla richiesta corrente. Tuttavia, se la latenza di risposta è elevata, vi può essere considerevole tempo di inattività sul bus.

Si possono ottenere prestazioni più alte usando un bus a *transazioni spezzate* (*split-transaction*), in cui una richiesta e la sua corrispondente risposta sono trattate come eventi separati, fra cui possono avere luogo altri trasferimenti. Si consideri una situazione in cui più processori hanno bisogno di fare richieste di lettura alla memoria. L'arbitraggio seleziona il primo processore a cui concedere l'uso del bus per la sua richiesta. Dopo che la richiesta viene fatta, si seleziona un secondo processore per fare la sua richiesta, invece di lasciare il bus inattivo. Supponendo che questa richiesta sia per un modulo di memoria diverso, i due accessi in lettura

procedono in parallelo. Se nessuno dei due moduli ha finito con il suo accesso, si seleziona un terzo processore per effettuare la sua richiesta e così via. Prima o poi un modulo di memoria completa il suo accesso in lettura: gli si concede l'uso del bus per trasferire i dati al processore richiedente. Quando gli altri moduli completano i loro accessi, si usa il bus per trasferire le loro risposte. La quantità effettiva di tempo fra ciascuna richiesta e la corrispondente risposta può variare con l'alternanza sul bus di richieste e risposte per transazioni differenti per un uso efficiente della larghezza di banda disponibile.

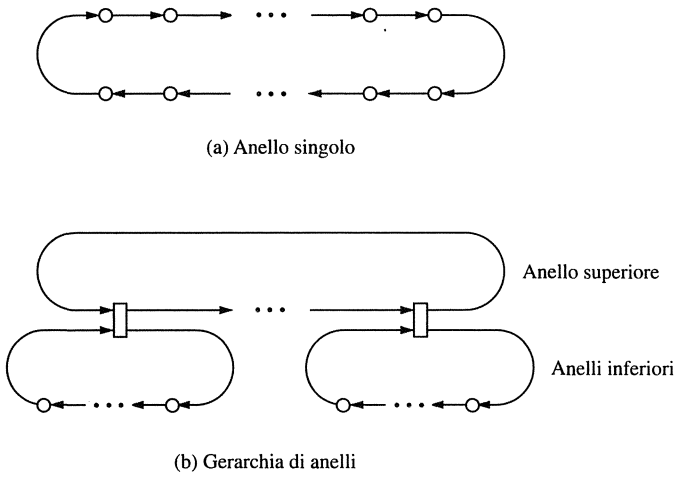
Il bus a transazioni spezzate richiede un protocollo più complesso. La principale fonte di complessità è la necessità di abbinare ciascuna risposta alla sua richiesta corrispondente. Questa necessità è di solito gestita associando un'etichetta (*tag*) unica a ciascuna richiesta che appare sul bus. Ogni risposta appare quindi con l'etichetta appropriata in modo che la fonte possa abbinarla alla richiesta originale.

Topologia ad anello

Una rete ad *anello* è formata da connessioni punto-punto fra nodi, come mostrato in Figura 11.4. La Figura 11.4a mostra un anello singolo. In un lungo anello singolo si ha un'alta latenza media per la comunicazione fra due nodi qualsiasi. Questa latenza elevata può essere attenuata in due modi diversi.

Si può aggiungere un secondo anello per connettere i nodi nella direzione opposta. Come risultato si ha un *anello bidirezionale* che dimezza la latenza media e raddoppia la larghezza di banda. Tuttavia, la gestione delle comunicazioni è più complessa.

Un altro approccio è quello di utilizzare una *gerarchia di anelli*. Una gerarchia a due livelli è illustrata nella Figura 11.4b. L'anello di livello superiore è collegato agli anelli di livello inferiore. Con questa sistemazione la latenza media per la comunicazione fra due nodi qualsiasi su ogni anello di livello inferiore si riduce. I trasferimenti fra nodi sullo stesso anello di livello inferiore non devono attraversare l'anello di livello superiore. I trasferimenti fra nodi su anelli di livello inferiore diversi comportano l'attraversamento di una parte dell'anello di livello superiore.



(a) Anello singolo

(b) Gerarchia di anelli

Figura 11.4 Reti di interconnessione basate su anelli.

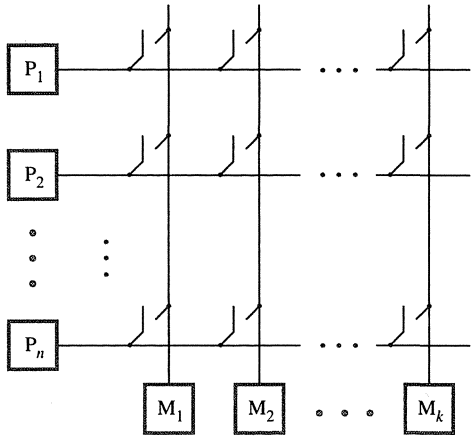


Figura 11.5 Rete di interconnessione crossbar.

Lo svantaggio del sistema gerarchico è che l'anello di livello superiore può diventare un collo di bottiglia quando molti nodi di anelli diversi di livello inferiore comunicano spesso fra loro.

Topologia crossbar

Una rete *crossbar* fornisce una connessione diretta fra ogni coppia di unità collegate. Si usa tipicamente in sistemi multiprocessore UMA per la connessione dei processori ai moduli di memoria. Essa permette molti trasferimenti simultanei se la stessa destinazione non è l'obiettivo di richieste multiple. Per esempio, si può realizzare la struttura in Figura 11.2 utilizzando una rete crossbar che consta di un insieme di commutatori come in Figura 11.5. Per n processori e k memorie occorrono $n \times k$ commutatori.

Topologia a maglia

Un modo naturale per collegare un gran numero di nodi è usare una maglia (*mesh*) bidimensionale, come mostrato in Figura 11.6. Ogni nodo interno della maglia ha quattro connessioni,

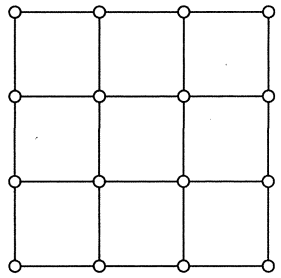


Figura 11.6 Una rete a maglia bidimensionale.

una per ciascuno dei propri vicini in orizzontale e in verticale. I nodi ai confini e agli angoli della maglia hanno meno vicini e quindi un minore numero di connessioni. Per ridurre la latenza di comunicazione fra nodi che sarebbero altrimenti distanti nella maglia, si possono inserire connessioni avvolgenti fra i nodi ai lati opposti della maglia. Una rete con tali collegamenti è detta *toroidale*. Tutti i nodi di un toro hanno quattro connessioni. La latenza media si riduce, ma la complessità di realizzazione per instradare richieste e risposte attraverso un toro è leggermente superiore rispetto al caso di una maglia semplice.

11.4 Coerenza di cache

Un multiprocessore a memoria condivisa è facile da programmare. Ogni variabile in un programma ha un unico indirizzo di locazione in memoria, a cui si accede da qualsiasi processore. Tuttavia, ogni processore ha la propria cache. Pertanto, è necessario affrontare la possibilità che copie di dati condivisi possano risiedere in varie cache. Quando un processore scrive una variabile condivisa nella propria cache, tutte le altre cache che contengono una copia di quella variabile avranno quindi il vecchio valore, errato. Esse vanno informate del cambiamento, sì che possano aggiornare la loro copia al nuovo valore o invalidarla. Questo è il problema di mantenere la *coerenza di cache*, che richiede una visione coerente dei dati condivisi in varie cache.

Nel Capitolo 8 si è discusso di due approcci di base per l'esecuzione di operazioni di scrittura di dati in una cache. L'approccio di scrittura immediata cambia i dati nella cache e nella memoria principale. L'approccio di scrittura differita cambia solo i dati nella cache; la copia della memoria principale viene aggiornata quando un blocco modificato di dati nella cache deve essere sostituito. Si possono usare approcci simili per affrontare la coerenza di cache in un sistema multiprocessore.

11.4.1 Protocollo di scrittura immediata

Un protocollo di scrittura immediata può essere realizzato in due modi. Una versione è basata sull'aggiornamento dei valori in altre cache, mentre la seconda si basa sul rendere invalide le copie in altre cache.

Si consideri per primo il protocollo di *aggiornamento*. Quando un processore scrive un nuovo valore su un blocco di dati nella sua cache, il nuovo valore viene scritto anche nel modulo di memoria contenente il blocco modificato. Poiché vi possono essere copie di questo blocco in altre cache, queste copie vanno aggiornate per riflettere le modifiche conseguenti all'operazione Write. Il modo più semplice per farlo è quello di trasmettere i dati scritti alle cache di tutti i processori del sistema. Quando ciascun processore riceve i dati trasmessi, esso aggiorna il contenuto del blocco di cache modificato se questo blocco è presente nella sua cache.

La seconda versione del protocollo di scrittura immediata è basata sull'*invalidazione* di copie. Quando un processore scrive un nuovo valore nella sua cache, questo viene inviato anche alla locazione di memoria appropriata e tutte le copie in altre cache vengono invalidate. Anche in questo caso, si può usare il broadcast per inviare la richiesta di invalidazione a tutto il sistema.

11.4.2 Protocollo di scrittura differita

Il mantenimento della coerenza con il protocollo di scrittura differita si basa sul concetto di proprietà di un blocco di dati nella memoria. Inizialmente, la memoria è il proprietario di tutti i blocchi; essa mantiene la proprietà di un blocco che è letto da un processore per porre una copia nella sua cache.

Se qualche processore vuole scrivere su un blocco nella sua cache, deve prima diventare il proprietario esclusivo di questo blocco. Per fare ciò, tutte le copie in altre cache devono prima essere invalidate con una richiesta trasmessa a tutte. Il nuovo proprietario del blocco può quindi modificare il contenuto a piacimento, senza dover fare altre azioni.

Quando un altro processore vuole leggere un blocco che è stato modificato, si deve trasmettere la richiesta di tale blocco al proprietario attuale. I dati vengono quindi inviati al processore richiedente dal proprietario corrente. I dati vengono anche inviati al modulo di memoria appropriato, che riacquista la proprietà e aggiorna il contenuto del blocco nella memoria. La cache del processore che era il precedente proprietario mantiene una copia del blocco. Quindi, il blocco è adesso condiviso con copie in due cache e nella memoria. Le richieste successive di lettura dello stesso blocco da parte di altri processori vengono servite dal modulo di memoria che contiene il blocco.

Quando un altro processore intende scrivere su un blocco che è stato modificato, l'attuale proprietario invia i dati al processore richiedente. Esso inoltre trasferisce la proprietà del blocco al processore richiedente e invalida la sua copia nella cache. Poiché il blocco viene modificato dal nuovo proprietario, il contenuto del blocco nella memoria non viene aggiornato. La richiesta successiva per lo stesso blocco è servita dal nuovo proprietario.

Il protocollo di scrittura differita ha il vantaggio di creare meno traffico rispetto al protocollo di scrittura immediata. Questo perché un processore probabilmente esegue varie scritture su un blocco della cache prima che questo blocco sia utile a un altro processore. Con il protocollo di scrittura differita, queste scritture sono eseguite solo nella cache, una volta che la proprietà sia acquisita con una richiesta di invalidazione. Con il protocollo di scrittura immediata, ogni scrittura deve essere eseguita anche nel modulo di memoria appropriato e deve essere trasmessa a tutte le altre cache.

Finora si è assunto che le richieste di aggiornamento e invalidazione in questi protocolli vengano trasmesse attraverso la rete di interconnessione. Quanto sia facile realizzare tali trasmissioni dipende in gran parte dalla struttura della rete di interconnessione. Quella più naturale per il supporto del broadcast è il bus singolo. Nei multiprocessori che collegano un numero modesto di processori ai moduli di memoria con un singolo bus, la coerenza di cache può essere realizzata usando uno schema detto *snooping* (spionaggio).

11.4.3 Cache snoopy

In un sistema con bus unico, tutte le transazioni fra processori e moduli di memoria avvengono attraverso richieste e risposte sul bus. In effetti, esse sono trasmesse a tutte le unità collegate al bus. Si supponga che ogni cache del processore abbia un circuito di controllo che osserva, o *spia* (*snoop*), tutte le transazioni sul bus. Si descrivono ora alcuni scenari per il protocollo di scrittura differita e come si ottiene la coerenza di cache.

Si consideri un processore che ha già letto una copia di un blocco dalla memoria nella sua cache. Prima di scrivere su questo blocco per la prima volta, il processore deve trasmettere una *richiesta di invalidazione* a tutte le altre cache, i controllori di queste accettano la richiesta e invalidano le copie dello stesso blocco. Questa azione fa sì che il processore richiedente diventi il nuovo proprietario del blocco. Il processore può quindi scrivere il blocco e marcarlo come in corso di modifica. Non sono necessarie ulteriori trasmissioni dallo stesso processore per scritture sul blocco modificato nella sua cache.

Adesso, se un altro processore trasmette una *richiesta di lettura* sul bus per lo stesso blocco, la memoria non deve rispondere poiché non è il proprietario attuale del blocco. Il

processore proprietario del blocco richiesto spia la richiesta di lettura sul bus. Poiché ha una copia modificata del blocco richiesto nella sua cache, asserisce un segnale speciale sul bus per evitare che la memoria risponda. Il proprietario trasmette poi una copia del blocco sul bus e marca la sua copia come pulita (non modificata). La risposta con i dati sul bus è accettata dalla cache del processore che ha emesso la richiesta di lettura. La risposta con i dati è accettata anche dalla memoria per aggiornare la propria copia del blocco. In questo caso, la memoria riacquista la proprietà del blocco: si dice che il blocco è in uno stato condiviso, poiché copie di esso sono nelle cache di due processori. La coerenza è mantenuta perché le due copie nelle cache e la copia del blocco nella memoria contengono gli stessi dati. Le richieste successive da qualsiasi processore sono servite dalla memoria.

Si consideri ora la situazione in cui due processori hanno copie dello stesso blocco nelle rispettive cache, ed entrambi i processori tentano di scrivere sullo stesso blocco nella cache contemporaneamente. Poiché il blocco è nello stato condiviso, la memoria è il proprietario del blocco. Quindi, entrambi i processori richiedono l'uso del bus per inviare un messaggio di invalidazione. A uno dei processori è concesso l'uso del bus per primo. Questo processore trasmette la sua richiesta di invalidazione e diventa il nuovo proprietario del blocco. Attraverso lo snooping del bus, la copia del blocco nella cache dell'altro processore viene invalidata. Quando all'altro processore è poi concesso l'uso del bus, esso trasmette una *richiesta di lettura esclusiva*, che combina una richiesta di lettura e una richiesta di invalidazione per lo stesso blocco. Il controllore per il primo processore spia la richiesta di lettura esclusiva, fornisce una risposta dati sul bus e invalida la copia nella sua cache. La proprietà del blocco è quindi trasferita al secondo processore che effettua la richiesta. La memoria non viene aggiornata in quanto il blocco viene modificato di nuovo. Poiché le richieste provenienti dai due processori vengono trattate in sequenza, la coerenza di cache viene mantenuta in ogni momento.

Lo schema appena descritto è basato sulla capacità dei controllori delle cache di spiare l'attività sul bus e fare le azioni appropriate. Tali schemi sono detti *tecniche di cache snoopy*.

Per motivi di prestazione, è importante che lo snooping non interferisca con il normale funzionamento di un processore e della sua cache. Si ha tale interferenza se il controllore della cache accede alle etichette della cache per ogni richiesta che appare sul bus. Nella maggior parte dei casi, la cache non conterrà una copia valida del blocco che è rilevante per una richiesta. Per eliminare interferenze inutili, ciascuna cache può essere provvista di una serie di etichette duplicate, che mantengono le stesse informazioni sullo stato dei blocchi nella cache, ma accessibili separatamente dai circuiti di snooping.

11.4.4 Coerenza di cache basata su directory

Il concetto di cache snoopy è facile da realizzare nei sistemi a singolo bus. Grandi sistemi multiprocessori a memoria condivisa utilizzano reti di interconnessione quali anelli e maglie. In tali sistemi, trasmettere ogni singola richiesta alle cache di tutti i processori è inefficiente. Una soluzione scalabile, ma più complessa, usa una *directory* (cioè un elenco) in ciascun modulo di memoria per indicare quali nodi possono avere copie di un dato blocco nello stato condiviso. Se un blocco viene modificato, la directory identifica il nodo che ne è l'attuale proprietario. Ogni richiesta da un processore deve essere prima inviata al modulo di memoria contenente il blocco in questione. Le informazioni di directory per quel blocco vengono usate per determinare l'azione da intraprendere. Una richiesta di lettura viene inoltrata al proprietario corrente se il blocco viene modificato. Nel caso di una richiesta di scrittura di un blocco condiviso, si inviano invalidazioni individuali solo ai nodi che possono avere copie del blocco in questione. Il costo e la complessità dell'approccio basato su directory per imporre la coerenza di cache ne limita l'uso ai sistemi di grandi dimensioni. Sistemi multiprocessori piccoli, compresi gli attuali chip multicore, utilizzano in genere lo snooping.

11.5 Multicalcolatori a scambio di messaggi

Un diverso modo di utilizzare più processori implica la realizzazione di ciascun nodo del sistema come un calcolatore completo con la sua memoria. Gli altri calcolatori nel sistema non hanno accesso diretto a questa memoria. I dati che devono essere condivisi vengono scambiati inviando messaggi da un calcolatore all'altro. Tali sistemi sono detti *multicalcolatori a scambio di messaggi* (*message-passing multicomputer*).

I programmi paralleli sono scritti in modo diverso per i multicalcolatori a scambio di messaggi che per multiprocessori a memoria condivisa. Per condividere dati fra i nodi, il programma in esecuzione sul calcolatore che è la fonte dei dati deve inviare un messaggio contenente i dati al calcolatore di destinazione. Il programma in esecuzione nel calcolatore di destinazione riceve il messaggio e copia i dati nella memoria di quel nodo.

Per facilitare lo scambio di messaggi, una unità speciale di comunicazione a ogni nodo è spesso responsabile dei dettagli di basso livello di formattazione e interpretazione dei messaggi che vengono inviati e ricevuti e delle operazioni di copia dei dati dei messaggi da e verso la memoria del nodo. Il calcolatore in ciascun nodo emette comandi all'unità di comunicazione. Il calcolatore continua quindi a eseguire altri calcoli, mentre l'unità di comunicazione gestisce i dettagli di invio e ricezione di messaggi.

11.6 Programmazione parallela per multiprocessori

I paragrafi precedenti hanno descritto sistemazioni dell'hardware di multiprocessori a memoria condivisa in grado di sfruttare il parallelismo di programmi applicativi. Il parallelismo disponibile può essere trovato in cicli con passate indipendenti, e anche in attività di livello superiore indipendenti.

Un programma sorgente scritto in un linguaggio di alto livello consente a un programmatore di esprimere l'elaborazione desiderata in un modo che sia di facile comprensione. Esso deve essere tradotto dal compilatore e dall'assemblatore nella rappresentazione in linguaggio macchina. L'hardware del processore è progettato per eseguire istruzioni in linguaggio macchina nella sequenza appropriata per svolgere l'elaborazione desiderata dal programmatore. Non può identificare automaticamente attività di alto livello indipendenti che potrebbero essere eseguite in parallelo. Anche il compilatore ha dei limiti nel rilevare e sfruttare il parallelismo. È pertanto responsabilità del programmatore partizionare in modo esplicito l'elaborazione complessiva nel programma sorgente in attività e specificare il modo in cui devono essere eseguite su più processori.

La programmazione per un multiprocessore a memoria condivisa è un'estensione naturale della programmazione convenzionale per un singolo processore. Si scrive un programma sorgente ad alto livello usando attività eseguite da un processore. Tuttavia si può anche indicare che alcune attività devono essere eseguite simultaneamente in diversi processori. La condivisione dei dati si ottiene mediante la definizione di variabili globali, lette e scritte da diversi processori nell'esecuzione dei compiti loro assegnati. I chip multicore attualmente utilizzati in calcolatori di uso generale, come quelli che realizzano l'architettura Intel IA-32, sono programmati in questo modo.

```
#include   <stdio.h>        /* Routine per ingresso/uscita. */
#define    N  100           /* Numero di elementi in ciascun vettore. */
double     a[N], b[N];      /* Vettori di cui calcolare il prodotto scalare. */
void       main (void)
{
           int i;
           double prod_scalare;

           <Inizializza i vettori a[], b[] (dettagli omessi).>
           dot_product = 0.0;
           for (i = 0; i < N; i++)
              prod_scalare = prod_scalare + a[i]*b[i];
           printf("Il prodotto scalare è %g\n", prod_scalare);
}
```

Figura 11.7 Programma C per il calcolo di un prodotto scalare.

Per illustrare la programmazione parallela, si consideri l'esempio del calcolo del prodotto scalare di due vettori, ognuno contenente N numeri. Un programma in linguaggio C per questo compito è mostrato in Figura 11.7. I dettagli di inizializzazione del contenuto dei due vettori sono omessi per focalizzare gli aspetti rilevanti alla programmazione parallela.

Il ciclo accumula la somma di N prodotti. Ogni passata dipende dalla somma parziale calcolata nella passata precedente, e il risultato calcolato nella passata finale è il prodotto scalare. Nonostante la dipendenza, si può suddividere il programma in attività indipendenti per l'esecuzione simultanea, sfruttando la proprietà associativa dell'addizione. Ogni attività calcola una somma parziale e il risultato finale si ottiene addizionando le somme parziali.

Per realizzare un programma parallelo per il calcolo del prodotto scalare bisogna rispondere a due domande.

1. Come si può fare in modo che più processori partecipino all'esecuzione parallela per calcolare le somme parziali?
2. Come si può garantire che ogni processore abbia calcolato la sua somma parziale prima che sia calcolato il risultato finale per il prodotto scalare?

Creazione di thread

Per rispondere alla prima domanda, si definiscono le attività che vengono assegnate ai diversi processori, e poi si descrive come si avvia l'esecuzione di queste attività in più processori. È possibile scrivere una versione parallela del programma per il prodotto scalare usando come parametri il numero di processori, P, e il numero di elementi in ciascun vettore, N. Si assuma per semplicità che N sia divisibile per P. Il calcolo complessivo comporta una somma di N prodotti. Per P processori si definiscono P attività indipendenti, dove ogni attività è il calcolo di una somma parziale di N/P prodotti.

Quando si esegue un programma in un singolo processore, vi è un thread attivo di controllo dell'esecuzione. Questo thread viene creato implicitamente dal sistema operativo (SO) quando inizia l'esecuzione del programma. Per un programma parallelo, si richiede che attività indipendenti siano trattate separatamente da più thread di controllo dell'esecuzione, uno per ogni processore. Questi thread devono essere creati in modo esplicito. Un tipico approccio è usare una routine chiamata *create_thread* in una libreria di supporto alla programmazione parallela. La routine di libreria accetta un parametro di ingresso, che è un puntatore a un sottoprogramma che deve essere eseguito dal thread appena creato. Un servizio del sistema operativo viene

richiamato dalla routine di libreria per creare un nuovo thread con una pila distinta, in modo che possa chiamare altri sottoprogrammi e avere le sue variabili locali. Tutte le variabili globali sono condivise fra tutti i thread.

È necessario distinguere i thread fra loro. Un approccio è quello di fornire un'altra routine di libreria chiamata *get_my_thread_id* che restituisce un numero intero univoco compreso tra 0 e $P - 1$ per ogni thread. Con queste informazioni, un thread può determinare il sottoinsieme appropriato del calcolo complessivo di cui è responsabile.

Sincronizzazione di thread

Per la seconda domanda bisogna determinare quando i thread hanno completato le loro attività, sì da poter calcolare correttamente il risultato finale. È quindi necessaria la *sincronizzazione* di più thread. Vi sono diversi metodi di sincronizzazione, spesso realizzati in routine di libreria aggiuntive per la programmazione parallela. Qui si considera un metodo detto *barriera*.

Lo scopo di una barriera è quello di costringere i thread ad attendere fino a quando non abbiano tutti raggiunto un punto specifico del programma in cui si effettua una chiamata alla routine di libreria per la barriera. Ogni thread che chiama la routine per la barriera entra in un ciclo di attesa attiva fino a quando l'ultimo thread chiama la routine e permette a tutti i thread di continuare la loro esecuzione. Questo assicura che i thread abbiano completato i loro rispettivi calcoli che precedono la chiamata alla barriera.

Esempio di programma parallelo

Dopo aver descritto le questioni connesse alla creazione di thread e alla loro sincronizzazione, nonché tipiche routine di libreria disponibili per la gestione dei thread, è possibile ora presentare come esempio un programma parallelo per il prodotto scalare. La Figura 11.8 mostra una routine principale, *main*, e un'altra routine chiamata *ParallelFunction* che definisce i compiti indipendenti per l'esecuzione parallela. Quando il programma inizia l'esecuzione, vi è un solo thread che esegue la routine *main*. Questo thread inizializza i vettori, quindi inizializza una variabile condivisa necessaria per la barriera di sincronizzazione. Per avviare l'esecuzione parallela, la routine *main* chiama la routine *create_thread* $P - 1$ volte per creare thread aggiuntivi, ciascuno dei quali esegue *ParallelFunction*. Quindi, il thread che esegue la routine *main* chiama *ParallelFunction* direttamente, sicché un totale di P thread sono in gioco nel calcolo complessivo. Il software del sistema operativo è responsabile per la distribuzione dei thread ai diversi processori per l'esecuzione parallela.

Ogni thread chiama *get_my_thread_id* da *ParallelFunction* per ottenere un identificatore intero univoco nell'intervallo da 0 a $P - 1$. Usando questa informazione, il thread calcola gli indici *iniziale* e *finale* per il ciclo che genera la somma parziale di questo thread. Dopo l'esecuzione del ciclo, scrive il risultato in un elemento distinto della matrice condivisa *partial_sums* usando il suo identificatore unico come indice della matrice. Quindi, il thread chiama la routine di libreria per la barriera di sincronizzazione e attende che altri thread completino il loro calcolo.

Dopo che l'ultimo thread chiama la routine di barriera a conclusione del suo calcolo, tutti i thread rientrano da *ParallelFunction*. Non c'è nessun altro calcolo da fare in *ParallelFunction*, così i $P - 1$ thread creati dalla chiamata di libreria nella routine *main* terminano. Il thread che ha chiamato *ParallelFunction* direttamente dalla routine *main* rientra per calcolare il risultato finale usando i valori nella matrice *partial_sums*.

Il programma in Figura 11.8 usa routine di libreria generiche per illustrare la creazione di thread e la sincronizzazione. Un'ampia raccolta di routine per la programmazione parallela in linguaggio C è definita nello standard IEEE 1003.1 [4]. Questa collezione è anche nota come libreria *POSIX threads* o *Pthreads*. Essa fornisce una varietà di meccanismi di gestione e sincronizzazione di thread. Realizzazioni di questa libreria sono disponibili per sistemi operativi in largo uso per facilitare la programmazione per multiprocessori.

```
#include   <stdio.h>          /* Routine per ingresso/uscita. */
#include   "threads.h"        /* Routine per creazione e sincronizzazione di thread. */

#define    N  100             /* Numero di elementi in ciascun vettore. */
#define    P  4               /* Numero di processori per l'esecuzione parallela. */

double     a[N], b[N];        /* Vettori di cui calcolare il prodotto scalare. */
double     somme_parziali[P]; /* Matrice di risultati calcolati dai thread. */
Barrier    bar;               /* Variabile condivisa a supporto della barriera di sincronizzazione. */

void       ParallelFunction (void)
{
           int mio_id, i, inizio, fine;
           double s;

           mio_id = get_my_thread_id(); /* Ottieni l'indentificatore unico di questo thread. */
           inizio = (N/P) * mio_id;     /* Determina inizio e fine usando l'identificatore del
                                           thread. */
           fine = (N/P) * (mio_id+1) - 1; /* Si assume che N sia divisibile per P. */
           s = 0.0;
           for (i=inizio; i<=fine; i++)
             s = s + a[i] * b[i];
           somme_parziali[mio_id] = s;  /* Salva il risultato nella matrice. */
           barrier (&bar, P);           /* Sincronizza con gli altri thread. */
}

void       main (void)
{
           int i;
           double prod_scalare;

           <Inizializza i vettori a[], b[] (dettagli omessi).>
           init_barrier (&bar);
           for (i = 1; i < P; i++)      /* Crea P-1 thread addizionali. */
             create_thread (ParallelFunction);
           ParallelFunction();          /* Il thread main si aggiunge all'esecuzione
                                           parallela. */
           prod_scalare = 0.0;          /* Dopo la sincronizzazione alla barriera, calcola
                                           il risultato finale. */
           for (i = 0; i < P; i++)
             prod_scalare = prod_scalare + somme_parziali[i];
           printf("Il prodotto scalare è %g\n", prod_scalare);
}
```

Figura 11.8 Programma parallelo in C per il calcolo di un prodotto scalare.

11.7 Modellazione della prestazione

La misura più importante della prestazione di un calcolatore è quanto rapidamente può eseguire i programmi. Quando si considera un processore, la velocità con cui vengono prelevate ed eseguite le istruzioni è influenzata dall'architettura dell'insieme di istruzioni e dalla progettazione hardware. Il numero totale di istruzioni eseguite è influenzato dal compilatore,

nonché dall'architettura dell'insieme di istruzioni. Il Capitolo 6 ha introdotto l'equazione di base della prestazione, che è un modello matematico di basso livello che tiene conto di tali considerazioni per un processore. I termini di quel modello includono il numero di istruzioni eseguite, il numero medio di cicli per istruzione e la frequenza di clock. Questo modello consente la previsione di tempi di esecuzione, una volta fornite informazioni sufficientemente dettagliate.

Si può usare un modello di livello più alto, che si basa su informazioni meno dettagliate, per valutare miglioramenti potenziali della prestazione. Si consideri un programma il cui tempo di esecuzione su un certo calcolatore è T. L'obiettivo è di valutare in che misura si possa ridurre il tempo di esecuzione quando si introduce un miglioramento delle prestazioni, quale l'elaborazione parallela. Si supponga che una frazione f_e del tempo di esecuzione sia influenzata dal miglioramento. La frazione rimanente, $f_u = 1 - f_e$, è immutata. Si assuma che p rappresenti il fattore per il quale la porzione di tempo $f_e \times T$ si riduce a causa del miglioramento delle prestazioni. Il nuovo tempo di esecuzione è:

$$T_p = T(f_u + f_e/p)$$

Il *guadagno di prestazione* (*speedup*) è il rapporto T/T_p o:

$$1/(f_u + f_e/p)$$

L'espressione qui sopra per il guadagno di prestazione è nota come *Legge di Amdahl*. Prende il nome da Gene Amdahl, che fu il primo a formalizzare il ragionamento esposto. È un modo di asserire l'osservazione intuitiva che il beneficio di un certo miglioramento delle prestazioni aumenta se esso riguarda una porzione maggiore del tempo di esecuzione.

Una volta determinata una ripartizione del tempo di esecuzione originale, è spesso utile determinare un limite superiore del guadagno di prestazione possibile. Per farlo, sia $p \to \infty$, per riflettere l'ideale, ma irrealistica, riduzione a zero del tempo di esecuzione dovuto alla frazione f_e. Il guadagno di prestazione risultante è $1/f_u$, il che significa che la porzione del tempo di esecuzione che non migliora è il fattore che limita le prestazioni. Un valore più piccolo di f_u dà un limite più grande sul guadagno di prestazione. Per esempio, $f_u = 0{,}1$ fornisce un limite superiore di 10, ma $f_u = 0{,}05$ fornisce una limite superiore di 20. Tuttavia, il previsto guadagno di prestazione con un valore realistico di p è generalmente ben al di sotto del limite superiore. Per esempio, utilizzando $p = 16$ con $f_u = 0{,}05$ si ha un guadagno di prestazione di solo $1/(0{,}05 + 0{,}95/16) = 9{,}1$, ben al di sotto del limite superiore di 20.

La conclusione importante di questa discussione è che la porzione non migliorata del tempo di esecuzione originale può limitare significativamente il guadagno di prestazione ottenibile, anche se la porzione migliorabile è migliorata da un fattore arbitrariamente grande. Se un programmatore può determinare, anche approssimativamente, le frazioni f_u e f_e di tempo di esecuzione prima di applicare un particolare miglioramento, la Legge di Amdahl può fornire indicazioni utili del miglioramento atteso. Tale informazione può essere usata per determinare se il previsto guadagno in prestazione giustifichi lo sforzo e la spesa per attuare il miglioramento.

Sommario

Tecniche fondamentali quali il pipelining e le cache sono strumenti importanti per migliorare le prestazioni e sono ampiamente utilizzate nei calcolatori. Le ulteriori tecniche di multithreading, elaborazione vettoriale (SIMD) e multiprocessing forniscono il potenziale per ulteriori miglioramenti di prestazione facendo un uso più efficiente delle risorse di elaborazione ed effettuando più operazioni in parallelo. Queste tecniche sono state integrate nei chip dei processori multicore di uso generale.

Note bibliografiche

1. NVIDIA Corporation, *NVIDIA C CUDA Programming Guide*, Version 3.1.1, July 2010, disponibile su http://www.nvidia.com.
2. Kirk D., Hwu W.M., *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kauffmann, Burlington, MA, USA 2010.
3. Khronos Group, *OpenCL Introduction and Overview*, June 2010, disponibile su http://www.khronos.org/opencl.
4. Institute of Electrical and Electronics Engineers, *1003.1 Standard for Information Technology-Portable Operating System Interface (POSIX): System Interfaces, Issue 6*, IEEE Standard 1003.1, 2001, disponibile su http://www.ieee.org.

C1 Progetto di un sistema su un chip

In questo caso di studio

- Progettazione di un sistema per la realizzazione su un chip FPGA
- Come usare strumenti CAD
- Come usare moduli parametrici
- Che cosa considerare tipicamente nella progettazione

Nel Capitolo 10 si è discusso l'uso di microcontrollori nei sistemi embedded. In un'applicazione embedded, è desiderabile usare quanti meno chip sia possibile. Idealmente, un singolo chip potrebbe realizzare l'intero sistema. Il termine *sistema su un singolo chip* (SOC) è in uso per designare questa tecnologia. In applicazioni semplici, alcuni dei microcontrollori disponibili in commercio possono realizzare tutte le funzioni necessarie. Questo è improbabile per applicazioni più complesse.

La progettazione di un microcontrollore per un'applicazione embedded complessa e la sua realizzazione sotto forma di un singolo chip specifico è un obiettivo ambizioso e costoso. È anche molto dispendioso in termini di tempo. Eppure, il tempo di sviluppo per la maggior parte dei prodotti di consumo deve essere breve. Un chip che realizzi un intero sistema per una certa applicazione si può sviluppare in un tempo molto più breve se il progettista ha accesso a moduli con circuiti predefiniti che siano disponibili in una forma facile da usare. Il circuito di un processore è uno dei moduli necessari. Tali circuiti sono detti *nuclei-processore* (*processor core*) nella letteratura tecnica. Si può ottenere una varietà di nuclei-processore, mediante accordi per le licenze, da un certo numero di aziende. Si possono ottenere altri moduli per realizzare memoria, interfacce di ingresso/uscita, circuiti di conversione A/D e D/A o circuiti di elaborazione digitale di segnale, DSP (*Digital Signal Processing*). Lo sviluppatore di un sistema completa quindi il progetto utilizzando i moduli disponibili e progettando il resto dei circuiti specifici per l'applicazione.

I fornitori di nuclei-processore e di altri moduli vendono i progetti piuttosto che i chip. Essi forniscono la *proprietà intellettuale*, IP (*Intellectual Property*), che altri possono usare per progettare i loro chip. Per facilitare lo sviluppo di prodotti basati su IP, sono disponibili una varietà di strumenti di assistenza alla progettazione, CAD (*Computer-Aided Design*).

Il costo di realizzazione di un chip progettato in modo personalizzato è un fattore importante. La fabbricazione di tali chip è costosa e, sebbene offrano prestazioni migliori e consumi ridotti, il loro costo può essere giustificato solo se sono necessari un gran numero di chip per applicazioni specifiche. Una possibilità alternativa è l'uso di tecnologia FPGA (*Field Programmable Gate Array*), ovvero di *matrici di porte logiche programmabili sul campo*.

Il Caso di studio completo è disponibile sul sito www.ateneonline.it/hamacher3e

C2 Architettura Intel IA-32

In questo caso di studio

- Organizzazione della memoria e struttura dei registri IA-32
- Registri e istruzioni per operazioni in virgola mobile
- Confronto con le istruzioni in virgola mobile Freescale ColdFire
- Istruzioni dell'estensione multimediale (MMX) dell'architettura
- Istruzioni vettoriali (SIMD) su operandi in virgola mobile

Il termine *Intel Architecture* (IA) è usato per designare tutti i processori prodotti da Intel Corporation. In questo caso di studio vengono descritte con un certo dettaglio alcune caratteristiche avanzate dell'architettura e del linguaggio macchina della famiglia di processori IA capaci di operare con indirizzi di memoria e dati a 32 bit; essi sono chiamati genericamente processori IA-32. L'attenzione verrà innanzitutto rivolta a richiamare e completare la presentazione di caratteristiche di base dell'architettura (quali l'organizzazione della memoria e la struttura dei registri) appena accennata nell'Approfondimento applicativo 2.1. Dando poi per acquisita la panoramica sul repertorio delle istruzioni di base IA-32, offerta nei successivi approfondimenti applicativi dei Capitoli 2 e 3, si presentano ulteriori caratteristiche dell'architettura e dell'insieme di istruzioni IA-32 atte a realizzare concetti e modi di esecuzione introdotti nei Capitoli 9 e 11, ovvero rispettivamente l'esecuzione di operazioni in virgola mobile e l'elaborazione vettoriale SIMD. Quest'ultima capacità di elaborazione è stata introdotta nei processori Intel per migliorarne le prestazioni nell'elaborazione di dati multimediali ed è fortemente orientata a tale finalità, per la quale sono state progettate apposite unità di elaborazione ed estensioni del repertorio delle istruzioni. Anche l'altra famiglia di processori CISC considerata negli approfondimenti applicativi, Freescale ColdFire, include un'unità di elaborazione in virgola mobile e relative istruzioni; questo offre l'opportunità di raffrontarne similarità e differenze, che vengono illustrate da un esempio di programma assemblativo per calcoli in virgola mobile sviluppato in entrambi i linguaggi macchina.

Informazioni dettagliate sull'organizzazione dell'insieme di istruzioni dei processori IA-32 e sul linguaggio macchina relativo sono disponibili sul sito ufficiale Intel (www.intel.com), dove peraltro la documentazione più aggiornata unifica le caratteristiche di questa famiglia a quelle della sua successiva evoluzione conservativa, designata IA-64, che comprende i processori IA capaci di operare con indirizzi di memoria e dati a 64 bit.

Il Caso di studio completo è disponibile sul sito www.ateneonline.it/hamacher3e

A

Elementi di logica della commutazione

In questa appendice

Questa appendice fornisce una presentazione concisa dei circuiti logici. Si apprenderanno:

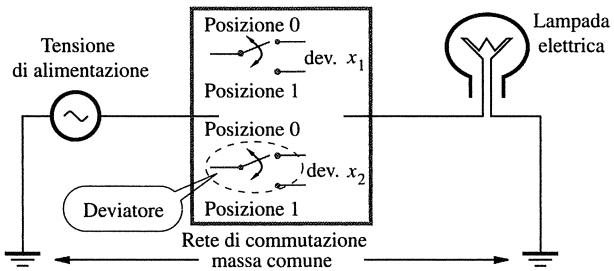
- Sintesi di funzioni logiche
- Flip-flop, registri e contatori
- Decodificatori e multiplatori
- Dispositivi logici programmabili
- Realizzazione pratica di circuiti logici

Nel calcolatore elettronico l'informazione è rappresentata mediante *segnali binari* e viene calcolata tramite *circuiti digitali* o *logici* (*digital circuit* o *network*), una famiglia particolare di circuiti elettronici. Il circuito digitale lavora su segnali binari che possono assumere due soli valori differenti di solito denotati con i simboli 0 e 1 (falso e vero). Nell'appendice si darà una presentazione sintetica delle funzioni logiche fondamentali e dei circuiti digitali per calcolarle, senza trascurare qualche nozione essenziale intorno alla tecnologia del circuito integrato. Per approfondimenti tecnologici sui circuiti integrati e per una trattazione sistematica di analisi e sintesi dei circuiti digitali si rimanda alla bibliografia [1, 2, 3, 4].

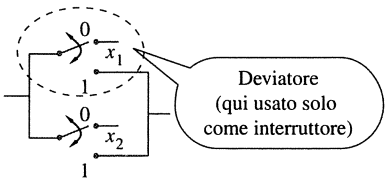
A.1 Funzioni logiche fondamentali

Conviene introdurre la logica binaria prendendo in esame un problema pratico che si presenta in qualsiasi abitazione domestica. Si supponga di volere controllare una lampada elettrica tramite due deviatori x_1 e x_2. Il deviatore ha due posizioni di funzionamento denotate con 0 e 1, rispettivamente, come mostra la Figura A.1a. Si rappresenta efficacemente lo stato del deviatore associandogli una variabile binaria, che può assumere solo i due valori 0 e 1. Nel seguito il nome del deviatore indicherà anche la variabile binaria associata, dunque $x_1, x_2 \in \{0, 1\}$. Per accendere la lampada occorre collegarne i morsetti ad alimentazione elettrica e massa. Lo schema di collegamento tra i morsetti dei deviatori e della lampada determina il modo di controllare lo stato di accensione di quest'ultima. La lampada è accesa se e solo se passando attraverso uno o più deviatori esiste almeno un percorso continuo tra alimentazione e morsetto della lampada stessa. Ora si immagini di porre $f = 1$ o $f = 0$ se la lampada è accesa o spenta, rispettivamente. Per quanto appena detto se $f = 1$ c'è almeno un percorso continuo passante per i deviatori, mentre se $f = 0$ tale percorso non esiste. Si può dunque interpretare il simbolo f come se fosse una funzione delle due variabili binarie x_1 e x_2, vale a dire $f(x_1, x_2)$, che assume valore nel codominio $\{0, 1\}$.

Si considerino ora tre modi differenti di controllare accensione e spegnimento della lampada. Come prima possibilità, si supponga di stabilire che la lampada debba essere accesa se

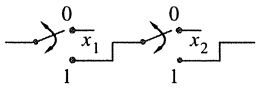


(a) Lampada elettrica controllata tramite due deviatori



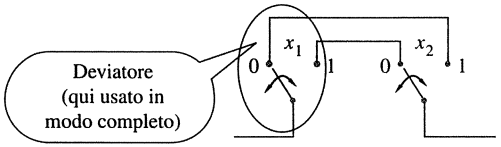
x_1	x_2	$f(x_1, x_2) = x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

(b) Deviatori collegati in parallelo (schema OR o somma logica)



x_1	x_2	$f(x_1, x_2) = x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

(c) Deviatori collegati in serie (schema AND o prodotto logico)



x_1	x_2	$f(x_1, x_2) = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

(d) Schema di collegamento XOR (differenza simmetrica)

Figura A.1 Esempio di rete di commutazione per controllo di lampada elettrica.

almeno un deviatore è in posizione 1. Si deve dunque avere $f = 1$ se una delle tre condizioni seguenti, mutuamente esclusive, è verificata:

$$x_1 = 1 \quad \text{e} \quad x_2 = 0$$

oppure:

$$x_1 = 0 \quad \text{e} \quad x_2 = 1$$

oppure:

$$x_1 = 1 \quad \text{e} \quad x_2 = 1$$

La Figura A.1b mostra come collegare i due deviatori per ottenere il comportamento richiesto. Entrambi hanno un morsetto laterale lasciato scollegato e pertanto in realtà sono usati come interruttori. A lato la situazione complessiva è riassunta tramite la *tabella di verità* (*truth table*): essa elenca tutte le combinazioni possibili di valore delle due variabili x_1 e x_2 (ovvero $2^2 = 4$ combinazioni in totale) e per ciascuna combinazione dà il valore corrispondente della funzione f; per comodità le combinazioni di valore sono numerate a lato come se fossero numeri naturali a due bit. Nel linguaggio della logica la tabella rappresenta la funzione di "somma logica" o OR delle due variabili binarie x_1 e x_2. In simbologia algebrica, precisamente dell'algebra di Boole, la somma logica si denota tramite l'operatore binario (cioè a due argomenti) "+" oppure "$\vee$". La forma algebrica della funzione f è dunque la seguente:

$$f(x_1, x_2) = x_1 + x_2 = x_1 \vee x_2$$

Si dice che x_1 e x_2 sono le *variabili di ingresso* e f è il *valore di uscita* della funzione. Nel seguito in genere si eviterà di elencare gli argomenti della funzione logica quando sono palesi, tranne che per metterli volutamente in evidenza.

Occorre dare subito alcune proprietà fondamentali dell'operazione di somma logica. Essa gode della proprietà commutativa espressa dall'identità seguente:

$$x_1 + x_2 = x_2 + x_1$$

e gode anche della proprietà associativa espressa dall'identità seguente:

$$x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3$$

pertanto si può estendere a $n \geq 2$ variabili come segue:

$$f = x_1 + x_2 + \ldots + x_n$$

La funzione f assume valore 1 solo se almeno una (o anche più d'una) variabile x_i ($1 \leq i \leq n$) ha valore 1. Essa modella il comportamento della schiera di n deviatori collegati in parallelo; la Figura A.1b mostra appunto il caso $n = 2$. Esaminando la tabella di verità si vede che inoltre valgono le due identità seguenti:

$$1 + x = 1$$

e:

$$0 + x = x$$

Ora si supponga di stabilire che la lampada debba essere accesa solo se entrambi i deviatori sono in posizione 1. La Figura A.1c mostra come collegare i deviatori per ottenere tale comportamento; a lato c'è la tabella di verità relativa. Si tratta della funzione di "prodotto logico" o AND, che algebricamente si denota con l'operatore binario "·" oppure "$\wedge$", nel modo seguente:

$$f(x_1, x_2) = x_1 \cdot x_2 = x_1 \wedge x_2$$

Come di consueto se non c'è pericolo di confusione si omette l'operatore di prodotto logico. Alcune proprietà fondamentali dell'operazione di prodotto logico sono le seguenti (come prima sono identità):

$$x_1 \cdot x_2 = x_2 \cdot x_1$$

e:

$$1 \cdot x = x$$

e anche:

$$0 \cdot x = 0$$

Come la somma, anche l'operazione di prodotto logico è associativa, vale a dire:

$$x_1 \cdot (x_2 \cdot x_3) = (x_1 \cdot x_2) \cdot x_3$$

pertanto si può estendere a $n \geq 2$ variabili come segue:

$$f = x_1 \cdot x_2 \cdot \ldots \cdot x_n$$

La funzione f assume valore 1 se e solo se tutte le variabili x_i ($1 \leq i \leq n$) hanno valore 1. Essa modella il comportamento della schiera di n deviatori collegati in serie; la Figura A.1c mostra appunto il caso $n = 2$.

L'ultima possibilità che interessa qui è anch'essa molto comune. Si supponga che i due punti luce siano collocati agli estremi di una scala: chiunque si aspetterebbe che azionandone uno, non importa quale, la lampada passi nello stato opposto di luminosità, giacché a motivo della rispettiva collocazione sarebbe molto scomodo doverli azionare entrambi. Si faccia l'ipotesi che la lampada sia spenta ($f = 0$) se entrambi i deviatori sono in posizione 0; allora portando in posizione 1 l'uno o l'altro, indifferentemente, la lampada si accende ($f = 1$). Azionando il primo deviatore si ha che $x_1 = 1$ e $x_2 = 0$, dunque la lampada è accesa; naturalmente riportando il deviatore x_1 in posizione 0 la lampada si spegne. Ma deve anche essere possibile spegnerla portando x_2 in posizione 1, cioè si deve avere $f = 0$ se $x_1 = x_2 = 1$. La Figura A.1d mostra come collegare i due dispositivi per ottenere tale comportamento; qui i due deviatori sono usati in modo completo. L'operazione logica corrispondente si chiama "differenza simmetrica" o OR esclusivo, in genere denotata come XOR (Exclusive OR, o anche EOR); algebricamente si rappresenta con l'operatore binario "$\oplus$". Alcune proprietà della differenza simmetrica sono date dalle identità seguenti:

$$x_1 \oplus x_2 = x_2 \oplus x_1$$

e:

$$1 \oplus x = \bar{x}$$

e anche:

$$0 \oplus x = x$$

Gli schemi mostrati in Figura A.1 sono proprio quelli tradizionali usati per realizzare l'impianto di illuminazione domestico. Il lettore può immaginare da sé come realizzare tre o più punti luce secondo i vari schemi. Per lo schema (d) occorre però un dispositivo nuovo, il *commutatore*, che generalizza il deviatore (come del resto questo generalizza l'interruttore).

L'operatore unario "$^{-}$" (soprallineatura) indica l'operazione di "negazione" (o inversione o complemento logico, da non confondere con quello aritmetico) o NOT. La funzione $f(x) = \bar{x}$ è a una sola variabile e assume valore 1 se e solo se x ha valore 0. Si dice anche che la variabile di ingresso x viene *negata* (o *invertita* o *complementata*).[1]

Illustrare l'idea di variabile binaria o logica, e conseguentemente di funzione logica, mediante deviatori e lampade elettriche va benissimo perché si tratta di dispositivi macroscopici semplici e familiari. Ma i concetti così esemplificati si adattano egualmente bene per modellare i circuiti microelettronici che elaborano il flusso di informazione nel calcolatore. La variabile

[1] Talvolta si scrive $/x$ invece di $\bar{x}$.

logica è rappresentata dal valore di tensione o corrente elettrica, invece che dalla posizione del deviatore. Per esempio si supponga che i valori di tensione elettrica presenti su ingressi e uscite del circuito possano essere solo 0 e +5 Volt e si convenga esprimano i valori logici 0 e 1, rispettivamente. Con tale convenzione è possibile rappresentare la relazione tra ingresso e uscita, cioè come funziona il circuito, sotto forma di tabella di verità della funzione logica che ne modella l'attività.

Il transistore è il dispositivo elettronico elementare tramite cui si possono progettare i circuiti, semplicissimi, che effettuano le operazioni logiche elementari AND, OR, XOR e NOT. Tali circuiti elettronici elementari vengono chiamati *porte* (*gate*) e la Figura A.2 ne mostra la rappresentazione grafica convenzionale. Spesso si rappresenta la porta NOT semplicemente sotto forma di pallino, e se la porta NOT si trova subito a monte di una porta AND, OR, XOR o d'altro tipo il pallino viene spostato in avanti appoggiandolo all'ingresso della porta a valle (più avanti se ne vedranno alcuni esempi).

A.2 Sintesi di funzioni logiche

Si consideri la *rete combinatoria* (*combinatorial network*) di Figura A.3a, costituita da porte AND e OR. Essa equivale all'espressione logica seguente:

$$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

La tabella di verità della funzione f è mostrata in Figura A.3b. La rete combinatoria funziona così: dapprima i valori logici presenti agli ingressi passano attraverso le porte AND e le uscite rispettive assumono valore conforme al funzionamento della porta AND; poi tali valori di uscita intermedi si propagano in ingresso alla porta OR che calcola ed emette il valore di uscita finale f. Pertanto la rete combinatoria calcola la funzione logica f.

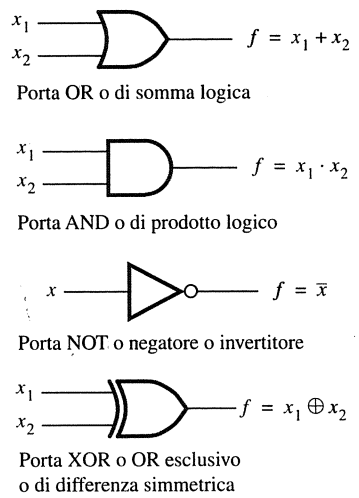
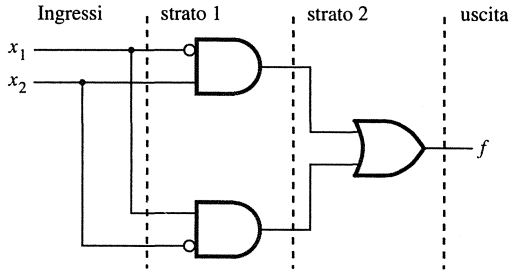


Figura A.2 Rappresentazione grafica delle porte logiche standard.



$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$

(a) Rete combinatoria della funzione XOR

#	x_1	x_2	$\bar{x}_1 \cdot x_2$	$x_1 \cdot \bar{x}_2$	$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2 = x_1 \oplus x_2$
0	0	0	0	0	0
1	0	1	1	0	1
2	1	0	0	1	1
3	1	1	0	0	0

(b)Tabella di verità della funzione XOR

Figura A.3 Sintesi della funzione XOR tramite porte AND, OR e NOT.

La tabella di verità della funzione f coincide con quella dell'operazione XOR (differenza simmetrica). Ne consegue che la rete combinatoria di Figura A.3a è una realizzazione circuitale, costruita usando porte AND, OR e NOT, della porta logica XOR. Si dice anche che la rete è una *sintesi* della funzione. L'espressione logica $\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$ è di tipo *somma di prodotti* (*Sum Of Products*, SOP), giacchè prima si calcolano i prodotti logici e poi la somma. Il circuito corrispondente è una rete combinatoria *a due livelli* (*two-level network*), cioè è costituito da due strati di porte logiche in cascata: il primo strato formato da porte AND e il secondo da una porta OR.

Si noti come a rigore si sarebbe dovuta denotare la funzione logica f nel modo seguente:

$$f = [(\bar{x}_1) \cdot x_2] + [x_1 \cdot (\bar{x}_2)]$$

mettendo in evidenza le parentesi per precisare l'ordine di valutazione delle varie operazioni logiche che figurano nell'espressione. Come nell'algebra dei numeri, anche nell'algebra di Boole si definisce l'ordine di precedenza tra operatori al fine di ridurre l'uso di parentesi: la negazione precede il prodotto che a sua volta precede la somma; inoltre, come si è già anticipato, se non c'è pericolo di confusione l'operatore di prodotto "·" si omette.

A.2.1 Sintesi intuitiva

Ritornando alla forma di tipo somma di prodotti occorre ora vedere come si possa sintetizzare la funzione logica partendo dalla tabella di verità per ricavarne la rete combinatoria. Si consideri

Tabella A.1 Tabella di verità di due funzioni logiche a tre variabili ciascuna

#	x_1	x_2	x_3	termine	f_1	termine	f_2
0	0	0	0	m_0	1	m_0	1
1	0	0	1	m_1	1	m_1	1
2	0	1	0	M_2	0	m_2	1
3	0	1	1	m_3	1	M_3	0
4	1	0	0	M_4	0	m_4	1
5	1	0	1	M_5	0	m_5	1
6	1	1	0	M_6	0	M_6	0
7	1	1	1	m_7	1	M_7	0

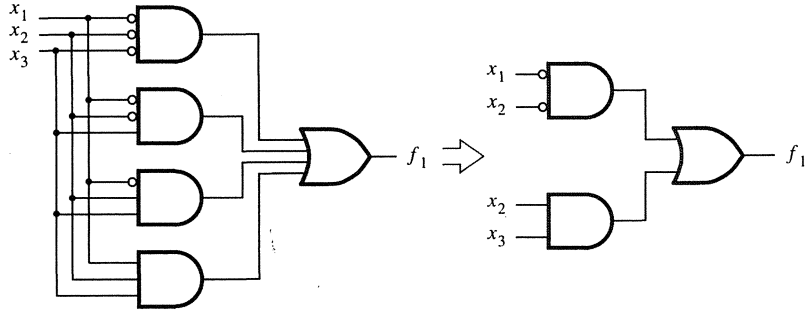
la tabella di verità data in Tabella A.1 e si supponga di dovere sintetizzare la funzione f_1 usando solo porte AND, OR e NOT. Per ciascuna riga della tabella in cui figura $f_1 = 1$ bisogna prevedere un termine prodotto nell'espressione della funzione, vale a dire una porta AND nella rete logica corrispondente, formato da tre fattori, cioè le tre variabili di ingresso. Occorre però negare alcune variabili e precisamente quelle che in ingresso hanno valore 0. Detto altrimenti, se in ingresso si ha $x_i = 0$ allora nel prodotto deve figurare il fattore $\bar{x}_i$, se in ingresso si ha $x_i = 1$ allora nel prodotto deve figurare il fattore x_i. Per esempio, nella quarta riga di Tabella A.1 la funzione f_1 assume valore 1 in corrispondenza dei valori in ingresso seguenti:

$$(x_1, x_2, x_3) = (0, 1, 1)$$

Il termine prodotto corrispondente è $\bar{x}_1 x_2 x_3$. Ripetendo tutto ciò per ogni riga dove f_1 vale 1 si ottiene l'espressione logica seguente:

$$f_1 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2 x_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3$$

La rete logica corrispondente all'espressione è mostrata a lato di Figura A.4. La forma di tipo somma di prodotti così ottenuta si chiama anche *prima forma canonica* della funzione.



$f_1 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2 x_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3$

(a) Prima forma canonica

$f_1 = \bar{x}_1\bar{x}_2 + x_2 x_3$

(b) Forma minima (tipo SOP)

Figura A.4 Sintesi della funzione f_1 in prima forma canonica e in forma minima.

A.2.2 Forma canonica

Il metodo di costruzione della prima forma canonica si applica a qualsiasi tabella di verità e dà sempre luogo a una sintesi a due livelli, di tipo somma di prodotti. In generale è strutturato così:

- si numerano le configurazioni delle $n \geq 1$ variabili della funzione come se esse fossero numeri naturali a n bit;
- si definisce *mintermine* (da termine *minimo*, *minterm*) una funzione combinatoria a $n \geq 1$ variabili assumente valore 1 in una sola configurazione delle variabili e 0 in tutte le altre $2^n - 1$ configurazioni;
- si scrive m_i per indicare il mintermine che assume valore 1 nella configurazione numero i ($0 \leq i \leq 2^n - 1$) delle variabili;
- la funzione f è semplicemente la somma logica di tutti i mintermini corrispondenti agli 1 nella colonna di uscita. Infatti la funzione vale 1 se e solo se uno dei mintermini costituenti vale 1, il che avviene in tutte e sole le configurazioni corrispondenti a ciascuno di essi: questo è esattamente il comportamento della funzione, come mostra la tabella di verità.

In riferimento alla Tabella A.1 si ha quanto segue:

$$f_1 = m_0 + m_1 + m_3 + m_7$$

e:

$$f_2 = m_0 + m_1 + m_2 + m_4 + m_5$$

Ora basta osservare che per sintetizzare il mintermine m_i si deve prendere il prodotto delle n variabili (fattori elementari), dove ciascuna di esse figura nel prodotto m_i in forma diretta se nella configurazione numero i vale 1, negata se vale 0. Chiaramente m_i varrà 1 solo nella configurazione numero i, perché in ogni altra almeno uno dei fattori elementari costituenti m_i assume valore 0. Per esempio, si hanno $m_0 = \bar{x}_1\bar{x}_2\bar{x}_3$, $m_1 = \bar{x}_1\bar{x}_2x_3$, $m_2 = \bar{x}_1x_2\bar{x}_3$, ..., $m_7 = x_1x_2x_3$. Sostituendo si riottiene la sintesi di f_1 già data sopra e si otterrebbe anche quella di f_2. Le reti combinatorie corrispondenti sono a due livelli: primo strato porte AND, secondo strato una porta OR, più eventualmente porte NOT nel primo strato.

Alternativamente, si può sempre sintetizzare la funzione data come tabella di verità in *seconda forma canonica*, di tipo *prodotto di somme* o POS (*Product Of Sums*). Un *maxtermine* M_i (da *termine massimo*, *maxterm*) è definito come una funzione di $n \geq 1$ variabili assumente valore 0 solo nella configurazione numero i delle variabili ($0 \leq i \leq 2^n - 1$) e 1 in tutte le altre. La funzione è data come prodotto logico (non somma) di maxtermini. Infatti essa vale 0 se e solo se uno dei maxtermini costituenti vale 0, il che avviene in tutte e sole le configurazioni corrispondenti: esattamente il comportamento della funzione. Esaminando la Tabella A.1 si ha quanto segue:

$$f_1 = M_2 \cdot M_4 \cdot M_5 \cdot M_6$$

e:

$$f_2 = M_3 \cdot M_6 \cdot M_7$$

Poi basta osservare che per sintetizzare il maxtermine M_i si deve prendere la somma logica delle n variabili (termini elementari), dove ciascuna di esse figura nella somma M_i in forma negata se nella configurazione numero i vale 1, diretta se vale 0 (esattamente l'opposto di quanto si fa con il mintermine: non dimenticare!). Chiaramente M_i varrà 0 solo nella configurazione numero i perché in ogni altra almeno uno dei termini elementari costituenti M_i assume valore 1. Per esempio, si hanno $M_0 = x_1 + x_2 + x_3$, $M_1 = x_1 + x_2 + \bar{x}_3$, $M_2 = x_1 + \bar{x}_2 + x_3$, ..., $M_7 = \bar{x}_1 + \bar{x}_2 + \bar{x}_3$. Sostituendo si ottiene la sintesi di f_1:

$$f_1 = (x_1 + \bar{x}_2 + x_3) \cdot (\bar{x}_1 + x_2 + x_3) \cdot (\bar{x}_1 + x_2 + \bar{x}_3) \cdot (\bar{x}_1 + \bar{x}_2 + x_3)$$

e, volendo, quella di f_2 (però se ne lascia il compito al lettore); l'operatore di prodotto logico "·" si potrebbe omettere (le parentesi specificano l'ordine di valutazione). La rete combinatoria corrispondente alla seconda forma canonica è anch'essa a due livelli, ma il primo strato contiene porte OR e il secondo una porta AND, più alcune porte NOT (nel primo strato).

Si tenga presente che la prima e seconda forma canonica di una medesima funzione possono avere complessità strutturale molto diversa. Si intuisce come convenga la prima forma piuttosto che la seconda se la colonna di uscita della funzione contiene pochi valori 1 piuttosto che pochi 0, rispettivamente. Come esempio ulteriore, tramite il metodo delle forme canoniche si potrebbe ricavare la forma di tipo somma di prodotti della funzione XOR (a due variabili). Naturalmente il metodo si generalizza a funzioni combinatorie date come tabelle di verità con qualunque numero $n \geq 1$ di variabili.

Si osservi che le forme canoniche di una data funzione sono uniche; funzioni diverse hanno forme canoniche diverse (del resto in Matematica *canonico* significa appunto *unico*). Dato che ogni funzione combinatoria si può sintetizzare in modo unico come somma di mintermini o prodotto di maxtermini, spesso per specificarla si elencano sotto segno di sommatoria i numeri dei mintermini costituenti, o di produttoria quelli dei maxtermini. Per esempio, in riferimento alla funzione f_2 di Tabella A.1 si hanno le due seguenti rappresentazioni equivalenti:

$$f_2 = \sum_1 \underbrace{(0,1,2,4,5)}_{mintermini} \qquad \text{e} \qquad f_2 = \prod_0 \underbrace{(3,6,7)}_{maxtermini}$$

e similmente per la funzione f_1. Chiaramente si ricava l'una dall'altra, per esclusione (se nella prima c'è il mintermine i nella seconda non c'è il maxtermine i e viceversa). Sono rappresentazioni comode e molto sintetiche.

In conclusione, ogni funzione combinatoria (ovvero tabella di verità) ammette sempre due sintesi in prima e seconda forma canonica (forma SOP e POS) determinate univocamente dalla funzione. La forma canonica corrisponde alla rete combinatoria a due livelli (ma non ogni rete a due livelli corrisponde a una forma canonica). Pertanto la rete a due livelli è un modello di circuito logico *universale*, cioè atto a sintetizzare qualunque funzione combinatoria.

A.3 Sintesi in forma minima

Si è appena visto come derivare la forma di tipo somma di prodotti della funzione partendo dalla tabella di verità. Due espressioni logiche si dicono *equivalenti* se hanno la medesima tabella di verità (idem per due reti combinatorie). Si vede facilmente che data la tabella di verità esistono numerose espressioni di tipo somma di prodotti e reti combinatorie a due livelli che le corrispondono (oltre alle due forme canoniche), e che dunque sono equivalenti. Per esempio l'espressione logica seguente è equivalente a quella della funzione f_1 ricavata nel paragrafo precedente:

$$\bar{x}_1\bar{x}_2 + x_2x_3$$

Per dimostrarlo basta ricavare la tabella di verità dell'espressione e verificare come essa coincida con quella mostrata in Tabella A.1. La Tabella A.2 mostra il processo di calcolo e il risultato, dividendolo in tre fasi: nella prima, per ogni combinazione di ingressi viene calcolato il termine $\bar{x}_1\bar{x}_2$; nella seconda il termine x_2x_3; infine nella terza le due colonne vengono sommate (logicamente) e si ottiene la colonna complessiva dell'espressione; essa risulta identica a quella di Tabella A.1.

Tabella A.2 Tabella di verità dell'espressione logica $\bar{x}_1\bar{x}_2 + x_2x_3 = f_1$

#	x_1	x_2	x_3	$\bar{x}_1\bar{x}_2$	x_2x_3	$\bar{x}_1\bar{x}_2 + x_2x_3 = f_1$
0	0	0	0	1	0	1
1	0	0	1	1	0	1
2	0	1	0	0	0	0
3	0	1	1	0	1	1
4	1	0	0	0	0	0
5	1	0	1	0	0	0
6	1	1	0	0	0	0
7	1	1	1	0	1	1

Per semplificare le espressioni logiche occorre sottoporle a una serie di trasformazioni algebriche. Le regole dell'algebra di Boole utili per tale compito sono date ora. La prima è la proprietà distributiva del prodotto rispetto alla somma:

$$x\,(y + z) = xy + xz$$

e poi l'identità seguente, chiamata anche legge di complemento logico:

$$x + \bar{x} = 1$$

La Tabella A.3 dà la dimostrazione della validità della proprietà distributiva. Dovrebbe essere chiaro che regole come quella, e in realtà tutte le identità dell'algebra di Boole, si possono dimostrare calcolando separatamente la tabella di verità della parte destra e sinistra e poi verificando che le due tabelle coincidano. La proprietà distributiva assume anche la forma seguente:

$$x + yz = (x + y)\,(x + z)$$

Essa afferma che si può distribuire la somma rispetto al prodotto. Nel seguito, tuttavia, non occorrerà farne uso.

Lo scopo della semplificazione dell'espressione logica è quello di ridurre il costo della rete combinatoria che la calcola, secondo un qualche criterio di valutazione del costo. Ciò che interessa qui è partire dalla prima forma canonica della funzione, che è di tipo somma di prodotti, e vedere se non esista una formulazione equivalente della funzione come *somma di prodotti minima*. Ma affinché il concetto di forma minima abbia senso bisogna dare la regola di costo dell'espressione, almeno per il tipo somma di prodotti. Il modo intuitivo per farlo è quello di contare il numero totale di porte e di ingressi alle porte necessari per sintetizzare la rete combinatoria corrispondente all'espressione logica, come mostra la Figura A.4.

Tabella A.3 Dimostrazione di equivalenza tra espressioni logiche

#	x	y	z	$y + z$	$x\,(y + z)$	xy	xz	$xy + xz$
0	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0
2	0	1	0	1	0	0	0	0
3	0	1	1	1	0	0	0	0
4	1	0	0	0	0	0	0	0
5	1	0	1	1	1	0	1	1
6	1	1	0	1	1	1	0	1
7	1	1	1	1	1	1	1	1

Per esempio l'espressione più lunga in Figura A.4 ha costo pari a 21, derivante dalla somma di 5 porte e 16 ingressi complessivi alle porte. Si noti che le porte NOT vengono trascurate da questo criterio di costo. Nella stessa figura il costo dell'espressione più breve vale solo 9, risultante dalla somma di 3 porte e 6 ingressi. Tale criterio di costo vale per qualunque rete combinatoria, ma è anche un po' vago giacché conta insieme porte e ingressi, ovvero quantità disomogenee.

Un criterio di costo ben formalizzato è quello cosiddetto *dei letterali*: vale solo per le espressioni logiche a due livelli di tipo sia SOP sia POS. Tale criterio stabilisce che il costo dell'espressione sia uguale al *numero di comparse di variabili* nell'espressione stessa, contandone anche le ripetizioni. Per esempio, la forma di tipo POS seguente (è la seconda forma canonica di f_1):

$$f_1 = (x_1 + \bar{x}_2 + x_3) \cdot (\bar{x}_1 + x_2 + x_3) \cdot (\bar{x}_1 + x_2 + \bar{x}_3) \cdot (\bar{x}_1 + \bar{x}_2 + x_3)$$

ha costo pari a $3 + 3 + 3 + 3 = 12$ letterali, cioè 12 comparse delle tre variabili x_1, x_2 e x_3, in forma diretta o negata. Si vede facilmente come il criterio di costo dei letterali di fatto conti il numero totale di ingressi del primo strato di porte logiche della rete a due livelli (strato AND o OR per forma SOP o POS, rispettivamente), non contando né quelli del secondo strato (porta OR o AND) né le porte AND e OR né i negatori NOT. Tuttavia il criterio è ragionevole perché in una rete combinatoria a due livelli il primo strato di porte ne consuma il maggior numero (e ciò vale soprattutto per reti a molti ingressi principali) e la sua dimensione è circa proporzionale al numero di ingressi delle porte costituenti, mentre nel secondo strato figura sempre una sola porta con molti meno ingressi che non nel primo e i negatori sono porte di taglia piccola, più o meno trascurabili. Il critero non vale per reti con tre o più strati, perché valutare il costo dell'intera rete considerando solo il primo di molti strati sarebbe chiaramente troppo riduttivo.

Si può dunque definire come *minima* un'espressione logica di tipo somma di prodotti che non ammette alcuna altra espressione logica di tipo somma di prodotti equivalente e di costo inferiore, secondo un criterio di costo fissato in anticipo. L'analoga definizione si applica all'espressione di tipo prodotto di somme. Negli esempi semplici che si considereranno qui è abbastanza facile vedere in modo intuitivo quando si sia arrivati a ottenere un'espressione logica minima. Per il momento dunque non verranno date dimostrazioni rigorose di minimalità.

La strategia per effettuare la trasformazione algebrica per semplificare l'espressione logica è la seguente: prima si raggruppano a coppie i termini che differiscono solo in quanto la stessa variabile figura in uno in forma diretta (x_i) e nell'altro in forma negata ($\bar{x}_i$); poi raccogliendo a fattore comune le variabili che figurano identiche in entrambi i termini resta il fattore $(x_i + \bar{x}_i)$, che per la legge di complemento logico precedente ha valore costante 1. Per esempio, applicando tale trasformazione alla funzione f_1 si ottiene quanto segue:

$$\begin{aligned} f_1 &= \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2x_3 + x_1x_2x_3 = \\ &= \bar{x}_1\bar{x}_2(\bar{x}_3 + x_3) + (\bar{x}_1 + x_1)\,x_2x_3 = \\ &= \bar{x}_1\bar{x}_2 \cdot 1 + 1 \cdot x_2x_3 = \\ &= \bar{x}_1\bar{x}_2 + x_2x_3 \qquad \text{(forma minima)} \end{aligned}$$

L'espressione logica ottenuta come risultato è minima e la rete combinatoria corrispondente (a due livelli) è mostrata in Figura A.4.

Tuttavia raggruppare i termini a coppie in modo tale da arrivare alla forma minima dell'espressione non è sempre un processo così lineare come potrebbe parere giudicando solo dall'esempio appena mostrato. Spesso occorre inframmezzare l'applicazione della regola seguente:

$$x + x = x$$

Essa si chiama *legge di idempotenza* e permette di duplicare (una o più volte) il termine rendendo possibili raggruppamenti a coppie che altrimenti sfuggirebbero.

Per esempio, si consideri la funzione f_2 di Tabella A.1. L'espressione di tipo somma di prodotti derivabile direttamente dalla tabella di verità (ovvero la prima forma canonica di f_2) è la seguente, come si è già visto:

$$f_2 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3$$

Duplicando il primo termine prodotto $\bar{x}_1\bar{x}_2\bar{x}_3$ e permutando i termini (giacché la somma logica è commutativa) si ottiene quanto segue:

$$f_2 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3$$

Raggruppando i termini a coppie come detto prima si ottengono i raccoglimenti seguenti a fattore comune, seguiti da semplificazione:

$$\begin{aligned} f_2 &= \bar{x}_1\bar{x}_2(\bar{x}_3 + x_3) + x_1\bar{x}_2\,(\bar{x}_3 + x_3) + \bar{x}_1\,(\bar{x}_2 + x_2)\,\bar{x}_3 = \\ &= \bar{x}_1 \cdot \bar{x}_2 \cdot 1 + x_1 \cdot \bar{x}_2 \cdot 1 + \bar{x}_1 \cdot 1 \cdot \bar{x}_3 \\ &= \bar{x}_1\bar{x}_2 + x_1\bar{x}_2 + \bar{x}_1\bar{x}_3 \end{aligned}$$

Si può raccogliere di nuovo tra i primi due termini e semplificando ancora si ottiene infine la forma minima seguente:

$$\begin{aligned} f_2 &= \bar{x}_1\bar{x}_2 + x_1\bar{x}_2 + \bar{x}_1\bar{x}_3 = \\ &= (\bar{x}_1 + x_1)\,\bar{x}_2 + \bar{x}_1\bar{x}_3 = \\ &= 1 \cdot \bar{x}_2 + \bar{x}_1 \cdot \bar{x}_3 = \\ &= \bar{x}_2 + \bar{x}_1\bar{x}_3 \qquad \text{(forma minima)} \end{aligned}$$

la cui minimalità è intuitivamente evidente: ogni variabile compare una volta sola.

Non è difficile constatare come l'espressione di f_2 appena ottenuta costi molto meno di quella di partenza rispetto a entrambi i criteri di costo introdotti prima. In particolare il costo in letterali dell'espressione di f_2 di partenza è $3 + 3 + 3 + 3 + 3 = 15$, mentre la forma ridotta appena ottenuta costa solo $1 + 2 = 3$ letterali.

Ciò completa la breve discussione sulla semplificazione algebrica delle espressioni logiche. L'utilità pratica di tale tecnica matematica deriva dal fatto, in sé piuttosto evidente, che la rete combinatoria corrispondente alla forma minima è più economica, in quanto contiene il minimo numero di porte logiche e ingressi fra tutte quelle equivalenti della stessa forma (SOP o POS). Pertanto ha interesse economico disporre di algoritmi per ricavare una forma minima di qualsiasi espressione logica data.

La Tabella A.4 elenca tutte le regole dell'algebra di Boole menzionate o usate finora. Esse si presentano a coppie ottenute scambiando il ruolo di somma e prodotto logico; tale

Tabella A.4 Regole (assiomi e teoremi) dell'algebra di Boole

Proprietà	Regola	Forma duale
commutativa	$x + y = y + x$	$x \cdot y = y \cdot x$
associativa	$(x + y) + z = x + (y + z)$	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
distributiva	$x + y \cdot z = (x + y) \cdot (x + z)$	$x \cdot (y + z) = x \cdot y + x \cdot z$
di idempotenza	$x + x = x$	$x \cdot x = x$
di involuzione (o di doppia negazione)	$\bar{\bar{x}} = x$	
di complemento	$x + \bar{x} = 1$	$x \cdot \bar{x} = 0$
teorema di De Morgan	$\overline{x + y} = \bar{x} \cdot \bar{y}$	$\overline{x \cdot y} = \bar{x} + \bar{y}$
dello 1 e dello 0	$1 + x = 1$	$0 \cdot x = 0$
dell'elemento neutro	$0 + x = x$	$1 \cdot x = x$

caratteristica è del tutto generale e va sotto il nome di *dualità*.[2]. Per il momento la legge (o teorema) di De Morgan e quella di involuzione (o di doppia negazione) non si sono ancora dimostrate di qualche utilità, ma lo saranno nel prossimo paragrafo. Molte delle regole elencate in Tabella A.4 corrispondono ad altrettante note proprietà di ragionamento logico, come la legge di doppia negazione e le due di complemento, le quali ultime due corrispondono ai principî del *terzo escluso* (*tertium non datur*) e di *non-contraddizione*, rispettivamente.

A.3.1 Metodo di Karnaugh

Ricavando la forma minima delle due funzioni f_1 e f_2 di Tabella A.1 qua e là è stato necessario decidere intuitivamente quale fosse il modo migliore di procedere. Ciò si vede per esempio nella decisione di duplicare il termine $\bar{x}_1\bar{x}_2\bar{x}_3$ (applicando la legge di idempotenza), giacché come fare il primo passo per semplificare f_2 non era affatto ovvio. C'è un metodo di tipo geometrico molto utile per ricavare rapidamente l'espressione logica di costo minimo della funzione, quanto meno se essa non ha troppe variabili in ingresso. L'essenza del metodo consiste nel rappresentare la tabella di verità della funzione in modo diverso, più adatto allo scopo. La nuova forma della tabella si chiama *mappa di Karnaugh* (dal nome dell'inventore, pronuncia *kaernaf*) come anche il metodo che ne deriva. Per la funzione a tre variabili la mappa di Karnaugh è rettangolare con due righe, quattro colonne e otto caselle, come mostra la Figura A.5a. Ciascuna casella contiene il valore di uscita che la funzione assume in corrispondenza di una sola delle otto triple dei valori in ingresso applicati come indici di riga e colonna alla casella in esame. Per esempio, la terza casella (da sinistra) della riga superiore della mappa a sinistra in Figura A.5a contiene il valore 0 che la funzione dà in uscita in corrispondenza della configurazione di ingresso $(x_1, x_2, x_3) = (1, 1, 0)$; nella tabella di verità sarebbe la riga numero 6. Per comodità si possono numerare tutte le caselle della mappa secondo le righe corrispondenti della tabella di verità; si veda la Figura A.5.

L'idea chiave sottostante alla costruzione della mappa è che caselle adiacenti orizzontalmente e verticalmente (cioè quelle che hanno in comune un lato; non basta che si tocchino per un vertice!) siano associate a configurazioni di ingresso adiacenti, vale a dire che differiscono per un solo bit. Esaminando le mappe date in Figura A.5 si può constatare che ciò è vero, proprio per come le configurazioni di variabili in ingresso fanno riferimento a righe e colonne della mappa.

Se accade che due caselle adiacenti nel senso geometrico sopraddetto contengano entrambe valore 1 c'è la possibilità di effettuare un passo di semplificazione della funzione. Per esempio nella mappa della funzione f_2 in Figura A.5 (in alto a destra) i due valori 1 contenuti nelle prime due caselle (a sinistra) della riga superiore della mappa (sono le caselle corrispondenti alle righe 0 e 2) corrispondono ai termini prodotto $\bar{x}_1\bar{x}_2\bar{x}_3$ e $\bar{x}_1x_2\bar{x}_3$, rispettivamente (sono i mintermini che si otterrebbero tramite la costruzione della prima forma canonica di f_2). Si vede subito come sia possibile effettuare la semplificazione seguente:

$$\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 = \bar{x}_1\,(\bar{x}_2 + x_2)\,\bar{x}_3 = \bar{x}_1 \cdot 1 \cdot \bar{x}_3 = \bar{x}_1\bar{x}_3$$

Essa si ottiene raccogliendo a fattore comune le variabili che figurano identiche in entrambi i termini prodotto e poi applicando la legge di complemento logico. Si tratta della stessa semplificazione di f_2 già vista prima in modo puramente algebrico, che qui ricompare ma vista sotto veste geometrica: se ne scopre immediatamente la possibilità semplicemente notando nella mappa la presenza di due caselle adiacenti ed entrambe contenenti 1. Più in ge-

[2] La cosiddetta legge di doppia negazione è autoduale e figura una volta sola.

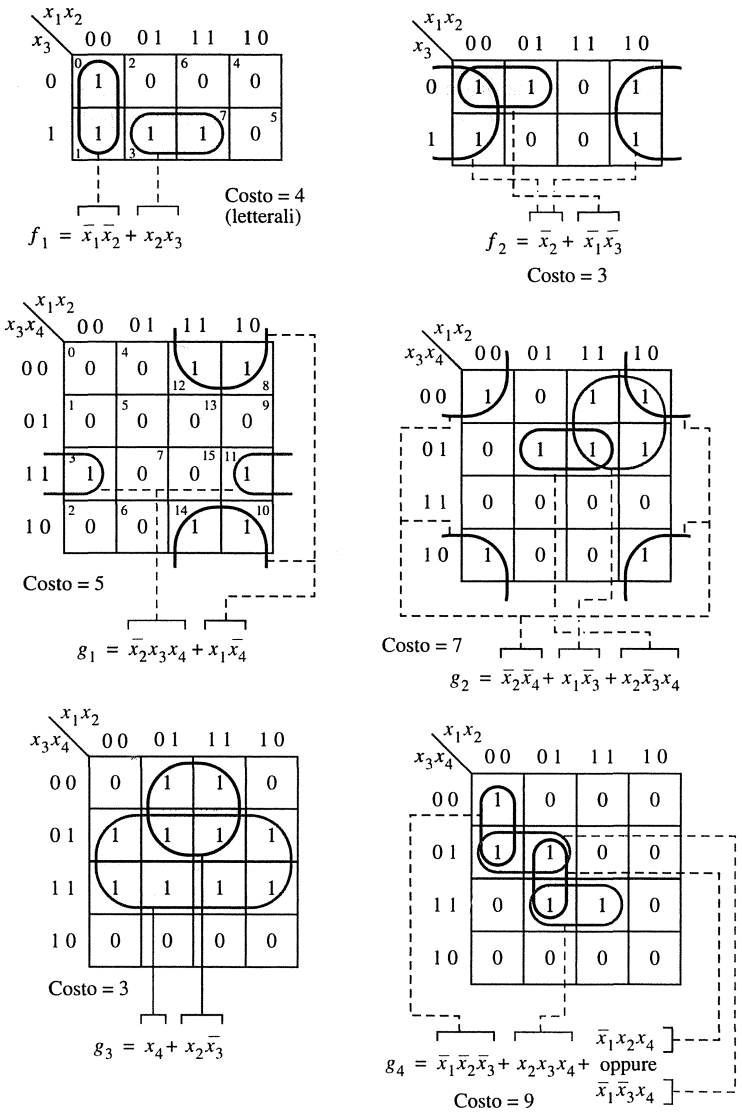


Figura A.5 Semplificazione con mappa di Karnaugh in forma a due livelli SOP.

nerale certi gruppi di caselle adiacenti e contenenti 1 danno luogo a semplificazioni; ma bisogna avvertire subito che il gruppo deve soddisfare determinate regole di costruzione: il gruppo deve presentare numero di caselle contenenti 1 pari a una potenza di due (cioè 2, 4, 8 ecc.) e la disposizione delle caselle deve corrispondere a uno schema di adiacenza ben preciso. Trovato il gruppo adatto, il termine prodotto che ne deriva è quello caratterizzato

dalle variabili di ingresso che *non mutano valore* passando da una casella all'altra nel gruppo. Se tutte le caselle del gruppo sono indicizzate dalla variabile $x_i = 0$, allora nel termine prodotto corrispondente al gruppo deve figurare $\bar{x}_i$, altrimenti (cioè se sono indicizzate con $x_i = 1$) deve figurare x_i ($1 \leq i \leq n$). L'adiacenza tra caselle va intesa in senso un po' più ampio di quanto non suggerisca l'intuizione geometrica immediata: le due caselle sul lato sinistro vanno considerate adiacenti alle due caselle sul lato destro, ciascuna con quella della riga rispettiva. Ne viene che il gruppo di 4 caselle contenenti 1 formato dalla colonna di sinistra immaginata appaiata alla colonna di destra (pertanto il gruppo di caselle numero 0, 1, 4 e 5) forma un quadrato dove, esaminando gli indici di riga e colonna, si vede che solo la variabile x_2 mantiene costante il proprio valore ($x_2 = 0$), le altre due variabili x_1 e x_3 lo cambiano *in tutti i modi e combinazioni possibili* ($(x_1, x_3) = 00, 10, 01, 11$, cioè esattamente 4 combinazioni corrispondenti alle 4 caselle del gruppo). Il gruppo di 4 caselle così individuato dà luogo alla semplificazione (algebrica) seguente:

$$
\begin{aligned}
&\bar{x}_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2x_3 = \\
&= \bar{x}_2\bar{x}_1\bar{x}_3 + \bar{x}_2x_1\bar{x}_3 + \bar{x}_2\bar{x}_1x_3 + \bar{x}_2x_1x_3 = \\
&= \bar{x}_2\,(\bar{x}_1\bar{x}_3 + x_1\bar{x}_3 + \bar{x}_1x_3 + x_1x_3) = \\
&= \bar{x}_2\,((\bar{x}_1 + x_1)\,\bar{x}_3 + (\bar{x}_1 + x_1)\,x_3) = \\
&= \bar{x}_2\,(1 \cdot \bar{x}_3 + 1 \cdot x_3) = \\
&= \bar{x}_2\,(\bar{x}_3 + x_3) = \\
&= \bar{x}_2 \cdot 1 = \\
&= \bar{x}_2
\end{aligned}
$$

da cui si comprende come le variabili che mutano valore di ingresso si semplifichino tutte, mentre resta solo quella che non muta valore e che dunque è comune a tutti i termini prodotto originari. È solito cerchiare sulla mappa i gruppi validi di caselle come mostra la Figura A.5.

Con le debite generalizzazioni, il metodo della mappa di Karnaugh funziona per un numero qualsiasi di variabili di ingresso, anche superiore a tre. Si ottiene la mappa di Karnaugh a quattro variabili appaiandone in verticale due da tre. La Figura A.5 mostra altri quattro esempi di mappe, da quattro variabili x_1, x_2, x_3 e x_4 ciascuna, e per ognuna riporta le forme minime delle funzioni rappresentate. Nella mappa a quattro variabili si possono formare gruppi di due, quattro e anche otto caselle adiacenti (secondo le regole enunciate prima). La mappa della funzione g_3 mostra appunto un gruppo da $2^3 = 8$ caselle (numero 1, 3, 5, 7, 9, 11, 13, 15) contenenti 1 costruito in modo corretto (costituito dalle due righe centrali): la variabile x_4 mantiene valore 1 per tutte le caselle del gruppo e le altre tre assumono tutte le 8 configurazioni possibili. Le caselle della colonna di sinistra e della riga superiore vanno considerate ordinatamente adiacenti a quelle della colonna di destra e della riga inferiore, rispettivamente. Ne viene che le quattro caselle d'angolo potrebbero formare (qualora tutte contenessero valore 1) un gruppo valido, costruito correttamente; ciò accade, per esempio, nella mappa della funzione g_2, dove il termine prodotto corrispondente è $\bar{x}_2\bar{x}_4$. Come già nella mappa a tre variabili, il termine associato al gruppo di caselle corretto risulta dal prodotto di tutte le variabili che non mutano valore tra le caselle del gruppo, prese in forma diretta se valgono 1, in forma negata altrimenti (cioè se valgono 0): Per esempio, il gruppo delle quattro caselle disposte a quadrato nell'angolo superiore destro della mappa di g_2 dà luogo al termine $x_1\bar{x}_3$ proprio perché le variabili di riga e colonna che non mutano valore sono x_1 e x_3 e mantengono i valori 1 e 0, rispettivamente; le variabili x_2 e x_4, invece, cambiano valore e formano tutte le 4 combinazioni possibili.

Esiste anche la mappa di Karnaugh a cinque variabili, ma non è più una griglia unica; la si può immaginare come formata dall'insieme di due mappe a quattro variabili, dove la quinta variabile di ingresso x_5 ha valore fisso a 0 per tutte le caselle della prima mappa, fisso

a 1 per tutte quelle della seconda. Si può immaginare il tutto come se le due mappe fossero sovrapposte a sandwich, quella sotto marcata con $x_5 = 0$ e quella sopra con $x_5 = 1$. Non è difficile immaginare come proceda la costruzione della mappa a sei e più variabili. La mappa a sei variabili è formata dal gruppo di quattro mappe da quattro variabili ciascuna (o se si preferisce di due da cinque). I gruppi di caselle adiacenti possono essere contenuti in una sola mappa da quattro variabili o estendersi su più mappe coprendo caselle corrispondenti. Si possono formare gruppi sempre più grandi, fermo restando che il gruppo deve contenere 2^i caselle per un intero $i \geq 1$ e che comunque la disposizione delle caselle del gruppo segue sempre le regole viste prima.

Il gruppo di caselle contenenti 1 valido secondo le regole enunciate prima si chiama *implicante*. Se l'implicante non è interamente contenuto in un altro implicante più esteso allora si dice *implicante primo* (qui primo ha significato di più esteso). Gli implicanti cerchiati in Figura A.5 sono tutti primi. D'ora in avanti verrà usata la nuova terminologia.

Ora bisogna vedere come usare la mappa di Karnaugh per ricavare la forma di tipo somma di prodotti di costo minimo, secondo il criterio di costo dei letterali. Come mostrano bene le varie mappe in Figura A.5, a implicanti estesi, cioè contenenti numerose caselle, corrispondono termini prodotto brevi, cioè costituiti da pochi fattori. Si comprende pertanto che per avere espressioni logiche di costo basso, secondo il criterio di costo dei letterali, è necessario *coprire tutte le caselle contenenti 1 usando solo implicanti primi e scegliendoli in modo da ridurre al minimo il costo complessivo della funzione*. Si prenda come esempio la funzione g_2 (a quattro variabili) la cui mappa è mostrata in Figura A.5. Come si è già argomentato prima, le quattro caselle d'angolo formano un implicante primo e il termine corrispondente è $\bar{x}_2\bar{x}_4$. C'è un secondo implicante primo di quattro caselle nell'angolo superiore destro della mappa, il cui termine associato è $x_1\bar{x}_3$.

Con questi due implicanti si coprono tutte le caselle contenenti 1 nella mappa, tranne quella indicizzata da $(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$. L'implicante primo più esteso che comprenda anche tale casella è da due caselle e corrisponde al termine $x_2\bar{x}_3x_4$, di costo pari a 3 letterali. Pertanto l'espressione logica di costo minimo che dà la funzione g_2 è la seguente:

$$g_2 = \bar{x}_2\bar{x}_4 + x_1\bar{x}_3 + x_2\bar{x}_3x_4$$

il cui costo in letterali è $2 + 2 + 3 = 7$. In modo del tutto analogo si possono ricavare le espressioni logiche di costo minimo (secondo il criterio sopraddetto) di tutte le altre funzioni mostrate in Figura A.5. Non sempre la soluzione di costo minimo è unica; ciò accade, per esempio, alla funzione g_4 dove se ne trovano due di pari costo: la prima comprende il termine $\bar{x}_1x_2x_4$, la seconda $\bar{x}_1\bar{x}_3x_4$ (e i due termini sono di pari costo, ovvero 3 letterali ciascuno). L'unicità della forma minima è abbastanza rara e in genere si dà in casi molto semplici, essendoci piuttosto di norma la presenza di soluzioni multiple. Può anche succedere che una casella non sia adiacente a nessun'altra e resti isolata; in tale caso essa dà luogo a un mintermine non semplificabile.

Dovrebbe essere ormai chiaro che il metodo di Karnaugh fornisce un algoritmo rigoroso per trovare la sintesi minima di tipo SOP di una funzione combinatoria qualunque, secondo il criterio di costo dei letterali. La rete combinatoria minima risultante è dunque sempre a due livelli. Il metodo si estende anche alla ricerca della forma minima di tipo POS (ancora secondo il criterio di costo dei letterali). Basta considerare i gruppi di caselle della mappa contenenti 0 (invece di 1) e costruire gli implicanti primi con le solite regole, però scambiando AND con OR e variabili dirette con negate (come si faceva con prima e seconda forma canonica). Qui non si daranno i dettagli del metodo in forma POS, lasciati al lettore. Si tenga presente che in generale le forme minime SOP e POS di una stessa funzione hanno costo differente e che è difficile prevedere quale sarà la meno costosa; pertanto andrebbero trovate entrambe e poi confrontate direttamente.

La natura geometrica della mappa di Karnaugh la rende molto adatta per individuare a occhio gli implicanti primi e dunque ricavare una forma minima della funzione. Crescendo la dimensione della mappa il tutto si fa più difficile, ma sono stati messi a punto algoritmi molto efficienti per automatizzare il processo di semplificazione; essi esulano però dagli scopi del libro, pertanto si rimanda alla bibliografia [2, 3] e soprattutto alla letteratura scientifica di settore.

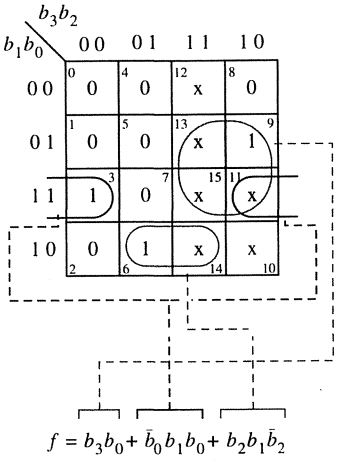
A.3.2 Condizione di indifferenza

In numerose situazioni pratiche accade che determinate configurazioni di bit non si presentino mai in ingresso al circuito digitale. Per esempio, si consideri la rappresentazione del numero intero decimale in codice BCD. La singola cifra decimale (da 0 a 9) è rappresentata tramite la configurazione di quattro bit, ovvero dalle quattro variabili binarie b_3, b_2, b_1 e b_0, come mostra la Figura A.6. Le quattro variabili possono assumere fino a $2^4 = 16$ combinazioni di valori differenti, ma solo dieci sono poste in corrispondenza con le cifre decimali da rappresentare. Le configurazioni rimanenti non sono usate, pertanto il circuito logico che dovesse ricevere in ingresso la codifica BCD non le vedrebbe comparire mai.

La Figura A.6a mostra la tabella di verità di una funzione logica che ha in ingresso la codifica BCD, vale a dire le variabili b_3, b_2, b_1 e b_0; per il momento la funzione effettiva calcolata non ha importanza, serve solo come esempio. Comunque di certo non è necessario precisare quali valori essa debba assumere in corrispondenza della configurazioni di ingresso che non si presentano mai; nella tabella si mette il simbolo "×" per rappresentare una *condizione di indifferenza* (*don't care condition* o semplicemente *don't care*). Durante il processo di sintesi della rete combinatoria a ciascuna condizione di indifferenza si può attribuire indipendentemente o valore 0 o valore 1 in modo arbitrario. Naturalmente conviene fare scelte

Cifra decimale	Codifica binaria #	b_3	b_2	b_1	b_0	f
0	0	0	0	0	0	0
1	1	0	0	0	1	0
2	2	0	0	1	0	0
3	3	0	0	1	1	1
4	4	0	1	0	0	0
5	5	0	1	0	1	0
6	6	0	1	1	0	1
7	7	0	1	1	1	0
8	8	1	0	0	0	0
9	9	1	0	0	1	1
Non usate	10	1	0	1	0	x
	11	1	0	1	1	x
	12	1	1	0	0	x
	13	1	1	0	1	x
	14	1	1	1	0	x
	15	1	1	1	1	x

(a) Tabella di verità



(b) Mappa a quattro variabili

Figura A.6 Sintesi con mappa di Karnaugh e condizioni di indifferenza.

che conducano alla semplificazione più spinta, cioè a costo minimo. Occorre resistere alla tentazione di sfruttare le condizioni di indifferenza per estendere il più possibile gli implicanti primi (pensando che così costerebbero di meno), cioè non è vero che porle sistematicamente a 1 sia la scelta migliore. Infatti, mettendo molti valori 1 nella mappa si creano sì implicanti primi estesi e poco costosi, ma può accadere che se ne aumenti il numero annullando il guadagno ottenuto se non addirittura mutandolo in perdita. Di solito si trovano buone scelte cercando di bilanciare l'estensione degli implicanti primi rispetto al loro numero, e non è sempre facile capire quali possano essere le migliori; naturalmente si danno anche casi limite.

Ora si può dire quale calcolo faccia la funzione f: assume valore 1 se e solo se la cifra decimale in ingresso (codificata BCD) è diversa da 0 e divisibile per 3. La Figura A.6b mostra la mappa di Karnaugh (a quattro variabili) di f con le condizioni di indifferenza. Si vede come occorrano tre implicanti primi per coprire tutta quanta la funzione e che alcune condizioni di indifferenza (ma non tutte) vadano poste a 1 per favorire la formazione di implicanti primi estesi; tuttavia ce n'è anche una (nell'angolo inferiore a destra, casella numero 10) che sarebbe dannoso porre a 1 perché costringerebbe a costruire un nuovo implicante primo per coprirla, senza peraltro dare contributo significativo alla copertura di altre caselle utili.

A.4 Sintesi con porte universali

Bisogna ora considerare due altre porte logiche fondamentali chiamate NAND e NOR, di largo uso pratico perché circuitalmente sono molto semplici. La Figura A.7 mostra le tabelle di verità delle due nuove operazioni (o funzioni elementari) logiche. Esse sono equivalenti alle funzioni AND e OR seguite da porta NOT, rispettivamente (donde il nome); in Figura A.7 sono riportate anche le rispettive rappresentazioni grafiche standard.

I simboli "↑" e "↓" (freccia in su e in giù) denotano gli operatori binari corrispondenti alle porte NAND e NOR, rispettivamente. Esplicitando la definizione dei due nuovi operatori e poi applicando il teorema di De Morgan, come figura in Tabella A.4, si ottiene quanto segue:

$$x_1 \uparrow x_2 = \overline{x_1 + x_2} = \bar{x}_1 + \bar{x}_2$$

e anche:

$$x_1 \downarrow x_2 = \overline{x_1 \cdot x_2} = \bar{x}_1 \cdot \bar{x}_2$$

#	x_1	x_2	f
0	0	0	1
1	0	1	1
2	1	0	1
3	1	1	0

$f = x_1 \uparrow x_2 = \overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$

#	x_1	x_2	f
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	0

$f = x_1 \downarrow x_2 = \overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

(a) Porta NAND

(b) Porta NOR

Figura A.7 Porte universali NAND e NOR a due ingressi e tabelle di verità.

La definizione di operazione NAND e NOR si estende senza difficoltà anche a più di due operandi, sempre appoggiandosi al teorema di De Morgan come segue:

$$x_1 \uparrow x_2 \uparrow \ldots \uparrow x_n = \overline{x_1 \cdot x_2 \cdots x_n} = \bar{x}_1 + \bar{x}_2 + \cdots + \bar{x}_n$$

e anche:

$$x_1 \downarrow x_2 \downarrow \ldots \downarrow x_n = \overline{x_1 + x_2 + \cdots + x_n} = \bar{x}_1 \cdot \bar{x}_2 \cdots \bar{x}_n$$

Dalle definizioni si vede come le due operazioni NAND e NOR godano entrambe della proprietà commutativa (ma non di quella associativa, come tra breve si dirà).

La porta NAND è *universale*, cioè per qualsiasi rete combinatoria ne esiste un'altra equivalente alla prima ma costruita usando solo porte NAND, nient'altro; lo stesso vale per la porta NOR. Tuttavia, ricavare la rete combinatoria corrispondente a una funzione combinatoria data servendosi solo di porte NAND oppure NOR non è così semplice come lavorando con le porte AND, OR e NOT. Una delle difficoltà maggiori deriva proprio dal fatto che le due operazioni "↑" e "↓" non godono della proprietà associativa; tra poco la questione verrà discussa con il dettaglio opportuno. Prima però si consideri una procedura semplice e generale per sintetizzare qualsiasi funzione combinatoria tramite porte NAND, data la tabella di verità della funzione. C'è un modo diretto per trasformare l'espressione logica di tipo somma di prodotti in una equivalente ma facente uso solo di operatore "↑", corrispondente dunque a una rete combinatoria costituita solo da porte NAND. L'esempio seguente basta a illustrare come funziona la trasformazione: l'espressione di partenza contiene solo l'operatore "↑" e la rete combinatoria corrispondente è costituita da tre porte NAND a due ingressi ciascuna; si badi bene che le parentesi indicano esplicitamente l'ordine di associazione (giacché l'operazione "↑" non è di per sé associativa) e che cambiandone la disposizione la funzione muta. Ecco che cosa si ottiene trasformando:

$$\begin{aligned}(x_1 \uparrow x_2) \uparrow (x_3 \uparrow x_4) &= \overline{\overline{(x_1 \cdot x_2)} \cdot \overline{(x_3 \cdot x_4)}} = \\ &= \overline{\overline{x_1 \cdot x_2}} + \overline{\overline{x_3 \cdot x_4}} = \\ &= x_1 x_2 + x_3 x_4\end{aligned}$$

La trasformazione richiede l'uso del teorema di De Morgan e della legge di involuzione (o se si preferisce di doppia negazione). La Figura A.8 mostra il processo di trasformazione in forma di successione di reti combinatorie equivalenti. La trasformazione è chiaramente di tipo generale, cioè funziona su qualsiasi forma di tipo somma di prodotti di partenza. Siccome ogni funzione combinatoria ammette espressione logica in forma di somma di prodotti (per esempio, la prima forma canonica) e la trasformazione appena vista è chiaramente percorribile in verso opposto, se ne ricava il metodo per trasformare qualunque funzione logica di tipo AND-OR-NOT in una di tipo solo NAND e dunque per sintetizzarla come rete combinatoria costituita solo da porte NAND, indipendentemente dal numero di variabili in ingresso. Tra l'altro si vede bene che il numero di porte NAND è uguale alla somma del numero di porte AND e OR originali. Resterebbe da dire come trattare la porta NOT (negatore), ma basta osservare che coincide con la porta NAND con ingressi unificati (si veda la Figura A.8); lo stesso vale usando la porta NOR.

Ora bisogna concludere in merito alla mancanza di proprietà associativa per le operazioni NAND e NOR. Tale proprietà gioca ruolo essenziale nello scomporre la porta a tre o più ingressi in una rete di porte a solo due ingressi. Le operazioni AND e OR sono associative, pertanto porte a tre o più ingressi si costruiscono come cascata o albero di porte a solo due ingressi, rispettivamente. La Figura A.9a mostra come fare per il caso di tre ingressi (tramite costruzione in cascata) e lascia intuire come estendere a quattro o più ingressi. La Figura A.9c mostra come fare per il caso di quattro ingressi (tramite costruzione ad albero); si estende nel modo ovvio a più ingressi. Siccome, però, né la porta NAND né la porta NOR sono associative la scomposizione in cascata o ad albero non si può fare. Quando dunque occorre

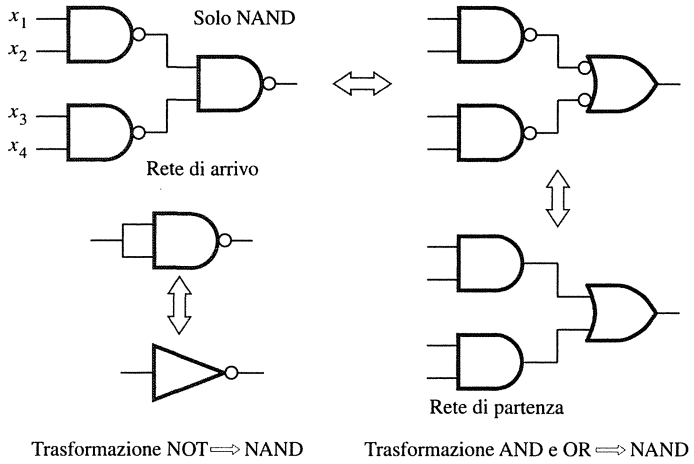
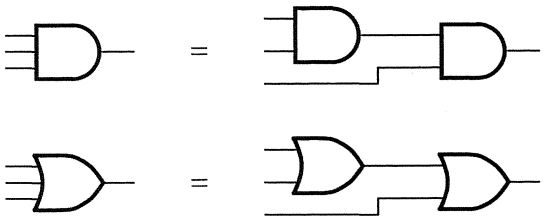


Figura A.8 Equivalenza tra sistema AND-OR e sistema NAND.

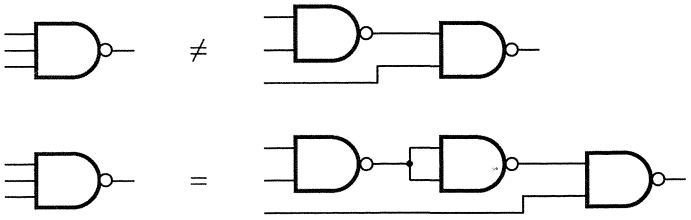
avere una porta NAND o NOR a tre o più ingressi bisogna in prima battuta sperare che essa sia già disponibile come componente finito di libreria. Tuttavia volendo (o dovendo) si può procedere anche per sintesi, ma la soluzione per la porta NAND (o NOR) non si presenta così semplice come per AND o OR. La Figura A.9b mostra che per la porta NAND la costruzione in cascata (in alto) è errata, e nonostante la soluzione corretta esista (in basso) richiede ben tre porte NAND invece di due. Tutto quanto appena detto vale anche per la porta NOR, che ha ruolo duale nei confronti della porta NAND. Per approfondimenti in merito al progetto di reti combinatorie con insiemi di porte di differente composizione si rimanda alla bibliografia [2, 3] e alla letteratura scientifica di settore.

È importante per il lettore comprendere bene che la funzione logica data per esempio tramite tabella di verità è essenzialmente unica (al limite si possono ordinare le righe della tabella in modo differente, ma ciò non cambia la funzione), ma che in generale le corrispondono infinite reti combinatorie; esse sono tutte strutturalmente differenti ma equivalenti e dunque corrispondenti alla stessa funzione logica. Inoltre non si deve fare confusione tra funzione logica, espressione logica e rete combinatoria. C'è corrispondenza biunivoca strutturale tra espressione logica e rete combinatoria; due reti combinatorie strutturalmente differenti ma equivalenti hanno espressioni logiche corrispondenti diverse come aspetto ma algebricamente trasformabili l'una nell'altra; le espressioni logiche corrispondenti a reti combinatorie equivalenti sono in definitiva due modi diversi di esprimere la stessa funzione logica tramite formulazione come legge analitica. In tale cornice il metodo della mappa di Karnaugh costituisce una tecnica a base geometrica per mettere ordine nella molteplicità delle trasformazioni algebriche e per renderle visibili; esistono, però, anche metodi (assai raffinati) di semplificazione algebrica diretti.

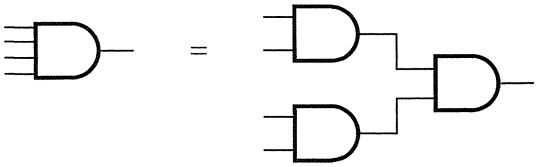
Dato dunque che ci sono infinite reti combinatorie equivalenti, nella pratica interessano quelle di costo minimo. Il criterio di costo dei letterali (o quello legato al numero di ingressi e porte nella rete) è molto importante, ma non è affatto l'unico. Spesso importa anche abbreviare il più possibile l'intervallo di tempo di propagazione dei segnali lungo la rete dagli ingressi all'uscita, vale a dire aumentarne la frequenza di operazione. Quest'ultimo però è un



(a) Costruzione in cascata delle porte AND e OR a tre ingressi usando solo porte AND e OR a due ingressi



(b) Costruzione della porta NAND a tre ingressi usando solo porte NAND a due ingressi (sopra errata sotto corretta)



(c) Costruzione ad albero della porta AND a quattro ingressi usando solo porte AND a due ingressi (idem per porta OR)

Figura A.9 Costruzione in cascata e ad albero di porte a tre e quattro ingressi.

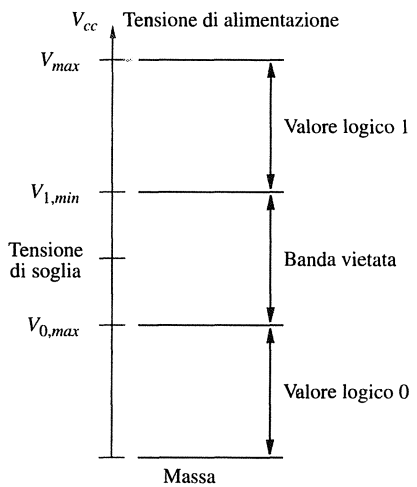
criterio di costo diverso e dà luogo all'insieme appropriato di concetti e algoritmi di semplificazione, differenti da quelli del metodo di Karnaugh (benché comunque non poche idee e modi di procedere siano comuni). Comprendere bene i principi di semplificazione della funzione logica è importante, ma oggigiorno le procedure di riduzione del costo vengono applicate di rado a mano. Infatti sono stati messi a punto sistemi automatici sofisticati e potenti per manipolare funzioni logiche e reti combinatorie, basati su calcolatore. Tali strumenti automatici vanno sotto nome generico di *sistemi CAD* (*Computer Aided Design*) per progetto digitale. Il progettista deve solo specificare al sistema CAD il *comportamento logico* della funzione di cui necessita (per esempio dandone la tabella di verità, che indica quali siano i valori di uscita della funzione ma non chiarisce affatto come sia fatta la rete combinatoria capace di calcolarli a partire dagli ingressi) ed esso procede a ricavare la *struttura circuitale* della rete

combinatoria corrispondente ottimale rispetto al criterio di costo che sempre il progettista ha indicato come quello da rispettare. Per approfondimenti su tale importante settore tecnologico si rimanda alla bibliografia [2, 3].

A.5 Tecnologia microelettronica

È tempo di dedicare attenzione alle tecnologie microelettroniche adatte a rappresentare la variabile logica e costruire la rete combinatoria, ovvero calcolare la funzione logica corrispondente. La scelta della grandezza fisica mediante cui rappresentare la variabile binaria dipende, come è ovvio, da considerazioni fisiche e tecnologiche. Nel circuito elettronico si usa o la *tensione elettrica* (voltaggio) o l'*intensità di corrente elettrica* (amperaggio); per la verità la prima possibilità è prevalente, benché la seconda non manchi di esempi.[3]

Per stabilire la corrispondenza tra la tensione elettrica, vale a dire una grandezza continua che può assumere infiniti valori, e la variabile binaria, che invece è discreta e assume solo due valori, si ricorre al concetto di *soglia di separazione*. Tutti i valori di tensione superiori a quello di soglia rappresentano, indifferentemente, il valore logico 1; tutti quelli inferiori il valore logico 0. In pratica, però, la tensione nel circuito elettronico reale è sempre affetta da rumore casuale che ne fa oscillare leggermente il valore effettivo; ne consegue che i valori di tensione molto vicini alla soglia la attraversano frequentemente, pertanto sono imprevedibili e dunque ambigui. Per evitare l'incertezza derivantene si conviene che la banda dei valori intorno alla tensione di soglia (sia inferiori sia superiori) sia vietata, come mostra la Figura A.10. Sono riportati i valori tipici per le tecnologie TTL e MOS alimentate a 5 Volt; entrambe sono molto usate nel calcolatore. Tutti i livelli di tensione elettrica sotto $V_{0,max}$ rappresentano



Valori tipici per la tecnologia TTL:

- $V_{cc} = 5$ Volt
- $V_{massa} = 0$ Volt
- $V_{1,min} \approx 2$ Volt
- $V_{0,max} \approx 0{,}8$ Volt
- Soglia $\approx 1{,}5$ Volt

Valori tipici per la tecnologia MOS:

- $V_{cc} = 5$ Volt
- $V_{massa} = 0$ Volt
- $V_{1,min} \approx 3{,}8$ Volt
- $V_{0,max} \approx 1{,}3$ Volt
- Soglia $\approx 2{,}5$ Volt

Figura A.10 Rappresentazione del valore binario mediante tensione elettrica.

[3] Per aggiungere qualcosa, nella fibra ottica il segnale binario trasmesso è rappresentato dalla potenza ottica; anche altre grandezze fisiche sono in uso, benché molto meno comuni.

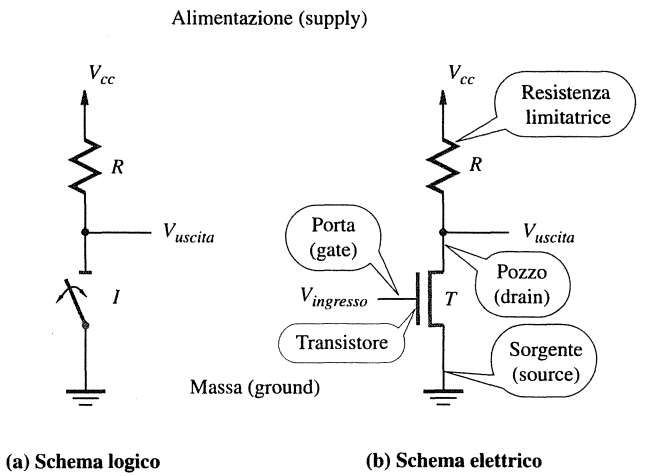
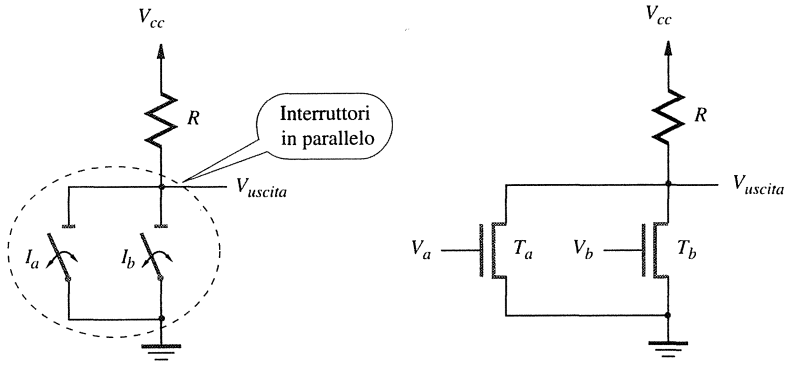


Figura A.11 Schema logico ed elettrico della porta NOT (negatore o invertitore).

il valore logico 0, tutti quelli sopra $V_{1,min}$ il valore logico 1, mentre quelli compresi tra $V_{0,max}$ e $V_{1,min}$ sono vietati e non si dovrebbero presentare mai, tranne che per errore del progettista o guasto del circuito.

Esistono numerose tecnologie microelettroniche per circuiti digitali. Qui se ne illustrano alcune partendo dall'idea di usare resistenze e transistori funzionanti come interruttori; la Figura A.11 illustra l'esempio di partenza. La Figura A.11a mostra il modello di funzionamento del circuito, incentrato sull'interruttore *I* (a due stati, chiuso e aperto). Quando *I* è in stato di chiuso, la tensione di uscita V_{uscita} è uguale a quella di massa, vale a dire 0 Volt. Quando invece *I* è in stato di aperto la tensione di uscita è uguale a quella di alimentazione V_{cc}. La Figura A.11b mostra lo schema elettrico effettivo del circuito, dove l'interruttore *I* è realizzato tramite il transistore *T*. Il transistore è appunto un dispositivo elettrico allo stato solido che, tra i modi di funzionamento fondamentali, ha quello di interruttore (oltre ad altri). Applicando tensione 0 Volt al morsetto di *porta* (*gate*) di *T* (cioè se $V_{ingresso} = 0$) il transistore si pone in stato di *interdizione* (*turned-off*), vale a dire si comporta come interruttore aperto e non permette il passaggio di corrente tra i due morsetti *sorgente* (*source*) e *pozzo* (*drain*); in base all'analogia con la Figura A.11a la tensione di uscita si porta al valore di alimentazione (cioè si ha $V_{uscita} = 0$). Applicando invece tensione V_{cc} Volt al morsetto di porta di *T* (cioè se $V_{ingresso} = V_{cc}$) il transistore si pone in stato di *conduzione* (*turned-on*), vale a dire si comporta come interruttore chiuso, dunque permette passaggio di corrente tra i due morsetti sorgente e pozzo e con la stessa giustificazione di prima la tensione di uscita si porta al valore di massa (cioè si ha $V_{uscita} = 0$). Interpretando i valori di tensione 0 e V_{cc} come i valori logici 0 e 1, rispettivamente, si vede che la relazione tra ingresso e uscita del circuito è quella della porta NOT, vale a dire il circuito proposto è una *realizzazione fisica* (implementazione) della porta NOT. La resistenza *R* serve essenzialmente a limitare l'intensità di corrente e non ha specifica funzione logica (ma mostra come ci siano dettagli elettrici da considerare che esulano dallo scopo del libro). Comunque, la resistenza *R* deve essere abbastanza grande da garantire che l'intensità di corrente V_{cc}/R sia modesta ma non troppo vicina a 0, altrimenti si potrebbe confondere con altre correnti dovute



(a) Schema logico **(b) Schema elettrico**

Figura A.12 Schema logico ed elettrico della porta NOR.

a inevitabili effetti secondari. La tecnologia microelettronica impiegante transistore e resistenza si chiama RTL (*Resistor Transistor Logic*) e di solito è alimentata con V_{cc} = 5 Volt. Esistono anche altre tecnologie con tensione di alimentazione più o meno elevata.

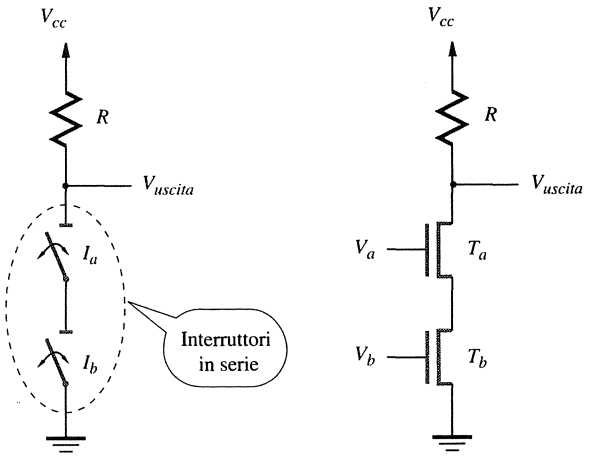
Ora si può passare a porte un po' più complesse. La Figura A.12 illustra il circuito elettronico corrispondente alla porta NOR. La Figura A.12a mostra il modello logico del circuito, dove i due interruttori I_a e I_b sono collegati in parallelo. Si vede che la tensione di uscita V_{uscita} sale a livello di alimentazione (cioè $V_{uscita} = V_{cc}$) solo se entrambi gli interruttori I_a e I_b sono in stato di aperto, altrimenti (cioè se almeno uno è in stato di chiuso, o anche tutti e due) scende a massa (cioè $V_{uscita} = 0$). Per la solita analogia tra transistore e interruttore, in Figura A.12b la tensione di uscita V_{uscita} vale V_{cc} se e solo se entrambi i morsetti di porta dei transistori T_a a T_b sono a 0 (cioè $V_a = V_b = 0$), altrimenti vale 0. Pertanto (con la solita corrispondenza tra tensione elettrica e valore logico) si vede che la relazione tra ingressi e uscita è la stessa della porta NOR, dove i valori di tensione ai morsetti di ingresso V_a e V_b del circuito corrispondono alle due variabili di ingresso x_1 e x_2 della porta. Si conclude che il circuito di Figura A.12b *realizza fisicamente* la porta NOR.

Non è difficile constatare che si ottiene la realizzazione fisica della porta NAND mettendo i due transistori in serie invece che in parallelo, come mostra la Figura A.13 complessiva (a sinistra il modello logico, a destra la realtà fisica). Si lascia al lettore il compito di ripetere ragionamento e analisi.

Infine si potrebbero benissimo dare i circuiti elettrici che realizzano le porte AND e OR. Basta osservare che la porta AND equivale a prendere la porta NAND e a collegarle in serie una porta NOT (per la legge di doppia negazione due negatori in cascata si annullano). Dunque basta collegare in serie i circuiti di porta NAND e NOT di Figura A.13b e Figura A.11b, rispettivamente. Analogo ragionamento darebbe la realizzazione fisica della porta OR.

A.5.1 Tecnologia CMOS

La serie di figure da A.11 ad A.13 illustra i principi della *tecnologia microelettronica NMOS*, dove l'acronimo NMOS indica il tipo di transistore su cui si basano le porte logiche rea-



(a) Schema logico **(b) Schema elettrico**

Figura A.13 Schema logico ed elettrico della porta NAND.

lizzate tramite tale tecnologia. Esistono due tipi di transistore a *metallo-ossido-semiconduttore* o MOS (*Metal-Oxide-Semiconductor transistor*): il tipo *negativo* o NMOS (*Negative MOS transistor*) e quello *positivo* o PMOS (*Positive MOS transistor*); quello visto prima per realizzare la porta NOT, NOR e le altre è il tipo NMOS. Entrambi funzionano come interruttori, ma sono controllati in ingresso da valori di tensione in opposizione e dunque da valori logici complementari. La Figura A.14 illustra i comportamenti di entrambi modellandoli come interruttori.

Quando la tensione al morsetto di base (sinonimo per porta) si trova al valore di alimentazione V_{cc}, il transistore di tipo NMOS è in conduzione (on) e si comporta come interruttore chiuso, altrimenti è in interdizione (off) e si comporta come interruttore aperto; la Figura A.14a illustra la situazione. Al contrario, quando la tensione al morsetto di base si trova al valore di alimentazione V_{cc}, il transistore di tipo PMOS è in interdizione e si comporta come interruttore aperto, altrimenti è in conduzione e si comporta come interruttore chiuso; la Figura A.14b illustra la situazione. L'allineamento tra metà superiore e inferiore di Figura A.14 aiuta a cogliere la complementarità dei due tipi di transistore. Si noti come la rappresentazione grafica standard del transistore di tipo PMOS comprenda il pallino di negazione sul morsetto di base (porta) come richiamo mnemonico all'opposizione di comportamento. Ciò vale pure per i nomi dei morsetti di *sorgente* (*source*) e *pozzo* (*drain*), scambiati tra i due tipi di transistore; anche il collegamento tra massa e alimentazione è invertito; il tutto si spiegherebbe esaminando più a fondo la natura fisica dei due transistori mediante i concetti e metodi della Meccanica Quantistica, il che esula dagli scopi del libro e d'altra parte non è indispensabile per quanto segue.

Il grande svantaggio della tecnologia NMOS e dei circuiti di porta che ne derivano (Figure da A.11 a A.13) sta nel consumo elevato di energia che essi esibiscono. Quando vi è continuità elettrica tra alimentazione e massa attraverso la resistenza R, fluisce corrente; non per ogni

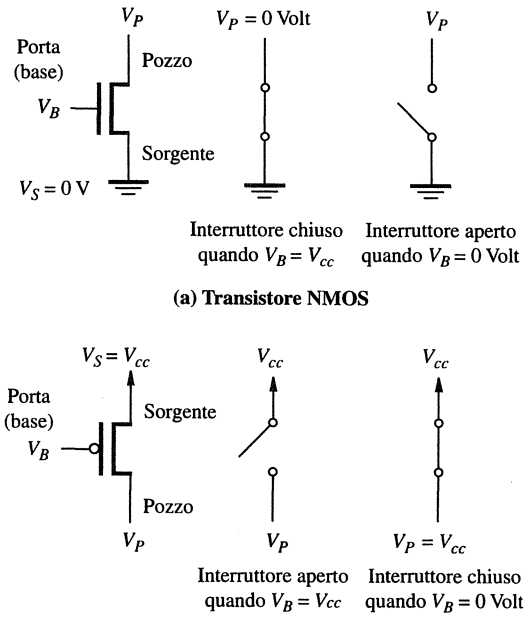


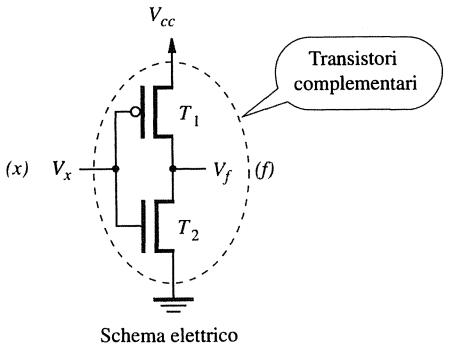
Figura A.14 Transistore NMOS e PMOS e funzionamento logico-elettrico.

configurazione di ingresso si presenta tale situazione, ma certamente per alcune sì.[4] Pertanto almeno per certe configurazioni di ingresso la porta in tecnologia NMOS dissipa energia elettrica per effetto Joule (cioè come calore) sulla resistenza R e tale difetto non è eliminabile.

La soluzione viene passando alla tecnologia che mischia i due tipi di transistore nello stesso circuito, cui si dà il nome espressivo di *tecnologia MOS complementare* o CMOS (*Complementary MOS technology*). La tecnologia CMOS fa a meno di resistenze (usa solo i due transistori MOS complementari), che sono difficili da miniaturizzare. Ma soprattutto la porta logica CMOS ha la (notevole) proprietà di *non consumare energia* quando si trova in stato stabile,[5] cioè non sta commutando (ma ne consuma in commutazione). La porta NOT di Figura A.15 illustra il principio di funzionamento della tecnologia CMOS. Quando si ha $V_x = V_{cc}$ (qui si pone $V_x = V_{ingresso}$ e $V_f = V_{uscita}$), cioè quando la variabile binaria di ingresso x ha valore logico 1, i transistori PMOS e NMOS T_1 e T_2 sono in interdizione e conduzione, rispettivamente, pertanto attraverso l'interruttore chiuso T_2 la tensione di uscita V_f si porta a 0 (massa), ovvero l'uscita f va a valore logico 0. Al contrario quando si ha $V_x = 0$ (massa), cioè

[4] Per capire a fondo si tenga presente che nel transistore MOS (di entrambi i tipi) non passa mai corrente tra morsetto di base (porta) e morsetti di pozzo o sorgente, ma solo (eventualmente) tra sorgente e pozzo.

[5] Per effetti dispersivi secondari il consumo non è esattamente nullo, ma comunque bassissimo.



Costo = 2 transistori

x	V_x	T_1	T_2	V_f	f
0	low	on	off	high	1
1	high	off	on	low	0

Tabella di verità

Figura A.15 Schema elettrico CMOS e funzionamento della porta NOT.

quando la variabile binaria di ingresso x ha valore logico 0, T_1 e T_2 sono in conduzione e interdizione, rispettivamente, e pertanto attraverso l'interruttore chiuso T_1 la tensione di uscita V_f si porta a V_{cc}, ovvero l'uscita f va a valore logico 1. La relazione tra variabile di ingresso x e uscita f è quella della porta NOT (come mostra la tabella di verità in Figura A.15b, dove low = 0 e high = 1), di cui il circuito CMOS proposto è dunque una realizzazione fisica.

La proprietà fondamentale di tale tipo di circuito è che i due transistori complementari T_1 e T_2 lavorano sempre in opposizione: quando uno è in conduzione l'altro è in interdizione e viceversa. Pertanto c'è sempre continuità elettrica tra l'uscita da un lato e o massa o alimentazione dall'altro, ma mai tra massa e alimentazione direttamente tranne per un brevissimo intervallo di tempo quando la porta commuta e i due transistori cambiano stato. Ne consegue che il circuito CMOS non consuma energia quando è in stato stabile, ma solo quando commuta, e comunque ne consuma assai poca perché la transizione è di breve durata. La potenza elettrica dissipata dalla porta in tecnologia CMOS è dunque proporzionale alla frequenza di commutazione.

Ora si può estendere il concetto CMOS a circuiti dotati di $n \geq 1$ ingressi come mostra la Figura A.16. Il circuito è diviso in due rami: il *ramo NMOS* (*pull-down network*) è formato solo da transistori di tipo N e quando $f(x_1, \ldots, x_n) = 0$ stabilisce continuità tra massa e uscita portandola a 0; il *ramo PMOS* (*pull-up network*) solo da transistori di tipo P e quando $f(x_1, \ldots, x_n)$ stabilisce continuità tra alimentazione e uscita portandola a V_{cc}. Il comportamento dei due rami è complementare cosicché in stato stabile non c'è mai continuità tra massa e alimentazione e dunque nessun consumo di energia.

Il ramo NMOS del circuito CMOS è strutturato sulla falsariga di quanto si è visto per le porte NOT e NOR in tecnologia NMOS (Figure da A.11 ad A.13). Il ramo PMOS è derivato da quello NMOS per complementarità (cioè scambiando a tutti i livelli la configurazione in serie con quella in parallelo), in modo da esibire il comportamento opposto. La Figura A.17 e la Figura A.18 illustrano come costruire il circuito CMOS della porta NAND e NOR, rispettivamente, e le rispettive tabelle di verità. Infine la Figura A.19 illustra come costruire in tecnologia CMOS la porta AND, come prima collegando in serie la porta NAND e la porta NOT.

Oltre ad avere consumo di potenza ridotto, il circuito in tecnologia CMOS ha il vantaggio di consumare superficie limitata sulla piastrina di silicio perché il transistore MOS è molto piccolo. Ciò dà luogo a due conseguenze benefiche rilevanti: si possono integrare sullo stesso componente miliardi di transistori, da cui risultano i potenti processori moderni e le grandi

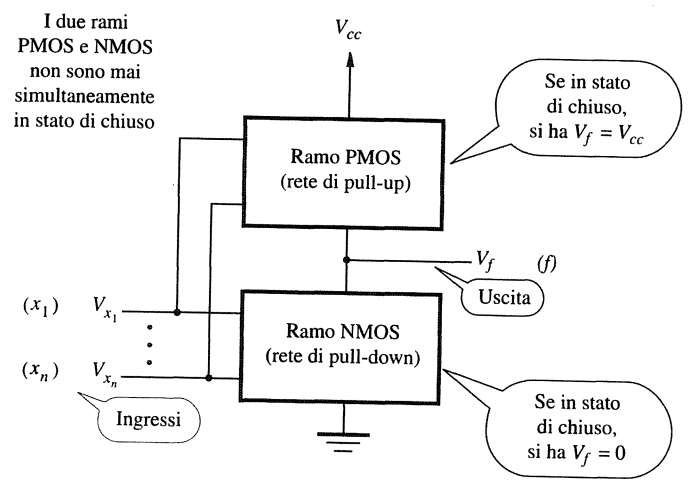
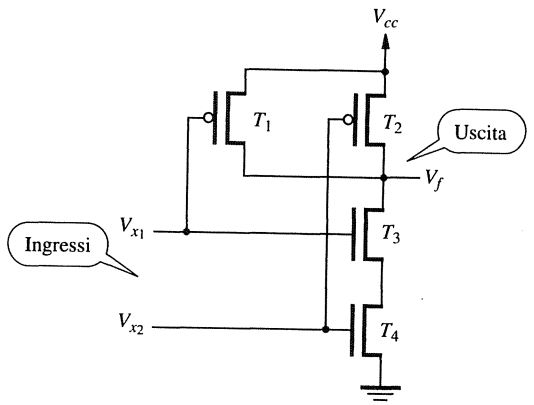


Figura A.16 Schema generale di rete combinatoria in tecnologia CMOS.



Costo = 4 transistori

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

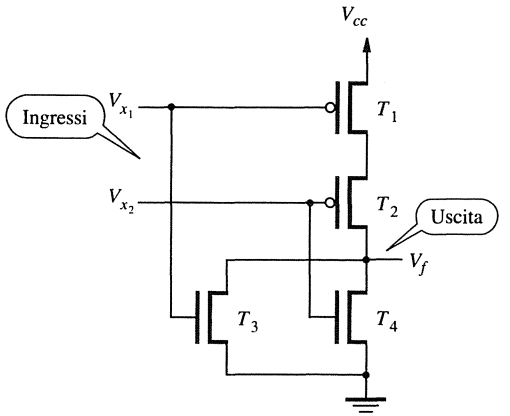
(b) Tabella di verità

(a) Schema elettrico

Figura A.17 Schema elettrico CMOS e funzionamento della porta NAND.

memorie elettroniche attuali; inoltre più piccolo è il transistore maggiore è la frequenza di commutazione massima e il risultato è che oggi i circuiti in tecnologia CMOS raggiungono frequenza dell'ordine del GigaHertz e perfino oltre.

La tecnologia CMOS ha numerose varianti funzionanti con tensione di alimentazione V_{cc} più o meno elevata, da 1,5 a 15 Volt; i due valori più comuni di alimentazione sono 5 e 3,3 Volt. Riducendo la tensione di alimentazione si riduce anche il consumo di potenza, giacché



Costo = 4 transistori

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

(b) Tabella di verità

(a) Schema elettrico

Figura A.18 Schema elettrico CMOS e funzionamento della porta NOR.

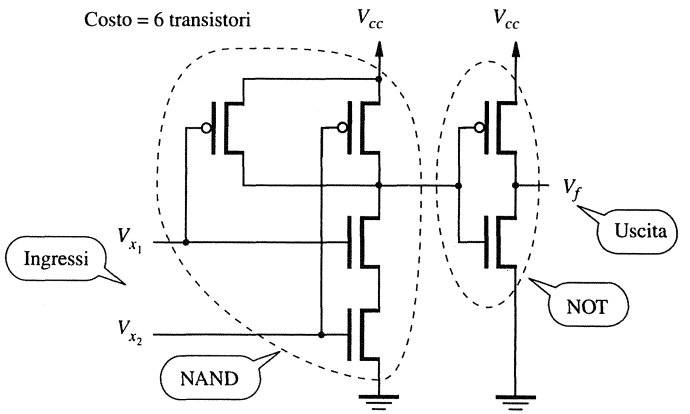


Figura A.19 Schema elettrico CMOS della porta AND derivata da NAND.

lo si dimostra essere proporzionale a V_{cc}^2 (quadrato della tensione di alimentazione), fatto che permette di integrare più transistori sulla stessa piastrina di silicio senza causare surriscaldamento. La bassa tensione di alimentazione presenta però anche uno svantaggio, ovvero maggiore sensibilità al rumore di fondo e ai disturbi elettromagnetici.

La Figura A.20 illustra come avvenga la transizione dell'uscita della porta CMOS tipica (la porta NOT) da tensione di alimentazione a tensione di massa. Sono riportati i valori tipici della tecnologia CMOS alimentata a 3,3 Volt; è ampiamente utilizzata in dispositivi dove si richiede consumo di potenza limitato. La curva, chiamata *caratteristica di trasferimento*, dà

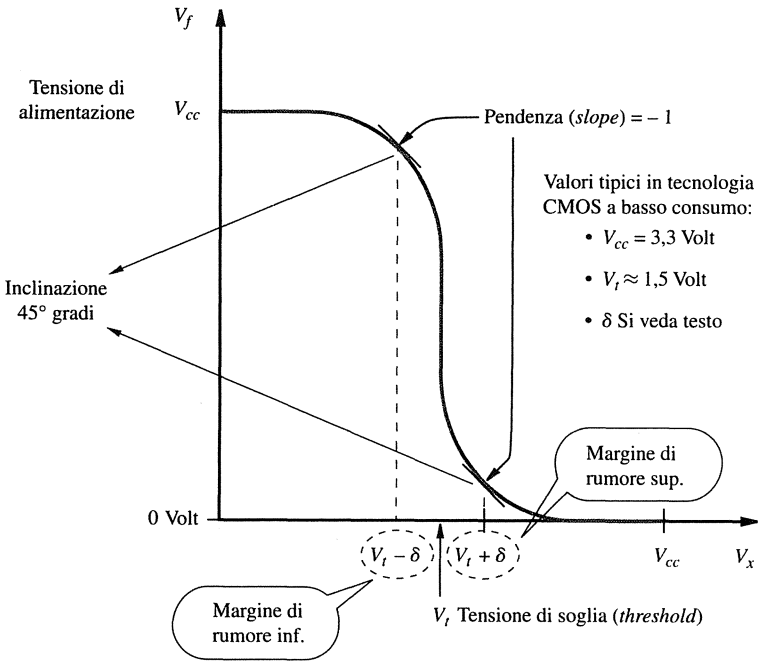


Figura A.20 Relazione tra tensione di ingresso e uscita nel negatore CMOS.

la tensione di uscita V_f in funzione di quella di ingresso V_x (si veda la Figura A.15) e mostra come la caduta sia brusca attraversando la tensione mediana $V_{cc}/2$ (si può convenire che la regione sia costituita dal tratto di curva dove la tangente ha *pendenza* (*slope*)[6] minore di -1, cioè più di 45°). Esistono un valore di tensione di *soglia* (*threshold*) V_t e un numero $\delta > 0$ vicino a 0 tali che se $V_x < V_t - \delta$ allora $V_f \approx V_{cc}$, mentre se $V_x > V_t + \delta$ allora $V_f \approx 0$. Ciò indica che non occorre che la tensione di ingresso V_x sia esattamente 0 o V_{cc} per produrre la tensione di uscita corretta, il che è una buona notizia perché anche se c'è rumore o disturbo la tensione di uscita si mantiene al valore logico giusto. Il massimo valore di disturbo tollerabile è chiamato *margine di rumore* (*noise margin*) e vale appunto $V_{cc} - (V_t + \delta)$ se in ingresso c'è il valore logico 1, oppure $V_t - \delta$ se in ingresso c'è il valore logico 0. Siccome in genere $V_t \approx V_{cc}/2$ e $\delta \ll V_t$, si ha che i due margini sono di poco superiore e inferiore a $V_{cc}/2$, rispettivamente. Pertanto la tecnologia CMOS esibisce margine di disturbo eccellente. La trattazione della tecnologia CMOS potrebbe continuare quasi all'infinito, ma è materia di elettronica e pertanto si rimanda alla letteratura scientifica di settore.

A.5.2 Ritardo di propagazione

Il circuito logico non è in grado di commutare istantaneamente da uno stato di funzionamento a un altro. Si esprime la *frequenza di lavoro* (*operating frequency*) del circuito dicendo quante

[6] La derivata prima della curva, ovvero il coefficiente angolare della tangente.

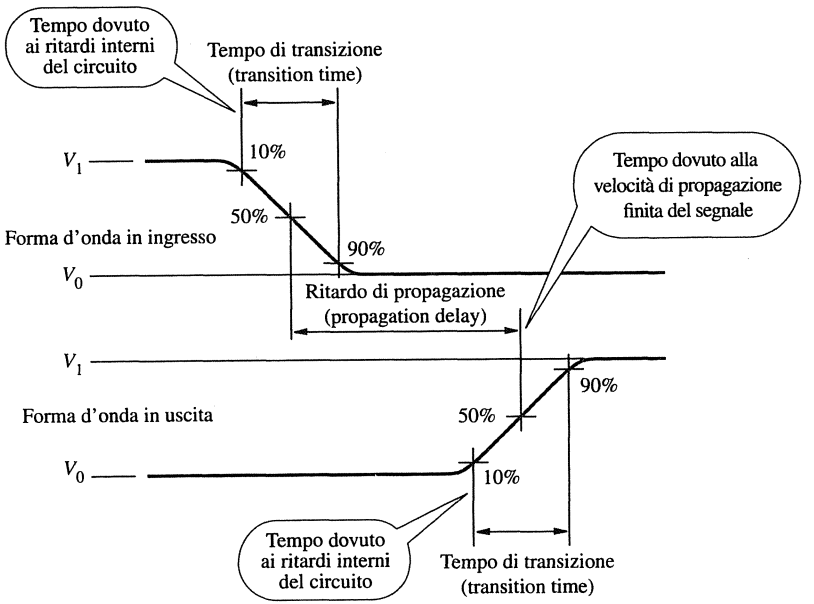


Figura A.21 Definizione e terminologia temporale del segnale.

volte esso commuta stato nell'unità di tempo, per esempio un secondo. Alla frequenza di commutazione si associa un secondo parametro caratteristico, il *ritardo di propagazione* (*propagation delay*) esemplificato in Figura A.21. Quando la configurazione in ingresso muta occorre un po' di tempo prima che l'uscita si porti al nuovo valore associato alla nuova configurazione, giacché tutti i segnali interni ritardano per propagarsi dagli ingressi all'uscita. Si suole misurare il ritardo di propagazione tra i punti mediani delle transizioni nelle forme d'onda di ingresso e uscita, come mostra la Figura A.21. Un terzo parametro importante è il *tempo di transizione* (*transition time*), che di solito si prende tra il 10% e il 90% dell'intervallo di tempo impiegato dal segnale per transitare di livello in modo completo, come mostra sempre la Figura A.21.

Il ritardo di propagazione è dato dai percorsi che collegano gli ingressi all'uscita. Il ritardo di ciascun percorso è la somma dei ritardi degli elementi circuitali attraversati. Il percorso più lento si dice *critico* (*critical path*) e determina il ritardo massimo. La frequenza massima a cui si può fare lavorare il circuito diminuisce all'aumentare del ritardo di propagazione massimo.

A.5.3 Vincoli di fan-in e fan-out

Il numero di ingressi di cui la porta logica è dotata è spesso indicato con il termine *fan-in*[7]; per esempio, le porte AND e OR di base hanno due ingressi, pertanto fan-in pari a 2. Il numero di ingressi collegabili in parallelo all'uscita della porta logica si chiama *fan-out*[8]. Le porte lo-

[7] Da *fan*, *ventaglio*; alla lettera significa il *ventaglio* (di fili) *in ingresso*.
[8] *Ventaglio* (di fili) *in uscita*.

giche reali non ammettono fan-in e fan-out elevati perché entrambi incidono negativamente sul ritardo di propagazione della porta e dunque sulla frequenza di lavoro massima del circuito.

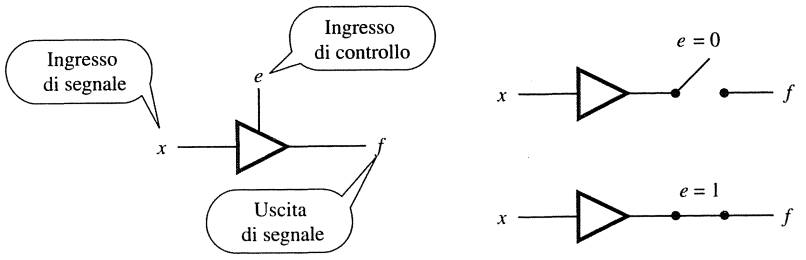
Inoltre ogni transistore MOS possiede una certa capacità (elettrica) parassita; più transistori ci sono più cresce la capacità parassita complessiva del circuito, la frequenza di lavoro massima diminuisce e la sensibilità ai disturbi aumenta, ovvero anche il margine di rumore diminuisce. Di solito bisogna limitare sia fan-in sia fan-out della porta, tipicamente a meno di 10. Se è necessario avere ingressi più numerosi di quanti non ne conceda il fan-in massimo bisogna aumentarne il numero tramite cascata o albero di porte, come mostra la Figura A.9. Se è necessario collegare in uscita alla porta più ingressi di altre porte (di qualunque tipo) di quanti non ne conceda il fan-out massimo bisogna duplicare o replicare la porta e spartire i collegamenti tra le varie porte operanti in parallelo.

A.5.4 Porta tri-state (o in alta impedenza)

Riesaminando gli schemi circuitali delle porte logiche illustrate finora si vedrebbe che è vietato collegare insieme le uscite di due porte. Il motivo è duplice: in primo luogo non si capirebbe quale valore logico dovrebbe uscire se una delle due porte emettesse 0 e l'altra 1; in secondo luogo (e in realtà conta anche di più) se si lavorasse con la tecnologia CMOS potrebbe accadere di stabilire continuità elettrica diretta tra alimentazione e massa causando cortocircuito e danneggiando seriamente il dispositivo. Ciò succederebbe proprio quando le due porte fossero in contraddizione, come detto sopra, giacché la prima avrebbe continuità tra uscita e massa, la seconda tra uscita e alimentazione e dunque l'insieme ottenuto collegando le uscite la stabilirebbe direttamente tra alimentazione e massa.

Tuttavia, nel calcolatore è frequentissimo che un componente prenda il segnale in ingresso da un numero anche elevato di possibili sorgenti che glielo mandano. Si può ovviare al problema interponendo il blocco funzionale multiplatore, che verrà spiegato nel paragrafo A.10, ma è una soluzione piuttosto costosa. C'è un'alternativa fornita da un tipo speciale di porta, chiamata *porta tri-state* (*tri-state buffer*). Tale porta ha tre stati di funzionamento (invece dei soliti due): emettere 1 o 0 (semplicemente lasciando passare il valore logico presente in ingresso), oppure porre l'uscita in stato (elettrico) di *alta impedenza*, che significa scollegarla elettricamente da tutto ciò che viene a valle.

La Figura A.22 illustra la porta tri-state. Essa ha due ingressi e un'uscita, come mostra la Figura A.22a. L'ingresso e o di *abilitazione* (*enable*) controlla la porta: se $e = 1$ l'uscita f assume lo stesso valore logico presente sull'ingresso x, cioè $f = x$; se $e = 0$ l'uscita f è tenuta in stato di alta impedenza, che si rappresenta con il simbolo Z, vale a dire $f = \mathrm{Z}$. In stato Z la resistenza equivalente al morsetto di uscita (si pensi al teorema di Thévenin nell'Elettrotecnica) diventa molto elevata, cioè l'uscita è isolata elettricamente. La Figura A.22b mostra il modello logico (a interruttori) della porta; il triangolo (somigliante al negatore ma mancante del pallino di inversione) rappresenta l'elemento funzionale che emette il valore di ingresso senza sottoporlo ad alcuna elaborazione; esso però *rinforza* in senso elettrico il segnale conferendogli potenza bastante a pilotare un certo numero di ingressi di altre porte collegate a valle di quella tri-state; insomma rigenera il fan-out del segnale. Il tutto si comporta come indica la tabella di verità di Figura A.22c (dove compare anche il simbolo Z). Infine la Figura A.22d mostra la realizzazione fisica della porta tri-state. La coppia di transistori NMOS e PMOS collegati in parallelo in uscita al circuito funziona come interruttore pilotato tramite il segnale e in forma diretta e negata (si deve ricordare che i due transistori sono complementari): se $e = 0$ entrambi i transistori sono in interdizione, la coppia funziona come interruttore aperto e l'uscita è in stato Z; se $e = 1$ entrambi sono in conduzione e l'uscita viene pilotata attivamente a 0 o 1. Il rinforzo elettrico del segnale è ottenuto facendolo passare tra due negatori in serie (si annullano), che non hanno effetto logico ma immettono nuova potenza nel segnale.

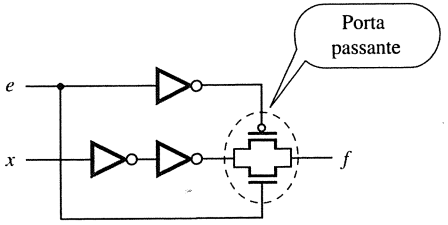


(a) Rappresentazione grafica

(b) Schema logico

e	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1

(c) Tabella di verità



(d) Schema elettrico

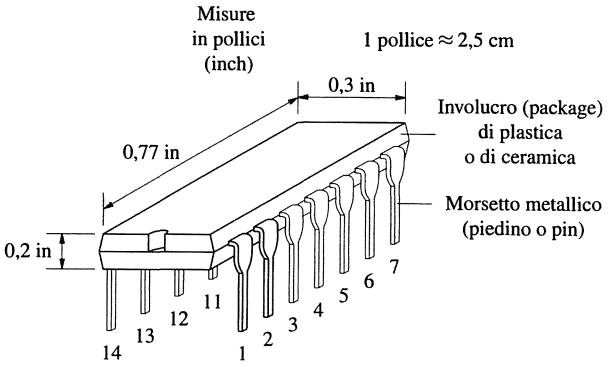
Figura A.22 Porta tri-state (o in alta impedenza).

La porta tri-state è utile anche per aumentare il fan-out quando sia subissato dal numero elevato di porte da collegare a valle di quella corrente. Ci sono porte tri-state di varia potenza, anche grande, in grado di elevare di molto il fan-out; a tale fine i due negatori in serie nella porta sono spesso costituiti da transistori di taglia maggiore del normale.

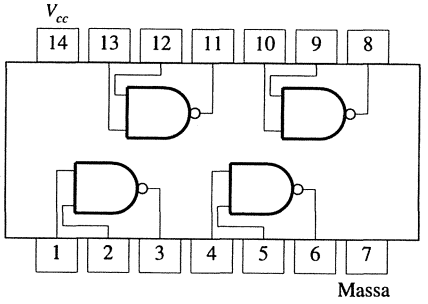
Ci si potrebbe chiedere perché l'interruttore finale sia formato da due transistori complementari invece di uno solo, come si è sempre fatto finora. La risposta è che il transistore di tipo N lascia passare bene il valore logico 0 ma male 1, e quello di tipo P fa l'opposto; in parallelo essi funzionano in modo soddisfacente per entrambi i valori. Comunque si tratta di un aspetto puramente elettrico, benché importante. La coppia di transistori così fatta viene talvolta chiamata *porta passante* (*pass transistor*).

A.5.5 Circuito integrato

Nei paragrafi precedenti si è discusso come realizzare circuitalmente svariate porte e funzioni logiche. Nella pratica attuale le porte si trovano raramente in forma isolata, piuttosto i circuiti logici sono raggruppati in *componenti integrati* (*integrated circuit*, IC) disponibili sul mercato. Il circuito integrato venne introdotto nei primi anni '60 e la tecnologia progredì rapidamente dando luogo a forme standard per una varietà notevole di componenti di base. Il circuito integrato in sé è una sottile piastrina di silicio, ma si presenta sempre incapsulato in un *involucro* (*package*) protettivo dotato di morsetti metallici o *piedini* (*pin*) esterni per collegarlo. Gli involucri seguono forme standard con numero e disposizione stabiliti di morsetti. La Figura A.23 illustra il componente integrato ospitante quattro porte NAND; sopra si vede l'involucro



(a) Aspetto esteriore



(b) Schema logico di circuito integrato contenente quattro porte NAND a due ingressi ciascuna

Figura A.23 Aspetto esteriore e schema logico di circuito integrato.

e sotto la corrispondenza tra porte e morsetti, dove quelli di massa e alimentazione sono in comune tra tutte le porte. Il componente integrato ospitante qualche porta logica soltanto (da una a dieci) è detto a *scala di integrazione piccola* o SSI (*Small Scale of Integration*).

I componenti di famiglia SSI danno ben poca funzionalità di elaborazione rispetto alla taglia fisica e inoltre la tipologia di morsetti esterni ne limita la prestazione. Per esempio, la porta NAND in tecnologia CMOS integrata su componente di famiglia SSI potrebbe presentare ai morsetti esterni ritardo di propagazione di circa 5 ns, come caso tipico; internamente però il ritardo potrebbe essere non superiore a 0,2 ns o qualcosa di simile; la differenza è tutta attribuibile ai morsetti. I componenti odierni integrano un numero di porte logiche enormemente superiore a quello esibito dalla famiglia SSI e con prestazione ben più elevata. Il componente integrato di famiglia a *scala di integrazione media* o MSI (*Medium Scale of Integration*) può contenere unità funzionali come addizionatore, sottrattore, registro di qualunque tipo, decodificatore e codificatore, multiplatore e altro ancora; quello di famiglia a *scala di integrazione grande* o LSI (*Large Scale of Integration*) unità come la ALU, il banco di registri o un piccolo processore; infine quello di famiglia a *scala di integrazione molto grande* o VLSI (*Very Large Scale of Integration*) memoria molto capace e processore potente; i processori moderni di uso

generale sono tutti di tipo VLSI. Comunque ciò non significa che siano spariti i componenti integrati ospitanti un qualche assortimento di porte logiche varie per uso libero da parte del progettista,[9] anzi oggi spesso tali componenti integrati multiuso sono perfino riconfigurabili. Nei prossimi paragrafi verrà presentata una carrellata di blocchi funzionali logici di vario genere, di famiglia essenzialmente MSI e in parte anche LSI, fino a quelli riconfigurabili.

A.6 Bistabili e flip-flop

La maggioranza dei circuiti logici deve essere capace di tenere memoria di informazioni di vario genere. Per esempio, il meccanismo di apertura e chiusura per armadietto, cassetta di sicurezza e simili, spesso ha circuito di controllo elettronico ed è ovvio che debba essere in grado di ricordare (a tempo indefinito) la combinazione numerica di apertura. Naturalmente un altro esempio, ben più importante e articolato, è dato dalla memoria del calcolatore, che deve essere in grado di mantenere tutto il complesso di informazioni costituenti programmi e dati.

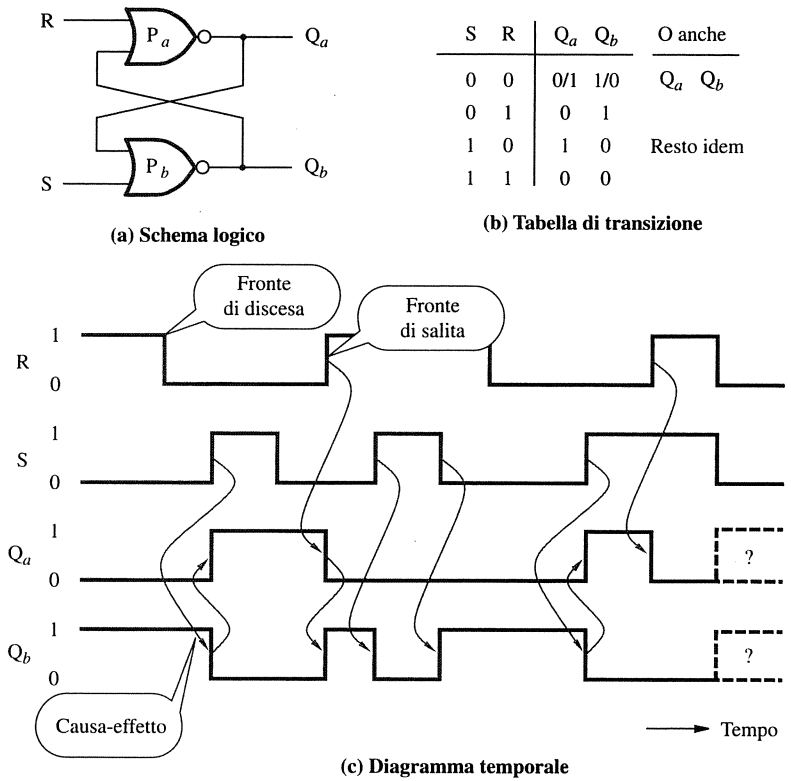
L'elemento circuitale fondamentale per tenere memoria di un bit di informazione è il *bistabile* (*latch*). Nei paragrafi precedenti sono state considerate solo reti combinatorie, che hanno struttura a strati: lo strato s^{esimo} manda segnale agli strati $(s + i)^{esimi}$ $(s, i \geq 1)$. Lo schema logico delle rete combinatoria *non può contenere cicli*; infatti le varie procedure di sintesi di funzioni logiche non danno mai origine a reti cicliche di porte logiche. Per avere capacità di memoria occorre generalizzare tale schema e passare a una famiglia di reti di porte logiche più ampia di quella combinatoria; esse contengono anche cicli, si chiamano *reti sequenziali* e sono nettamente più complesse da studiare delle reti combinatorie.

A.6.1 Bistabile asincrono

Occorre pertanto procedere per gradi e il bistabile è appunto l'esempio più semplice di rete sequenziale. Si consideri la coppia di porte logiche NOR retroazionate mostrata in Figura A.24a. L'effetto della retroazione è di fare rientrare l'uscita della porta NOR in ingresso alla porta stessa che la genera, sia pure passando attraverso un'altra porta (qui dello stesso tipo). Bisogna dunque esaminare il funzionamento di tale nuovo tipo di circuito. Si parta dalla situazione dove gli ingressi hanno valori logici R = 1 e S = 0. Propagando tali valori attraverso le porte logiche fino a raggiungere le uscite Q_a e Q_b si vede facilmente che risulta $Q_a = 0$ e $Q_b = 1$. Per dimostrarlo si ragiona così: dato che la porta NOR P_a ha l'ingresso R posto a 1, l'uscita Q_a deve valere 0 (e non importa sapere che cosa valga il secondo ingresso di P_a, giacché con 1 in ingresso NOR emette certamente 0 in uscita); l'uscita Q_a è retroazionata in ingresso alla porta NOR P_b; allora P_b ha ingressi posti entrambi a 0 e l'uscita Q_b vale 1 (qui è essenziale avere appurato che entrambi gli ingressi di P_b siano a 0); l'uscita Q_b è retroazionata in ingresso alla porta NOR P_a e così si scopre che P_a ha in ingresso R e Q_b, entrambi di valore 1. È importante osservare come dati gli ingressi si riescano a stabilire univocamente le uscite, sia pure propagando i segnali lungo percorsi circolari; finora dunque non c'è niente di essenzialmente nuovo rispetto a come si comporta la rete combinatoria.

Ora, se R viene portato a 0 lasciando S a 0 (cosicché si ha S = R = 0) l'uscita Q_a della porta NOR P_a continua a valere 0, giacché, come appena detto, il secondo ingresso di P_a è Q_b e vale 1 (e con 1 in ingresso NOR emette certamente 0 in uscita). Se poi si porta S a 1 lasciando R a 0, ragionando in modo analogo a quanto fatto prima si può verificare che Q_a

[9] La cosiddetta *glue logic*, cioè porte logiche sfuse usate per attaccare insieme (*glue*, *colla*) componenti con interfacce non del tutto compatibili.



S	R	Q_a	Q_b	O anche
0	0	0/1	1/0	Q_a Q_b
0	1	0	1	
1	0	1	0	Resto idem
1	1	0	0	

Figura A.24 Bistabile asincrono di tipo SR in versione NOR.

e Q_b si portano a 1 e 0, rispettivamente; ciò si potrebbe anche dedurre osservando che il circuito ammette asse di simmetria orizzontale e che riflettendolo rispetto a tale asse si scambiano P_a con P_b, R con S e Q_a con Q_b, rispettivamente. Se infine si porta S a 0 lasciando R a 0 (cosicché si ha di nuovo S = R = 0) l'uscita Q_a della porta NOR P_a continua a valere 1 per lo stesso ragionamento fatto prima con la medesima configurazione di ingresso. Si noti bene che ci sono due situazioni in cui si è posto S = R = 0, ma nella prima si è avuto $Q_a = 0$ mentre nella seconda $Q_a = 1$.

Si può dunque tentare di riassumere sotto forma di tabella tutto il comportamento così analizzato, come mostra la Figura A.24b. Si nota subito che si ha $Q_b = \overline{Q}_a$ tranne quando S = R = 1, sicché Q_b non contiene informazione aggiuntiva rispetto a quanta non ne dia già Q_a. La tabella associa le configurazioni di uscita alle configurazioni di ingresso, *ma non è una tabella di verità: nella prima riga (dove* S = R = 0*) l'uscita* Q_a potrebbe valere 0 o 1, giacché, come visto sopra, entrambe le situazioni sono possibili. Non si può stabilire che cosa valga Q_a solo sulla base della configurazione di ingresso corrente (come basterebbe per la rete combinatoria), bisogna anche conoscere qualcosa del comportamento passato del circuito, cioè come ingressi e uscite siano transitati di valore in precedenza. Pertanto la tabella si

chiama tabella di transizione (*transition table*). Tale fenomeno è caratteristico della rete sequenziale, la distingue dalla rete combinatoria e rappresenta appunto il *possesso di memoria*. Si dice che un circuito possiede memoria se il suo comportamento dipende dagli ingressi correnti e anche, in modo più o meno forte, da quelli precedenti.

La Figura A.24b mostra come denotare la tabella di transizione: nella prima riga, dove R = S = 0, le uscite Q_a e Q_b potrebbero valere 0 o 1 e 1 o 0, rispettivamente; pertanto si riportano come valori di uscita ancora i simboli Q_a e Q_b per indicare che le due uscite non hanno valore definito univocamente, bensì mantengono quello assunto in precedenza, qualunque fosse; nelle altre righe, invece, il valore delle uscite è determinato e lo si può indicare esplicitamente.

Siccome la rete sequenziale ha comportamento in parte dipendente dal passato, ovvero dal tempo, spesso se ne rappresenta il funzionamento tramite diagramma temporale come fatto in Figura A.24c per il bistabile in esame. La prima parte del diagramma riproduce proprio l'analisi fatta sopra. Gli istanti di tempo in cui le forme d'onda rappresentanti il livello 0 o 1 del segnale transitano di valore si chiamano *fronti di transizione*, di salita o discesa secondo il caso. Il diagramma mette anche in evidenza tramite frecce grige le *relazioni causa-effetto* intercorrenti tra i vari fronti di transizione di segnale (si veda al riguardo anche il Capitolo 7 in merito al bus del calcolatore, dove tale notazione è usata ampiamente).

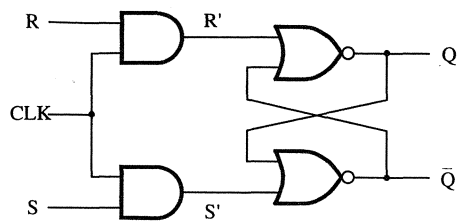
I nomi S e R stanno per *set* e *reset*, ovvero *ingresso di impostazione* (a 1) e *di ripristino* (a 0); è convenzione comune chiamare Q (in Figura A.24a è l'uscita Q_a) l'uscita del bistabile intendendone la forma diretta (benché se serve si possa usare anche la forma negata che è inevitabilmente presente). Riassumendo il funzionamento del bistabile SR (bistabile set-reset), si vede quanto segue: attivando S a 1 si forza Q a 1 (si scrive 1 nel bistabile); attivando R a 1 si forza Q a 0 (si scrive 0 nel bistabile); lasciando sia S sia R disattivi a 0 l'uscita Q *mantiene il valore logico che aveva assunto in precedenza*, qualunque esso fosse. Ciò giustifica il nome *bistabile*, cioè circuito logico che a parità di ingressi ha due stati di funzionamento Q = 0 oppure Q = 1, vale a dire un bit, ed è capace di tenere memoria del bit corrente a tempo indefinito finché, manovrando gli ingressi, non si decida di cambiarlo attivamente. Il bistabile SR è la rete sequenziale più semplice capace di tenere memoria di un bit di informazione. Si dice *asincrono* in quanto privo di ingresso di clock e il cambiamento di stato avviene (eventualmente) quando transitano gli ingressi.

Il diagramma temporale di Figura A.24c mostra che se si attivano R e S e poi si tenta di disattivarli simultaneamente, allora, secondo che la discesa di R precedesse o seguisse di poco quella di S (non può esistere simultaneità esatta, non è fisicamente possibile), l'uscita Q si porterebbe a 1 o 0, rispettivamente, in modo imprevedibile. Tale comportamento si accorda male con quello di tenere memoria di un bit, come sarebbe la funzione desiderata, pertanto al bistabile SR non andrebbe mai applicata la configurazione di ingresso R = S = 1.

A.6.2 Bistabile sincrono

In numerose applicazioni occorre agganciare il comportamento del bistabile al segnale che scandisce il tempo. Si deve pertanto aggiungere al bistabile SR un nuovo ingresso, chiamato *clock*, per definire la suddivisione del tempo in cicli (di clock). Il bistabile risultante si dice di tipo *sincrono*, mentre la versione originale senza ingresso di clock si dice *bistabile asincrono*. La Figura A.25 illustra lo schema logico, la tabella di transizione e il diagramma temporale del bistabile SR in versione sincrona.

Si prenda in esame la Figura A.25a. Quando l'ingresso CLK (*clock*) ha valore logico 1 i segnali interni S′ e R′ coincidono con gli ingressi esterni S e R, rispettivamente, e se S e R vengono azionati il bistabile sincrono si comporta come quello asincrono. Quando invece



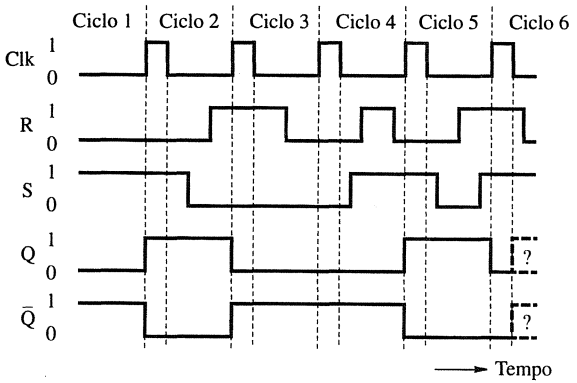
(a) Schema logico

Stato futuro

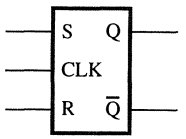
CLK	S	R	Q(t + 1)
0	x	x	Q (t) (Non cambia)
1	0	0	Q (t) (Non cambia)
1	0	1	0
1	1	0	1
1	1	1	x

Stato corrente

(b) Tabella di transizione



(c) Diagramma temporale



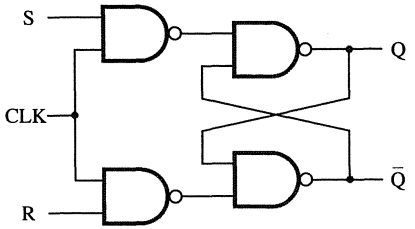
(d) Rappresentazione grafica

Figura A.25 Bistabile sincrono di tipo SR (con trasparenza).

l'ingresso CLK ha valore logico 0, i segnali interni S′ e R′ sono anch'essi senz'altro a 0 e in un certo senso lo stato del bistabile è bloccato al valore corrente, qualunque configurazione si applichi a S e R. Il comportamento del bistabile SR sincrono è dunque agganciato alla suddivisione del tempo in cicli. Anche il bistabile SR sincrono è una rete sequenziale.

La tabella di transizione del bistabile SR sincrono è simile a quella della versione asincrona, ma con l'aggiunta dell'ingresso CLK. La Figura A.25b mostra la nuova tabella, dove Q(t) e Q (t + 1) indicano lo *stato corrente* (o *stato presente*) del bistabile e quello che esso assumerà nel ciclo di clock consecutivo, chiamato anche *stato futuro* (o *stato prossimo*, *next state*); il simbolo t indica il ciclo di clock (t = 1, 2, 3, ...). Per le stesse ragioni esposte prima il comportamento del bistabile in presenza della configurazione di ingresso CLK = 1 e S = R = 1 è lasciato indefinito.

La Figura A.26 mostra come costruire il bistabile SR sincrono tramite porte NAND invece che NOR. Si lascia al lettore il compito di dimostrare che la versione NAND ha lo stesso comportamento di quella NOR, salvo scambiare le posizioni degli ingressi S e R. Infatti in Figura A.24 l'ingresso R va nella stessa porta da cui esce Q, mentre in Figura A.26 è S ad andare nella stessa porta di Q.

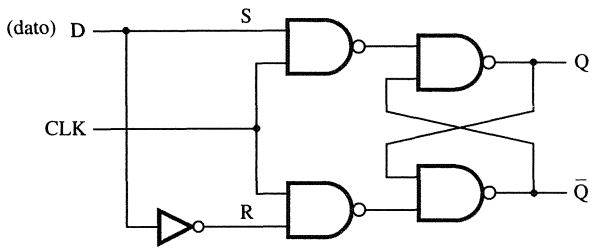


(a) Schema logico

CLK	S	R	Q(t + 1)
0	x	x	Q (t)
1	0	0	Q (t)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Tabella di transizione

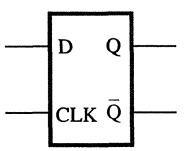
Figura A.26 Bistabile sincrono di tipo SR in versione NAND.



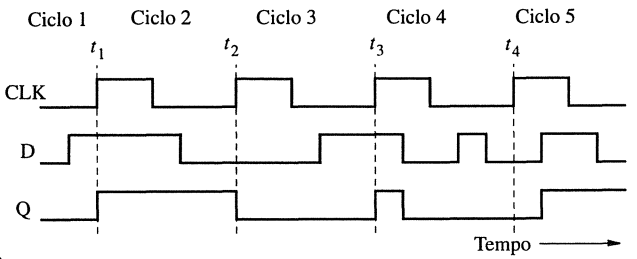
(a) Schema logico

CLK	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

(b) Tabella di transizione



(c) Rappresentazione grafica



(d) Diagramma temporale

Figura A.27 Bistabile sincrono di tipo D (con trasparenza).

I bistabili visti finora hanno sempre gli stessi ingressi di controllo S e R. La Figura A.27 illustra un secondo tipo di bistabile sincrono, con tipologia di ingresso diversa. I due ingressi S e R sono unificati in un solo ingresso chiamato D (che sta per *dato*); l'ingresso D viene inviato in forma diretta e negata sui due segnali interni che assolvono la funzione originaria di S e R, rispettivamente. È chiaro che si ha sempre R = S̄, cioè S e R sono complementari, e

inoltre S = D. Quando CLK = 1 l'uscita Q assume valore 1 o 0 se D = 1 o D = 0, rispettivamente, giacché è come dire S = 1 o R = 1. Ciò significa che quando l'ingresso di clock è alto (ovvero a 1) il bistabile di tipo D campiona il valore logico dell'ingresso D e lo tiene in memoria fino al ciclo di clock successivo.

A.6.3 Flip-flop master-slave

Spiegando il funzionamento del bistabile sincrono SR di Figura A.25 si è sempre fatta l'ipotesi che quando si ha ingresso di clock alto, cioè CLK = 1, gli ingressi S e R siano stabili; le transizioni di S e R avvenivano solo quando si aveva CLK = 0, ovvero quando lo stato del bistabile era bloccato. Esaminando il circuito si vede subito che se S e R transitano di valore mentre l'ingresso di clock è alto lo stato cambia subito e l'uscita Q transita immediatamente. Il bistabile sincrono di tipo D di Figura A.27 ha lo stesso difetto, che va sotto il nome di *trasparenza*: è come se quando si ha CLK = 1 il bistabile sia assente e non eserciti alcuna funzione di memoria, giacché mutamenti di ingresso si riflettono subito in mutamenti di uscita. Detto in altro modo durante l'intervallo di trasparenza il bistabile non esibisce comportamento bene agganciato alla suddivisione del tempo in cicli di clock giacché potrebbe cambiare stato più di una volta in un solo ciclo.

In numerose situazioni la trasparenza è un comportamento indesiderabile, soprattutto quando il bistabile è usato come componente di base per costruire registri e contatori. Per esempio nel registro a scorrimento si potrebbe avere la propagazione del bit attraverso due o più bistabili nel corso di un solo ciclo di clock, comportamento del tutto errato.

Il bistabile sincrono di tipo *master-slave* (o *a memoria ausiliaria*) risolve il problema; la Figura A.28a ne illustra la struttura. Si collegano in serie due bistabili sincroni di tipo D come quello visto in Figura A.27. Il bistabile a monte, noto come *master* o *bistabile principale*, prende il segnale esterno D e il segnale CLK in forma diretta; quello a valle, noto come *slave* o *bistabile ausiliario*, prende l'uscita del master e il segnale CLK in forma negata. Grazie a tale accorgimento master e slave non sono mai contemporaneamente in condizione di trasparenza, dunque non lo è neppure il bistabile master-slave complessivo.

Quando si ha CLK = 1 lo stato del master segue subito la transizione dell'ingresso esterno D, ma lo slave per il momento ignora il nuovo valore logico Q_m che il master gli manda. Lo slave ha appunto la funzione di tenere bloccata l'uscita $Q = Q_s$ per tutto l'intervallo di tempo in cui il master può campionare l'ingresso esterno D. Quando l'ingresso CLK scende da 1 a 0, lo stato corrente del master viene trasferito nello slave e va in uscita Q. Quando si ha CLK = 0 lo slave è in condizione di trasparenza, ma il master non lo è e dunque lo slave non è sensibile alle transizioni dell'ingresso esterno D. L'uscita Q può variare solo in corrispondenza del fronte di discesa dell'ingresso di clock o poco dopo a motivo del ritardo di propagazione delle porte logiche. Il diagramma temporale di Figura A.28b dà qualche esempio di transizione di stato. Si noti come il passaggio di ciclo avvenga sul fronte di discesa del clock.

Il bistabile sincrono la cui uscita può commutare solo in corrispondenza della transizione tra ciclo di clock corrente e consecutivo (transizione delimitata per esempio dal fronte di discesa dell'ingresso di clock) è comunemente chiamato *flip-flop*. La Figura A.28 illustra il flip-flop master-slave di tipo D a commutazione sul fronte di discesa. Negando il segnale di clock oppure distribuendone la forma negata e diretta a master e slave, rispettivamente, si ottiene la versione a commutazione sul fronte di salita (se ne lascia l'analisi al lettore).

Infine, la Figura A.28c mostra la rappresentazione grafica del flip-flop master-slave di tipo D. L'ingresso di clock è segnalato dal triangolo, la convenzione è standard. La presenza del pallino di negazione serve a precisare che il fronte dell'ingresso di clock dove l'uscita

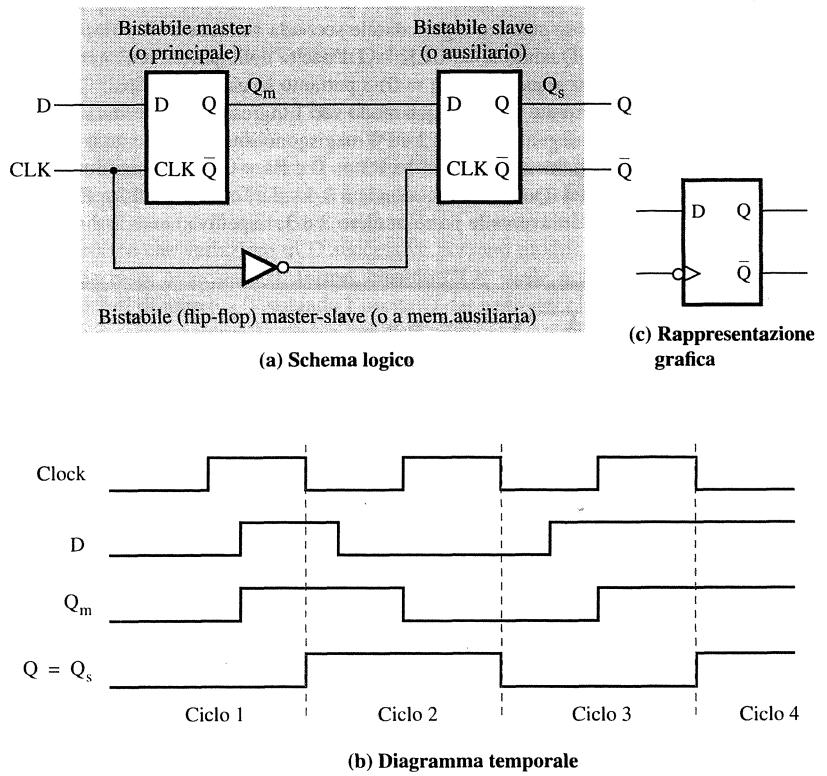


Figura A.28 Flip-flop di tipo D master-slave.

può commutare è quello di discesa; il triangolo da solo segnalerebbe quello di salita (anche questa convenzione è standard).

A.6.4 Flip-flop edge-triggered

Se il flop-flop può commutare l'uscita solo nell'istante del fronte di transizione dell'ingresso di clock si dice di tipo *edge-triggered* o a *sincronismo di fronte*. Nel flip-flop edge-triggered ingresso e uscita sono del tutto disaccoppiati l'uno dall'altra. Se la commutazione di uscita avviene su fronte di salita (CLK sale 0 → 1) il flip-flop edge-triggered si dice di tipo *positivo*, se su fronte di discesa (CLK scende 1 → 0) di tipo *negativo*. Affinché il flip-flop edge-triggered funzioni correttamente bisogna che il fronte attivo del segnale di clock sia netto e istantaneo.

Il flip-flop master-slave spiegato nel paragrafo precedente è di tipo edge-triggered negativo. Esso però può campionare la transizione di valore dell'ingresso D in un istante qualsiasi interno all'intervallo di tempo di attività dell'ingresso CLK. Si può tuttavia immaginare un nuovo tipo di flip-flop edge-triggered dove l'ingresso D sia valido e venga dunque campionato

esclusivamente quando l'ingresso di clock presenta fronte di transizione, e sullo stesso fronte venga mandato in uscita; per tutto il resto del ciclo di clock lo stato del flip-flop è bloccato.

La Figura A.29a mostra lo schema logico di tale seconda versione di flip-flop edge-triggered, piuttosto complessa. Quando si ha CLK = 1 l'uscita delle porte NOR numero 2 e 3 vale senz'altro 0 (vale a dire si ha P1 = P2 = 0) e pertanto le uscite principali Q e $\overline{Q}$ mantengono bloccato lo stato corrente in qualunque modo vari l'ingresso D. Si verifica facilmente però che in tale situazione i segnali interni P3 e P4 reagiscono immediatamente a transizioni di D tenendosi a $\overline{D}$ e D, rispettivamente, e si ha P3 = $\overline{D}$ e P4 = D; essi sono dunque complementari, avendosi P3 = $\overline{P4}$. Quando CLK scende a 0, i valori correnti di P3 e P4 passano, venendo negati, su P1 e P2 attraverso le porte numero 2 e 3, rispettivamente, e dunque si ha

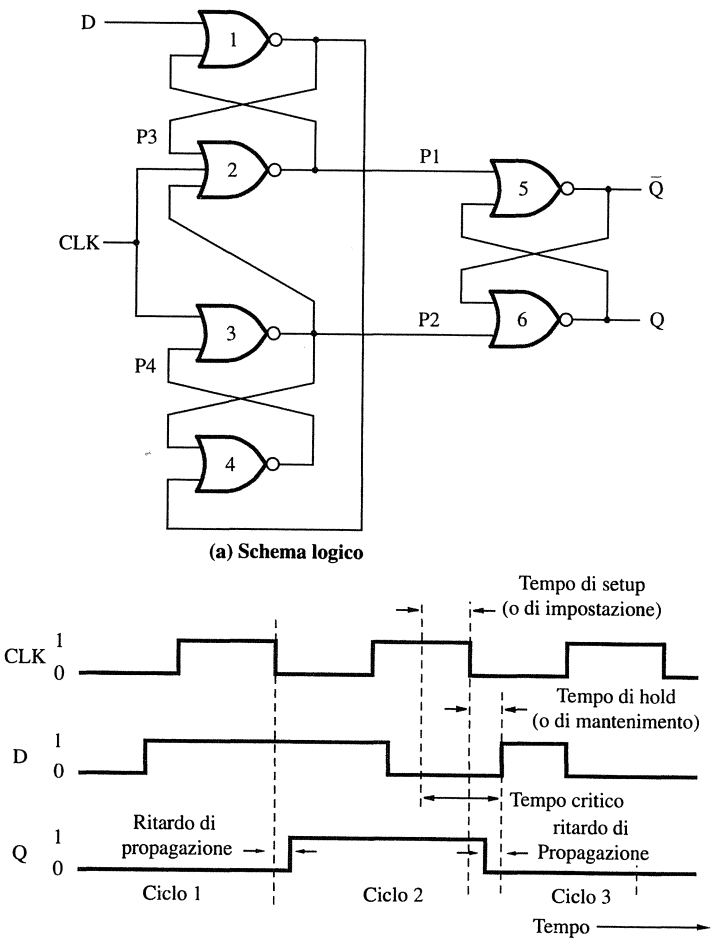


Figura A.29 Flip-flop di tipo D edge-triggered.

P1 = $\overline{P3}$ e P2 = $\overline{P4}$. Siccome comunque P3 e P4 sono complementari, si ha anche che P1 = $\overline{P3}$ = $\overline{\overline{P4}}$ = P4 = D e P2 = $\overline{P4}$ = P3 = $\overline{D}$ (si veda prima). Pertanto le porte NOR numero 5 e 6 (collegate e usate come se formassero un bistabile asincrono) acquisiscono D come nuovo stato e lo mandano in uscita Q. In particolare si ha che P2 = $\overline{D}$ e Q = $\overline{P2}$, dunque ne segue Q = $\overline{P2}$ = $\overline{\overline{D}}$ = D, cioè si memorizza proprio l'ingresso D.

Si può verificare come, dopo che CLK sia sceso a 0 e ormai si tenga stabile a livello basso, ulteriori transizioni dell'ingresso D non abbiano più effetto sui segnali interni P1 e P2. Ci sono due casi: se D valeva 0 quando CLK è sceso a 0 il bit 1 presente su P2 va in ingresso alle porte numero 2 e 4, così tenendo P1 e P2 a 0 e 1, rispettivamente, indipendentemente da variazioni di D successive; se D valeva 1 quando CLK è sceso a 0 il bit 1 presente su P1 impedisce che variazioni di D successive possano modificare l'uscita della porta numero 1, che resta bloccata a 0. Infine, quando CLK risale a 1 i segnali interni P1 e P2 vengono forzati di nuovo a 0 isolando l'uscita Q dal resto del circuito e i segnali interni P3 e P4 ricominciano a seguire le variazioni di D come si è argomentato prima. L'analisi del bistabile è così completa: solo il valore presente su D al fronte di discesa del clock importa, viene memorizzato nel bistabile e va subito in uscita Q; ogni altro valore di D è semplicemente ignorato.

La Figura A.29b mostra il diagramma temporale del flip-flop edge-triggered così descritto, di tipo negativo. Il valore logico presente su D immediatamente prima del fronte di discesa di CLK viene campionato, immesso come stato nel flip-flop e mandato in uscita; ogni altro valore di D viene ignorato. Tutto avviene dunque solo sul fronte di discesa. Tuttavia c'è un intervallo di *tempo critico* centrato sul fronte attivo (di discesa) dell'ingresso CLK durante cui l'ingresso D va tenuto stabile. L'intervallo critico è diviso in due parti: il *tempo di setup* o *di impostazione* (*setup time*) prima del fronte, e il *tempo di hold* o *di mantenimento* (*hold time*) dopo il fronte. Inoltre la commutazione dell'uscita Q segue il fronte attivo con breve ritardo, dovuto al tempo di propagazione delle porte logiche costituenti il flip-flop. Questo flip-flop edge-triggered ha comportamento ideale e poco sensibile a rumore (giacché il valore di D conta solo sul fronte di discesa del clock), ma è costoso in numero di porte logiche e pertanto non molto usato, preferendosi in genere la versione master-slave.

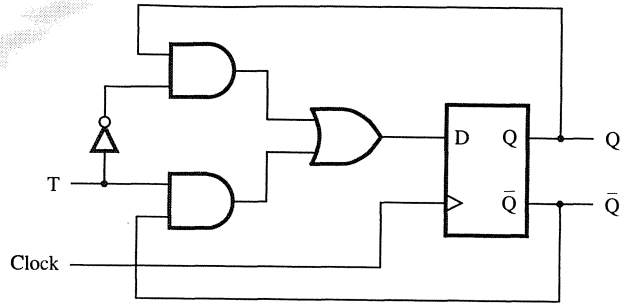
A.6.5 Flip-flop di tipo T

I flip-flop più diffusi sono quelli di tipo D e in genere sono usati per costruire registri ed elementi funzionali simili. Tuttavia, esistono applicazioni dove tornano utili altri tipi di flip-flop. Per esempio, il contatore di solito fa uso di flip-flop di tipo T, come si dirà nel Paragrafo A.8. Se l'ingresso T ha valore 1, il flip-flop di tipo T commuta stato a ogni ciclo di clock secondo la scansione del tempo imposta dall'ingresso di clock CLK.

La Figura A.30 illustra il flip-flop di tipo T. Lo schema logico mostrato in Figura A.30a deriva da quello del flip-flop di tipo D. Per giustificarlo basta osservare che quando si pone T = 1 si ha D = $\overline{Q}$ pertanto nel ciclo di clock successivo a quello corrente il flip-flop assume stato complementare a quello presente; se invece si pone T = 0, si ha D = Q e lo stato viene riconfermato. In Figura A.30b, A.30c e A.30d sono riportati la tabella di transizione, la rappresentazione grafica convenzionale e il diagramma temporale, rispettivamente. Questa volta il flip-flop è di tipo edge-triggered positivo (l'uscita commuta sul fronte di salita di CLK).

A.6.6 Flip-flop di tipo JK

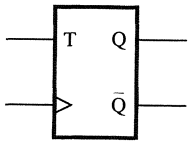
Un altro tipo di flip-flop occasionalmente utile è quello cosiddetto JK, che combina le caratteristiche dei flip-flop di tipo D e T; la Figura A.31 lo illustra. La tabella di transizione di Figura A.31b ne dà il funzionamento: le prime tre righe definiscono lo stesso comportamento



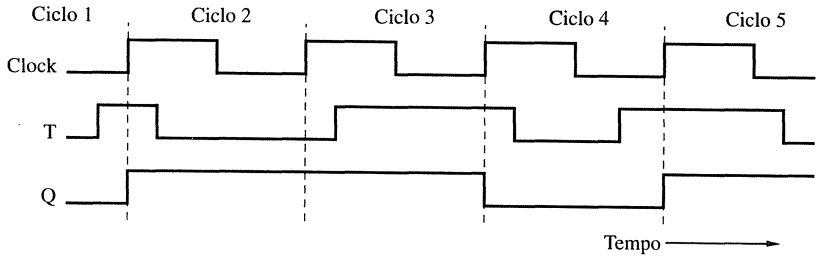
(a) Schema logico

T	Q(t + 1)
0	Q(t)
1	Q̄(t)

(b) Tabella di transizione



(c) Rappresentazione grafica



(d) Diagramma temporale

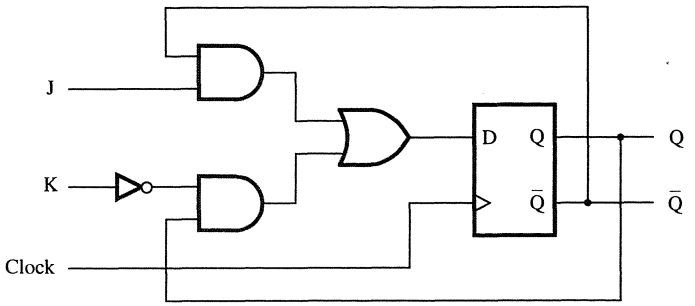
Figura A.30 Flip-flop di tipo T.

del flip-flop di tipo SR di Figura A.25b con la corrispondenza S ≡ J e R ≡ K. Quando si ha J = K = 1 (configurazione vietata per SR) lo stato prossimo è il complemento di quello corrente, proprio come fa il flip-flop di tipo T.

Si può facilmente convertire il flip-flop di tipo D in uno di tipo JK imponendo la relazione seguente tra ingresso e uscita (la cui analisi è lasciata al lettore):

$$D = J\bar{Q} + \bar{K}Q$$

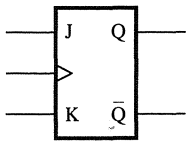
La Figura A.31a mostra lo schema logico del flip-flop di tipo JK, costruito riproducendo esattamente la relazione appena data. La Figura A.31c mostra la rappresentazione grafica convenzionale (è di tipo positivo). Il flip-flop JK è versatile: si può usare per costruire registri, come il flip-flop D, ma anche per costruire contatori perché se si unificano gli ingressi J e K cortocircuitandoli si comporta esattamente come il flip-flop T. Il lettore lo può verificare da sé tramite la relazione $D = J\bar{Q} + \bar{K}Q$ ponendo J = K = T.



(a) Schema logico

J	K	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q̄(t)

(b) Tabella di transizione



(c) Rappresentazione grafica

Figura A.31 Flip-flop di tipo JK.

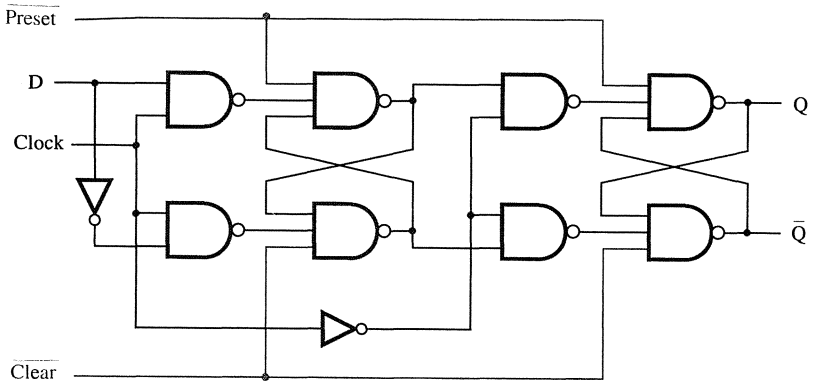
A.6.7 Flip-flop con preset e clear

Lo stato futuro (o prossimo) del flip-flop (quello che esso assumerà nel ciclo di clock consecutivo al ciclo corrente) è determinato dallo stato corrente e dalla configurazione di ingresso. Talvolta occorre forzare il flip-flop in uno stato definito indipendentemente dallo stato corrente e dagli ingressi normali. Per esempio, quando si accende il calcolatore bisogna mettere tutti i flip-flop in stato noto a priori; in pratica di solito ciò significa azzerarli, ma può darsi alcuni vadano invece posti inizialmente a 1.

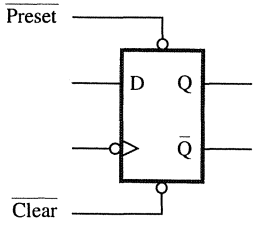
Si possono aggiungere due nuovi ingressi al flip-flop master slave: ingresso di *preset* o *precarica* per forzarlo a 1; ingresso di *clear* o *azzeramento* per forzarlo a 0. La Figura A.32 illustra il bistabile master-slave modificato con preset e clear (schema logico e rappresentazione grafica). I due nuovi ingressi agiscono indipendentemente dagli ingressi normali D e CLK (in pratica dunque agiscono in modo asincrono) e sono attivi bassi, cioè esercitano la rispettiva funzione portandoli a 0. Ovviamente però non vanno mai attivati insieme. Spesso si aggiungono funzioni di preset e clear anche ad altri tipi di flip-flop.

A.7 Registro seriale-parallelo (ser.-par. register)

Il *registro* (*register*) è un blocco funzionale *sequenziale* fondamentale per la costruzione del processore e del calcolatore. Esso può contenere una parola di lunghezza fissata (esistono registri per ogni lunghezza di parola comunemente usata) e in sostanza è costituito da una schiera di flip-flop, uno per ogni bit da tenere nel registro. Tutti i flip-flop sono collegati in parallelo



(a) Schema logico



(b) Rappresentazione grafica

Figura A.32 Flip-flop master-slave con preset e clear.

all'ingresso di clock del registro e pertanto sono sincronizzati. Si può scrivere nel registro una nuova parola o leggere la parola contenuta correntemente, che ne rappresenta lo stato. L'operazione di scrittura lavora in modo sincrono su tutti i flip-flop costituenti il registro.

Il calcolo richiede spesso le operazioni di scorrimento e rotazione della parola, pertanto occorrono i circuiti logici in grado di eseguirle. Il *registro a scorrimento* o *seriale* (*shift register*) permette in ogni ciclo di clock di fare scorrere il contenuto verso destra o sinistra di una posizione, secondo la scansione del tempo imposta dall'ingresso di clock. La Figura A.33 mostra lo schema logico del registro a scorrimento verso destra a quattro bit. Esso è costituito da una schiera di quattro flip-flop di tipo D collegati in serie; a ogni ciclo di clock il contenuto del flip-flop F_i viene trasferito nel flip-flop F_{i+1} ($1 \leq i < 4$), vale a dire la parola da quattro bit contenuta nel registro scorre verso destra di una posizione. Un nuovo bit entra a sinistra dall'ingresso seriale e uno esce a destra dall'uscita seriale. Retroazionando l'uscita seriale sull'ingresso seriale si ottiene il registro a rotazione.

Affinché il registro funzioni correttamente bisogna che la parola scorra esattamente di una posizione a ogni ciclo di clock. Ciò impone di usare il tipo di bistabile appropriato. Il semplice bistabile sincrono trasparente di tipo D, come quello illustrato in Figura A.27, è inadatto. Infatti, quando l'ingresso di clock è attivo (CLK = 1), il bistabile è in condizione di trasparenza e l'ingresso D si propaga rapidamente in uscita; chiaramente nel registro il bit

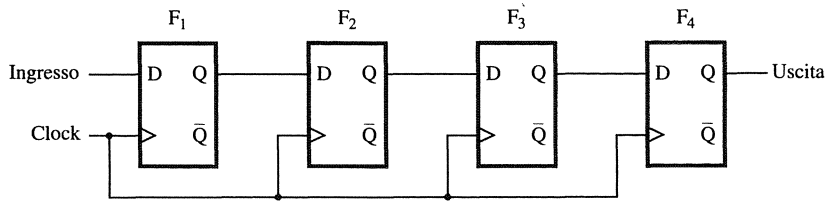


Figura A.33 Registro a scorrimento (shift register) di tipo semplice.

potrebbe percorrere in un solo ciclo di clock un certo numero di bistabili, vale a dire la parola scorrerebbe di due o più posizioni. Inoltre il numero di posizioni non sarebbe facilmente controllabile perché dipenderebbe dal ritardo di propagazione del singolo bistabile, che è un parametro tecnologico soggetto a una certa varianza. Si risolve il problema usando il flip-flop master-slave o edge-triggered.

Il registro a scorrimento con funzione di caricamento in parallelo e possibilità di leggerne l'intero contenuto è un circuito sequenziale di notevole versatilità e utilità. Per costruirlo basta prendere il registro a scorrimento semplice e aggiungergli un po' di porte logiche, come mostra la Figura A.34, dove si vede un nuovo ingresso di comando "$\overline{shift}\backslash load$". Il registro seriale-parallelo a quattro bit mostrato nella figura può funzionare in modi differenti: come registro a scorrimento verso destra (ponendo l'ingresso di comando $\overline{shift}\backslash load = 0$); come registro parallelo dove caricare in un solo ciclo di clock la parola da quattro bit presente sugli ingressi paralleli

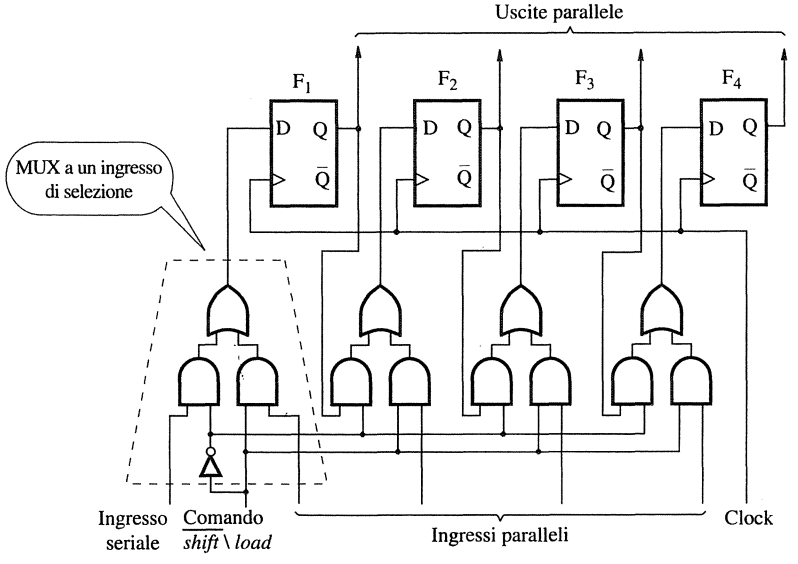


Figura A.34 Registro seriale-parallelo (serial-parallel register) a quattro bit.

(ponendo il comando $\overline{shift}\backslash load = 1$); come registro parallelo da cui leggere la parola da quattro bit contenuta (non occorre comando, il contenuto è sempre visibile sulle quattro uscite parallele). L'analisi del circuito è lasciata al lettore, ma come punto di partenza si può osservare che la rete combinatoria in ingresso a ciascun flip-flop è un multiplatore (multiplexer) a un ingresso di selezione e due ingressi di dato. Si potrebbe aggiungere il comando *hold* per tenere fisso il contenuto: se *hold* = 1 il contenuto è fisso, se *hold* = 0 il comando $\overline{shift}\backslash load$ opera come si è detto prima. Si potrebbe aggiungere anche la funzione di scorrimento verso sinistra aumentando ancora il numero di ingressi di comando (se ne lascia il compito al lettore, come esercizio), e magari aggiungere perfino una porta tri-state alle uscite parallele per abilitarle o disabilitarle. Talvolta il registro seriale-parallelo equipaggiato con tutte le funzioni ragionevoli di caricamento, scorrimento a destra e sinistra, abilitazione ecc. viene chiamato *registro universale* (*universal register*). Se ne trovano in commercio parecchie versioni, in realtà poco diverse le une dalle altre; in genere sono componibili in serie per ottenere registri di lunghezza in bit crescente.

A.8 Contatore binario (binary counter)

Il *contatore binario* (*binary counter*) è un blocco funzionale *sequenziale* importante quanto il registro, tanto che sarebbe impossibile sottovalutarne l'utilità. Il contatore serve naturalmente per contare, ma anche per scandire il tempo, per generare sequenze di segnali di controllo e per suddividere il segnale di clock a frequenza elevata in sottomultipli a frequenza minore, ovvero in generale per scalarlo.

La Figura A.35 illustra il contatore binario a tre bit o modulo 8, costruito utilizzando tre flip-flop di tipo T. Si ricordi che quando l'ingresso T vale 1 il flip-flop di tipo T commuta stato a ogni ciclo di clock. Esaminando la Figura A.35a si vede dunque che a ogni ciclo di clock l'uscita intermedia Q_0 commuta, cioè transita da 0 a 1 nel primo ciclo, da 1 a 0 nel secondo, da 0 a 1 nel terzo e così via. Pertanto la frequenza della forma d'onda di Q_0 è la metà della frequenza originale del segnale mandato all'ingresso di clock. Siccome Q_0 funziona come ingresso di clock del flip-flop successivo che ha anch'esso ingresso T tenuto a 1, la forma d'onda in uscita su Q_1 ha frequenza dimezzata rispetto a Q_0 e ridotta a un quarto della frequenza originale di clock. Similmente, Q_2 funziona a frequenza un ottavo e così via se il contatore fosse a più di tre bit. Il diagramma temporale di Figura A.35b mostra gli andamenti delle uscite dei flip-flop nell'ipotesi che essi commutino sul fronte di salita del rispettivo ingresso di clock, nonché le relazioni di causa-effetto che le legano (frecce grige).

Tale schema di contatore binario viene anche detto *a propagazione* (*ripple counter*) perché i fronti dell'ingresso di clock si propagano a onda lungo la catena di flip-flop, ma dimezzando la frequenza a ogni flip-flop attraversato. Chiaramente il numero binario a tre bit che rappresenta lo stato del contatore percorre la sequenza 000, 100, 010, ..., 011, 111, cioè (con significatività dei bit crescente verso destra) in decimale 0, 1, 2, ..., 6, 7, e poi ricircola a 0 riprendendo il conteggio; per tale motivo si dice che il contatore è modulo $2^3 = 8$ a conteggio verso l'alto. Se ogni flip-flop ha ritardo di propagazione pari a Δ, l'uscita Q_2 subisce ritardo cumulativo pari a 3Δ, come mostra appunto il diagramma temporale. Il ritardo cumulativo potrebbe diventare eccessivo se il numero di bit fosse elevato e costringerebbe a diminuire la frequenza di clock, ma in generale per contatori di dimensione normale rimane abbastanza contenuto. La generalizzazione del contatore a $n \geq 1$ bit (pertanto modulo 2^n) è piuttosto ovvia.

Il modello di contatore di Figura A.35 è asincrono, perché non tutti i flip-flop sono collegati in parallelo all'ingresso di clock esterno, ma solo quello più a sinistra (il primo). Dal diagramma temporale si vede del resto che le commutazioni dei flip-flop sono leggermente ritardate l'una rispetto all'altra, il che va contro il sincronismo. Al costo di qualche porta logica aggiuntiva si può ottenere il contatore sincrono, i cui flip-flop commutano simultaneamente e che pertanto può funzionare a frequenza maggiore di quello asincrono. Si può provare a progettarne uno retroazionando un registro sincrono parallelo a $n \geq 1$ bit su un incrementatore, vale a dire un circuito aritmetico che riceve in ingresso dal registro un numero intero naturale a n bit e lo emette in uscita (sempre a n bit) con valore aumentato di uno, per rimandarlo in ingresso al registro e dunque ricaricarlo lì nel ciclo di clock successivo, e così via (si veda anche il Capitolo 9); il registro può avere un ingresso di ripristino (reset) per azzerarne il contenuto all'inizio del conteggio; a ogni ciclo di clock il contenuto del registro si incrementa di uno, in modo perfettamente sincrono con il clock, e arrivato al valore $2^n - 1$ riparte da 0 (contatore ricircolante modulo 2^n); in altre versioni il conteggio si ferma al tetto massimo (contatore bloccante) e bisogna dare il comando di ripristino (reset); per concludere, il riporto in uscita all'incrementatore segnala quando il conteggio ha raggiunto il tetto massimo (dove o ricircola o si ferma). Il lettore può esercitarsi nel mettere a punto queste e altre varianti ragionevoli.

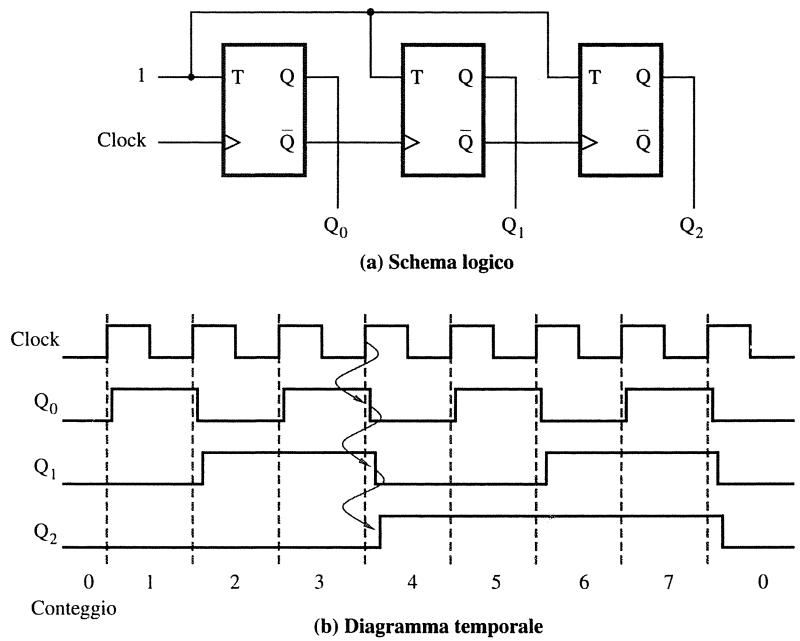
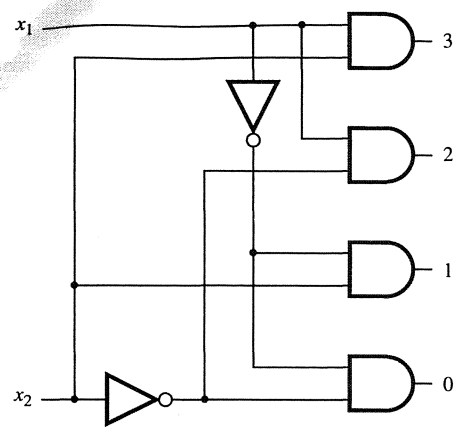


Figura A.35 Contatore binario (binary counter) verso l'alto modulo 8.

A.9 Decodificatore (decoder)

Gran parte dell'informazione che il calcolatore elabora è rappresentata in forma codificata. Per esempio, se l'istruzione macchina ha codice operativo da n bit si possono specificare fino a 2^n operazioni differenti. Tuttavia per identificare l'operazione da eseguire occorre decodificare il codice operativo. Il *decodificatore* (*decoder*) da $n \geq 1$ bit è un blocco funzionale



(a) Schema logico

x_1	x_2	Uscita attiva
0	0	0
0	1	1
1	0	2
1	1	3

(b) Tabella di verità (semplificata)

Figura A.36 Decodificatore (decoder) a due ingressi.

combinatorio dotato di n ingressi e 2^n uscite, che si possono immaginare come se fossero numerate in decimale da 0 a $2^n - 1$. Il decodificatore attiva la linea di uscita corrispondente al numero binario da n bit presente sugli ingressi.

La Figura A.36a mostra lo schema logico del decodificatore binario a $n = 2$ bit: i due ingressi x_1 e x_2 specificano quale linea di uscita attivare (portandola a 1); a lato la tabella di verità di Figura A.36b mostra il comportamento complessivo del decodificatore; l'analisi dello schema logico è lasciata al lettore.

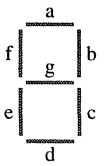
Esistono altri tipi utili di decodificatore. Per esempio spesso occorre decodificare la cifra decimale rappresentata in codice BCD a quattro bit attivando selettivamente una tra 10 uscite. Altre volte si deve decodificare la cifra decimale BCD generando l'insieme di segnali necessari per pilotare il display a sette segmenti da una cifra, come mostra la Figura A.37.

Il display è formato da sette segmenti illuminabili disposti in modo opportuno, nominati $a, b, ..., f$. Ciascun segmento è pilotato (acceso o spento) da un segnale di controllo individuale. Per esempio, se $a = 0$ o $a = 1$ il segmento a è spento o illuminato, rispettivamente. La tabella di verità in Figura A.37 definisce lo schema di accensione adatto per ciascuna cifra BCD in ingresso e in Figura A.37c è mostrato lo schema logico della rete combinatoria risultante, sintetizzata tramite porte NAND e NOT. Il lettore può esaminare la rete a verificare che corrisponde alla tabella di verità.

A.10 Multiplatore (multiplexer)

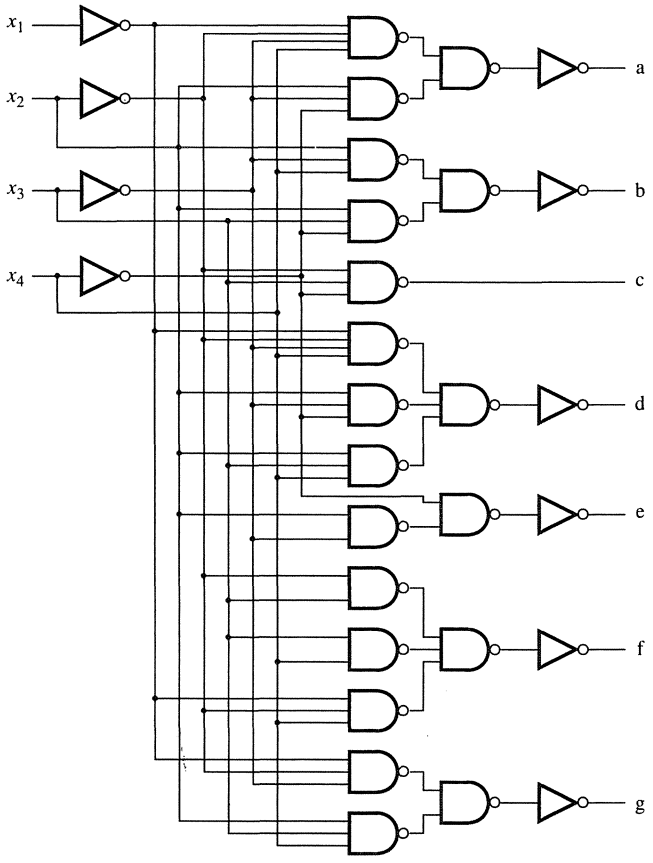
Il decodificatore seleziona la linea di uscita da attivare in funzione della configurazione di ingresso. La linea di uscita selezionata viene attivata a 1, mentre tutte le altre sono tenute disattive a 0. Esistono anche altri blocchi funzionali dotati di capacità di selezione. Il *multiplatore* (*multiplexer*) è un blocco funzionale combinatorio dotato di $n \geq 1$ ingressi di selezione, di 2^n ingressi di dato e di un'uscita di dato. La Figura A.38 illustra il multiplatore a $n = 2$ ingressi di selezione e 4 ingressi di dato.

(a) Display a sette segmenti



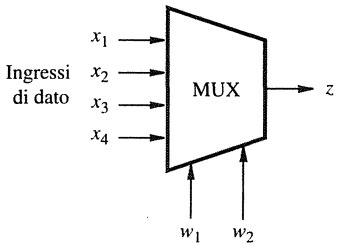
No.	x_1 x_2 x_3 x_4	a	b	c	d	e	f	g
0	0 0 0 0	1	1	1	1	1	1	0
1	0 0 0 1	0	1	1	0	0	0	0
2	0 0 1 0	1	1	0	1	1	0	1
3	0 0 1 1	1	1	1	1	0	0	1
4	0 1 0 0	0	1	1	0	0	1	1
5	0 1 0 1	1	0	1	1	0	1	1
6	0 1 1 0	1	0	1	1	1	1	1
7	0 1 1 1	1	1	1	0	0	0	0
8	1 0 0 0	1	1	1	1	1	1	1
9	1 0 0 1	1	1	1	1	0	1	1

(b) Tabella di verità



(c) Schema logico

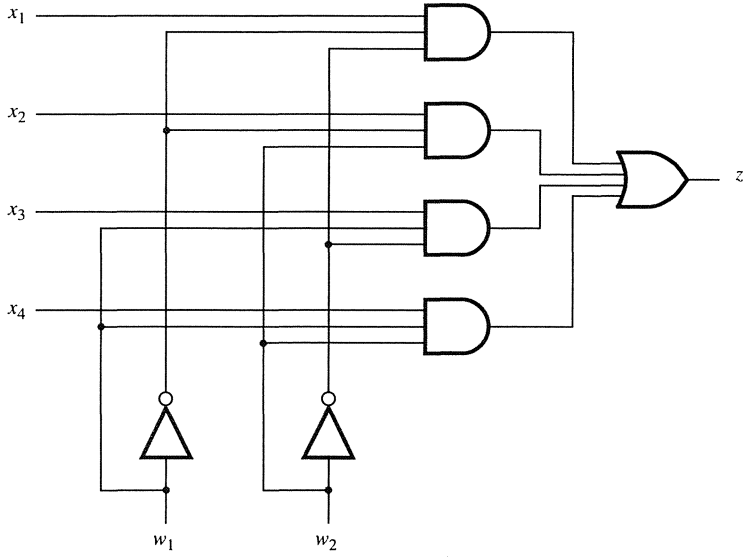
Figura A.37 Decodificatore da codice BCD a codice sette segmenti.



(a) Rappresentazione grafica

w_1	w_2	z
0	0	x_1
0	1	x_2
1	0	x_3
1	1	x_4

(b) Tabella di verità

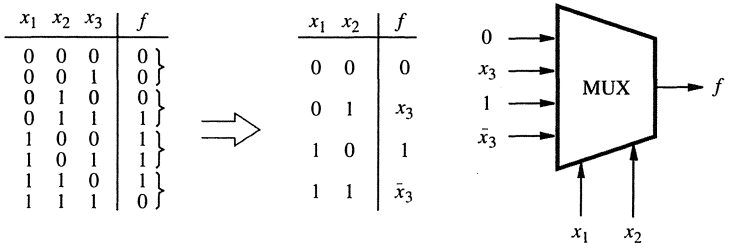


(c) Schema logico

Figura A.38 Multiplatore (multiplexer) a due ingressi di selezione.

I due ingressi di selezione sono chiamati w_1 e w_2; le quattro configurazioni di ingresso su w_1, w_2 definiscono quale dei quattro ingressi x_1, x_2, x_3 e x_4 debba essere convogliato sull'uscita z. La Figura A.38a mostra la rappresentazione grafica, la Figura A.38b la tabella di verità e la Figura A.38c lo schema logico del multiplatore. Non è difficile vedere come generalizzare lo schema logico al caso $n > 2$, usando più porte logiche o anche ricorrendo a una costruzione ad *albero*, dove per esempio il multiplatore con $n = 4$ si ottiene mediante un albero a due strati di multiplatori con $n = 2$ (quattro dei quali nel primo strato e uno nel secondo).

L'uso più ovvio del multiplatore consiste nel convogliare più linee di dato su una sola selezionando in ogni istante di tempo quale dato debba passare in uscita. Per esempio, se occorre collegare gli ingressi di un registro a 16 bit a caricamento parallelo a 4 possibili sorgenti di dato (ciascuna da 16 bit) basta prevedere una schiera di 16 multiplatori con $n = 2$ (cioè dotati di 2 ingressi di selezione e 4 ingressi di dato), dove ciascun multiplatore convoglia



(a) Tabelle di verità

(b) Schema logico

Figura A.39 Sintesi di funzione combinatoria mediante multiplatore.

verso il registro uno tra 4 bit provenienti da sorgenti differenti e tutti i multiplatori sono pilotati in parallelo dagli stessi segnali di selezione.

Il multiplatore è un blocco funzionale versatile e si può anche usare per la sintesi di funzioni combinatorie. Si consideri la funzione f la cui tabella di verità è data in Figura A.39. Si può compattare la tabella osservando che per certi gruppi di righe l'uscita è funzione solo della variabile di ingresso x_3; si possono unificare tali righe mettendo in evidenza x_3 nella colonna di uscita. Ciò fatto la funzione f figura avere solo due variabili in ingresso, x_1 e x_2, da cui dipende l'uscita. Si può osservare che ciascuna configurazione di x_1, x_2 definisce l'uscita: 0, 1, x_3 o $\bar{x}_3$. Ciò suggerisce di effettuare la sintesi della funzione f usando un blocco funzionale multiplatore con $n = 2$, dove i due ingressi di selezione sono pilotati da x_1 e x_2 e i quattro ingressi di dato da 0, 1, x_3 o $\bar{x}_3$. Chiaramente in tale modo il multiplatore riproduce la tabella di verità compattata e dunque calcola la funzione f. Il metodo qui è dato su un esempio ma è del tutto generale: si può fare la sintesi di qualsiasi funzione logica a tre variabili tramite il multiplatore a due ingressi di selezione (e quattro ingressi di dato). Similmente, si può fare la sintesi della funzione a 4 variabili tramite il multiplatore con $n = 3$, e così via.

A.11 Componenti programmabili (FPGA)

I blocchi funzionali complessi sono di tipo programmabile, vale a dire si possono configurare sul campo di lavoro definendone la funzione. La famiglia di componenti FPGA (*Field Programmable Gate Array*), ovvero *matrici programmabili di porte logiche*, è un esempio significativo e di grande importanza pratica. La Figura A.40 mostra lo schema logico di principio del componente FPGA tipico.

Il componente FPGA consiste in una matrice di elementi funzionali logici (in Figura A.40 sono indicati in chiaro) collegabili secondo uno schema flessibile controllato tramite elementi funzionali di interconnessione (in Figura A.40 sono ombreggiati). Questi ultimi contengono interruttori programmabili usabili per collegare o scollegare gli elementi funzionali logici. Vari bus interni di connessione permettono molta flessibilità nei collegamenti tra elementi funzionali. Il componente è anche dotato di porte o buffer di ingresso e uscita per pilotare i morsetti esterni (*pin*). Prima dell'uso il componente FPGA va programmato caricando la configurazione che gli si desidera dare; si veda la bibliografia [2, 3].

Gli elementi funzionali logici del componente FPGA possono essere di tipo combinatorio o sequenziale. Oltre a questi, numerosi modelli di FPGA contengono anche un numero più o meno

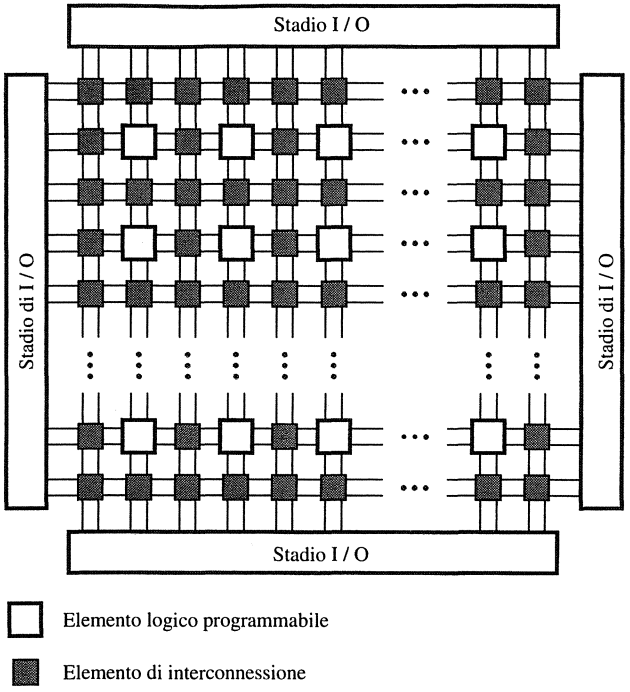


Figura A.40 Schema logico di componente FPGA.

grande di celle di memoria (la Figura A.40 non le mette in evidenza), che permettono di configurare blocchi funzionali come la coda FIFO, il banco di memoria ROM o RAM, e altro ancora.

La diffusione crescente di componenti di tipo FPGA è dovuta al fatto di permettere al progettista di realizzare reti logiche anche complesse su un unico componente integrato già disponibile commercialmente. Usando gli strumenti CAD di progetto per FPGA si riesce a definirne la funzionalità in pochi giorni di lavoro, di contro ai molti mesi di lavoro occorrenti per progettare da zero un circuito integrato VLSI. La famiglia FPGA è attraente anche dal punto di vista economico: il componente FPGA più complesso non costa più di qualche centinaio di euro e il costo di sviluppo tramite FPGA è molto modesto rispetto al costo di sviluppo di un componente VLSI specificamente dedicato a una funzione data. Per approfondimenti intorno alle varie famiglie di componenti programabili (la famiglia FPGA non è l'unica) si rimanda alla bibliografia e alla letteratura scientifica di settore.

Sommario

Lo scopo principale dell'appendice è rendere il lettore familiare con i concetti di base del progetto logico e dargli un'idea su struttura e funzionamento dei *circuiti digitali* (*reti logiche*), che costituiscono tutta o gran parte del calcolatore. Una certa familiarità con tali argomenti, sia pure ancora a livello iniziale, è indispensabile per affrontare e comprendere i capitoli principali del libro, quelli che contengono il nucleo essenziale dei concetti inerenti il calcolatore. Come si è accennato nell'appendice presente, oggigiorno il progetto delle reti logiche è una materia complessa ancorché affascinante e comprende nozioni di base consolidate e metodologie avanzate tuttora in fase di ricerca. Inoltre molto progetto logico sofisticato (per esempio il progetto di un processore) è svolto tramite *strumenti software* CAD (*Computer Aided Design*), per usare efficacemente i quali bisogna avere pratica ma anche conoscere parecchia teoria: *algebra di Boole*, *logica formale*, altre parti della *matematica* e dell'*informatica teorica*. Sono tutte competenze che un buon progettista impiega tempo ad acquisire e padroneggiare.

L'invenzione del *circuito integrato* (*Integrated Circuit* o IC), avvenuta nei primi anni '60, e in seguito la messa a punto di *strumenti software* CAD, hanno profondamente cambiato il modo di fare progetto logico. La varietà di componenti IC disponibili è grandissima e continua ad aumentare mentre il costo diminuisce, e anche i metodi di progetto logico stessi sono in trasformazione e progresso continui. Nell'appendice sono state spiegate in modo sintetico le nozioni essenziali intorno alle reti combinatorie e ai bistabili, e sono stati illustrati alcuni dei componenti IC (*blocchi funzionali*) di base per la costruzione del calcolatore. In particolare sono stati illustrati il transistore, la porta logica, il concetto di rete combinatoria a due livelli e come progettarla in forma canonica e minima, il concetto di rete sequenziale e alcune versioni di bistabile sincrono e asincrono, e infine i blocchi funzionali logici fondamentali, vale a dire porta tri-state, decodificatore, multiplatore, registro (seriale e parallelo), contatore e alcuni altri. Tutto questo è davvero solo l'inizio del progetto logico. Gli approfondimenti possibili sono numerosi e pertanto si rimanda alla bibliografia, per esempio [2, 3].

Dal punto di vista del progettista di calcolatori, oltre alla varietà e potenza di funzioni messe a disposizione dai blocchi funzionali logici di base, contano soprattutto il costo e la velocità dei circuiti digitali che egli progetta come parti del calcolatore. L'uno e l'altra sono migliorabili limitando il più possibile il numero di circuiti integrati costituenti il calcolatore: si vorrebbero avere pochi componenti, ciascuno capace di realizzare quante più funzioni possibile. Così si va nella direzione di progettare componenti integrati di taglia grande, contenenti numeri elevatissimi di transistori organizzati in reti logiche sofisticate e multifunzionali. La famiglia di componenti integrati programmabili FPGA (*Field Programmable Gate Array*) ha proprio tali caratteristiche e rappresenta una soluzione valida per molte applicazioni attuali, ma si trovano anche altre famiglie di componenti programmabili simili; si veda la bibliografia al riguardo, per esempio [1].

Tuttavia ci sono anche due altri aspetti che stanno assumendo importanza crescente nel progetto logico. Occorre potere *collaudare* (*testare*) il circuito integrato in modo ragionevolmente semplice ma efficace, per lo scopo duplice di verificarne e garantirne il funzionamento corretto in sede di fabbricazione, e di riuscire a rilevarne il malfunzionamento ed eventualmente anche a ripararlo dopo che ormai sia stato installato e reso operativo. Inoltre oggi spesso si chiede di avere sistemi di calcolo *affidabili* (*reliable*), cioè resistenti al guasto o almeno capaci di segnalarlo in tempo utile per evitare conseguenze indesiderabili. Si può ottenere tale caratteristica aggiungendo al circuito parti o funzioni *ridondanti*, se non addirittura *duplicandolo* completamente. Ma è fin troppo evidente che la ridondanza, e a maggior ragione le duplicazione completa, porta inevitabilmente all'aumento del costo del circuito. È compito di un buon progettista sapere valutare i rapporti tra queste diverse esigenze per potere scegliere la soluzione più soddisfacente. L'affidabilità dei circuiti integrati è stata ampiamente studiata insieme, più recentemente, alla *sicurezza* (*security*), e vi sono molti buoni testi al riguardo; per approfondimenti si rimanda pertanto alla letteratura scientifica.

Note bibliografiche

1. Brown S. e Vranesic Z., *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill, Burr Ridge, IL (USA) 2009.
2. Bolchini C., Brandolese C., Salice F., et al., *Reti logiche*, 2/e, Apogeo, Milano 2008.
3. Fummi F., Sami M., Silvano C., *Progettazione digitale*, 2/e, McGraw-Hill, Milano 2007.
4. Mano M.M. & Ciletti M.D., *Digital Design*, 4/e, Prentice Hall, Upper Saddle River, NJ (USA) 2007.

Indice analitico

A

addizionatore
– a propagazione di riporto, 322, 323
– algebrico, 324
– – in virgola mobile, 355
– completo, 322
– con anticipo del riporto, 326, 327, 328
– specifica logica, 322
addizione
– algebrica, 13, 14, 323
– con anticipo del riporto, 325
– – funzione di generazione, 325
– – funzione di propagazione, 326
– in complemento a due, 13, 323
– modulo N, 12
algebra di Boole, 420
ALU, 5, 164, 169
APIC, 137
applicazione embedded, 221, 359
– controllo
– – della temperatura, 380
– – della velocità, 380
architettura carica/immagazzina, 41
archiviatore, 146
area di attivazione, 75
– puntatore alla, 75
ARM, 33, 42
– registri, 43
assemblaggio, 144
– in due passate, 144
assemblatore, 58, 62, 143
– direttiva di, 59, 144
– – DATAWORD, 61
– – definizione di macro, 82
– – END, 61
– – EQU, 59, 144
– – invocazione di, 82
– – ORIGIN, 61, 66, 146
– – RESERVE, 61
– – START, 66
– macroassemblatori, 82
attuatore, 109, 379
– con solenoide, 379

B

banco di registri, 134, 139, 164, 169, 388
– a doppia porta, 170
– ausiliario, 140
– temporanei, 188
BCD, 67
bistabile, 443
– asincrono, 445
– clear, 453
– di tipo D, 448
– di tipo T, 451
– edge-triggered, 449
– JK, 451
– master-slave, 448
– preset, 453
– sincrono, 445
– SR, 445
– stato corrente, 446
– stato prossimo, 446
bit, 4, 10, 34, 112
– cella, 258
– – di memoria dinamica, 262
– – – lettura, 263
– – – rinfresco, 263
– – – scrittura, 263
– – di memoria EPROM, 273
– – di memoria PROM, 272
– – di memoria RAM statica, 260
– – – lettura, 261
– – – scrittura, 261
– – – tecnologia CMOS, 261
– – di memoria ROM, 272
– di controllo, 19, 20
– – di parità, 19
– di esito, 91, 99, 100
– – condizione logica, 99
– di lettura/scrittura, 276
– di modifica (dirty bit), 283
– di riporto, 84
– di segno, 10
– – estensione, 15, 333
– – riduzione, 15
– di stato, 112, 237, 238, 366, 368, 370
– di validità, 288
– ordine di enumerazione, 37
– Start, 241
– Stop, 241
bitmap, 23
bolla, 199, 203
buffer
– di ricezione, 369, 370
– di trasmissione, 369, 370
– interstadi, 196, 197
bus, 188, 221, 222
– arbitraggio, 232
– – priorità, 233
– – round-robin, 232
– asincrono, 228
– – handshake, 228
– – relazioni causa-effetto, 230

– banda, 268
– ciclo di, 224, 231
– di controllo, 223
– di dato, 223
– di indirizzo, 223
– pilota del, 188, 232
– protocollo di, 223
– – multiciclo, 227
– – PCI, 247
– – segnale di riscontro, 227, 228
– ritardo di propagazione, 225, 230
– sfasamento temporale (skew), 229
– sincrono, 223
– – diagramma temporale, 224, 226
– – frequenza di clock, 224
– – segnale di clock, 223
– standard di interconnessione, 242
– – FireWire, 246
– – ISA, 252
– – PATA, 252
– – PCI, 247
– – PCI Express, 253
– – SAS, 253
– – SATA, 252
– – SCSI, 251
– – USB, 242
– transazione, 248
– unità master, 223
– – iniziatore, 248
– unità slave, 223
– – obiettivo, 248
byte, 36
– indirizzamento, 36

C

cache, 181, 255, 278, 280, 396
– a *n* vie, 287
– coerenza, 289, 396, 397, 398
– dati scaduti, 289
– del processore, 297
– flush, 289
– hit, 282
– indirizzamento, 284
– – associativo, 285, 286
– – associativo a gruppi, 287, 288
– – campo
– – – blocco, 284
– – – etichetta (tag), 284, 286, 287
– – – gruppo, 287
– – – spiazzamento, 284, 286, 287
– – diretto, 284, 285
– istruzione di precarico, 298
– lettura
– – differita, 283
– – immediata, 283, 296
– linea di, 282
– livelli di, 297
– livello, 279
– miss, 202, 283
– non bloccante (lockup-free), 299
– posizione di, 282
– principio di funzionamento, 281
– scrittura, 282
– – differita, 283, 396, 397
– – immediata, 282, 396
– – – aggiornamento, 396
– – – invalidazione, 396
– separate per istruzioni e dati, 210, 284, 297
– snoopy, 397, 398
calcolatore, 1
– architettura, 1
– cloud, 2
– embedded, 2
– generazioni, 29
– Grid, 2
– mainframe, 30
– personali, 2
– prestazione del, 26
– server, 2
– supercalcolatori, 2
cambio di contesto, 160
campionamento, 22, 23, 243
– di indirizzo di colonna o CAS, 263
– di indirizzo di riga o RAS, 263
– frequenza di, 22, 243
canale di DMA, 277
caricatore, 66, 145
ciclo, 113
– di attesa, 113, 116, 401
– vettorizzabile, 389, 390
circuito, 175, 441
– antirimbalzo, 235
– arbitro, 232
– di controllo, 175
– di rinfresco, 265, 271
– di somma e riporto, 323
– integrato (IC), 30, 441
CISC, 41, 89, 93
CMOS, 434
codice, 18, 40, 144
– a ridondanza ciclica (CRC), 20
– BCD, 18, 85
– binario, 19
– correttore, 20
– di caratteri, 19
– – ASCII, 17
– – UCS, 19
– – Unicode, 19
– – UTF-8, 19
– di condizione, 91
– di controllo e correzione di errore, 19
– di correzione di errore (ECC), 310
– distanza di Hamming, 20
– mnemonico, 40, 58, 144
– – suffisso, 96, 99
– operativo, 58, 144
codifica autosincronizzante, 308
codificatore ASCII, 235
ColdFire, 33, 42
– registri, 43
collegamento, 244
– a festone, 246
– ad albero, 244, 253
– commutatore, 253, 395
– corsia, 254

– larghezza di banda, 393
– peer-to-peer, 246
– plug-and-play, 243, 244, 245, 247, 250
– ponte, 247
– punto-punto, 244, 246, 253
– throughput effettivo, 393
collegatore, 145
comparatore, 180, 184
compilatore, 146, 391
– direttiva, 153
– – asm, 153, 154
– – define, 150, 151
– ottimizzante, 147, 151, 195
– vettorizzante, 389
complemento logico, 412
compressione di dati, 24
– adattativa, 25
– audio, 25
– AV, 25
– basata su dizionario, 25
– di immagini, 25
– differenziale, 25
– relativa, 25
– run-length, 24
– variable-length, 25
condensatore, 262
condizione
– di indifferenza, 425
– di salto, 47
– di stato, 113
– suffissi di, 99
conflitto, 198
– su dati, 198
contatore, 369
– binario, 456
– delle iterazioni, 100
– di passi, 185, 187
– di programma (PC), 44, 163
– di rinfresco, 265, 271
contatore/temporizzatore, 370
– modalità, 370
– – contatore, 370
– – temporizzatore, 371
– registri, 371
conteggio dinamico, 204
controllo di errore, 19, 315
controllore
– del disco, 277, 309
– – buffer dei dati, 311
– – cache, 311
– di cache, 289
– di DMA, 275
– – registri, 276
– di memoria, 270
conversione A/D, 361, 364
convertitore
– A/D, 243
– D/A, 243
– parallelo-seriale, 234
– seriale-parallelo, 234, 241
core, 26
correzione di errore, 20, 315
costanti, 151
CPU, 163
cronometro di riflessi, 372
– circuito, 374
– programma C, 376
CUDA, 391

D

dato, 3
debugger, 66, 138, 148
– esecuzione passo-passo, 149
– punto di osservazione, 149
decodificatore, 457
– di indirizzo, 222, 234, 258
– di istruzioni, 185
densità di integrazione, 30
– legge di Moore, 30
destinazione, 44, 47
– di salto, 47
diagramma temporale, 157
disco, 309
– controllore del, 309, 312
– flessibile, 312
– magnetico, 300, 307
– – cilindro, 310
– – codifica di Manchester, 308
– – fattore di blocco, 310
– – formattazione, 310
– – latenza rotazionale, 311
– – partizione, 311
– – – di memoria virtuale, 311
– – – file system, 311
– – settore, 309, 310
– – tecnologia, 307
– – tempo di accesso, 300, 311
– – tempo di posizionamento, 311
– – traccia, 308, 309, 310
– – Winchester, 308
– ottico, 313
– – CD, 313
– – CD-R, 316
– – CD-ROM, 313, 315
– – CD-RW, 316
– – DVD, 317
– – settore, 315
– – traccia, 315
– pilota del, 308, 312
– unità, 312
distanza di Hamming, 20
divisione, 344
– con ripristino, 345
– schema di circuito, 345
– senza ripristino, 346
domotica, 363
driver del dispositivo, 161

E

EA (indirizzo effettivo), 50
eccezione, 132, 134, 138, 149, 300
– aritmetica, 351
– di errore di accesso a memoria, 140
– di istruzione non definita, 139, 140
– software, 139
– tipo di, 140

EIC (controllore esterno delle interruzioni), 139
elaborazione vettoriale (SIMD), 388
emissione multipla, 212, 220
esecuzione
– controllata, 66
– errori di, 66
– parallela, 26
– – a livello di istruzione, 26
– sequenziale, 44

F
fan-in, 326, 439
fan-out, 439
fase
– di esecuzione, 45, 164
– di prelievo, 45, 164
file, 318
flip-flop, 189, 448
forma canonica
– prima, 415
– seconda, 416
forno a microonde, 360
– diagramma a blocchi, 361
– magnetron, 360
– microcontrollore, 360
FP (puntatore all'area di attivazione), 75
FPGA (*field programmable gate array*), 31, 359, 461, 463

G
generatore di idirizzi delle microistruzioni, 193
generatore di indirizzi delle istruzioni, 164, 175, 184, 185
GUI (interfaccia utente grafica), 145

H
handshake, 230
– procedura, 230
– segnali, 238
hardware, 1
HDTV, 244
hex, 67
host, 245, 391
hub, 244

I
IA-32, 33, 42
IC. *Vedi* circuito integrato
IE. *Vedi* abilitazione di interruzione
implicante, 424
indirizzamento
– spazio di, 34
– – unificazione, 110, 113
indirizzo, 34
– assoluto, 145
– di rientro, 70, 71, 120, 372
– effettivo (EA), 50
– etichetta, 61, 62, 144
– fisico, 300, 303
– logico, 300
– – campo, 302
– – – numero di pagina, 301
– – – spiazzamento, 301
– multiplazione, 270
– traduzione, 301, 305
– – tempo medio, 305
– – tramite tabella TLB, 304
– USB, 244, 245
– vettore delle interruzioni, 152
– vettore di interruzione, 124
– – indice, 137
– vettore di interruzioni, 158
– vettori per le interruzioni, 372
– virtuale, 300
Interconnessione, 188
– bus, 189
interfaccia, 110, 234
– circuito di, 113, 223
– – d'ingresso, 236
– – di uscita, 239
– connettore, 250, 252
– di ingresso, 235, 365
– di uscita, 239, 365
– dispositivo di ingresso, 222
– parallela, 234, 365
– – registro, 367
– ROM di configurazione, 250, 251
– seriale, 238, 368
– – registro, 370
– – UART, 368
– USB, 245
interfaccia processore-memoria, 8, 164, 183, 256
interruzione, 3, 9, 116, 123, 149, 277
– abilitazione di (IE), 123, 125, 136, 367, 369, 370, 371
– annidamento, 124
– annidate, 372
– bit di controllo, 122
– conferma di, 120
– controllo di, 122, 125, 371
– controllore esterno, 137, 139
– da errore, 370
– – di ricezione, 368
– disabilitazione, 134
– fault di pagina, 305
– latenza di, 121, 140, 141
– livello di priorità, 125, 372
– – maschera, 136
– non mascherabile, 136
– richiesta di, 9, 116, 367, 371
– – multiple, 124
– routine di servizio, 9
– servizio di, 120
– – scansione, 124, 134, 371
– – vettorizzata, 124, 372
– software, 140, 149, 158
inversione, 412
IOPL (livello di privilegio di I/O), 141
IR (registro di istruzione), 45, 163
IRQ, bit di, 124, 371
ISA (insieme di istruzioni macchina), 33, 38
istruzione, 38, 40, 115, 149, 176
– a due indirizzi, 89
– a tre indirizzi, 44
– addizione, 44
– binaria, 62, 176
– – codifica di, 176

– commento, 62
– condizionale, 48
– deallocazione della memoria in pila, 82
– di caricamento multiplo di registri, 81
– di chiamata, 70
– – a sottoprogramma, 70
– di divisione, 86
– di I/O, 115, 117, 222
– di moltiplicazione, 86
– – a quattro operandi, 100
– di rientro, 70
– – da interruzione, 121, 154
– di rotazione, 86
– di salto, 47, 70
– – condizionato, 92, 99
– – incondizionato, 98
– di salvataggio multiplo, 82
– di scorrimento logico, 83
– di servizio, 152
– – vettorizzata, 152
– di sottrazione, 100
– di trappola, 149
– di trasferimento, 97
– – multiplo, 97
– effetto collaterale, 218
– ingombri, 62
– logica, 80
– – bit a bit, 80
– macchina, 38, 40
– – codifica numerica, 103
– – formati binari, 106
– nulla (NOP), 199
– privilegiata, 139, 141, 306
– rientro
– – da eccezione, 134
– – da interruzione, 137, 149, 372
– scorrimento aritmetico, 86
– simbolica, 62
– sintassi, 96
– tipo di, 96, 127
– – aritmetiche, 97
– – confronto, 97
– – controllo, 96
– – logiche, 97
– – scorrimento, 97
– – speciali, 127
– – trasferimento, 96
– trap, 139
– vettoriale, 388
istruzione, 3
– esecuzione, 7
– macchina, 3, 7

J
jitter, 243

K
Karnaugh, 421

L
LCD, 360, 362
LED, 372
libreria, 146, 158
LIFO, 67, 71
linguaggio
– assemblativo, 40, 58, 143, 147
– del processore, 58
– di alto livello, 146, 147
– – C, 150, 372
– – – CUDA C, 391
– – – OpenCL, 391
– – supporto alle interruzioni, 154
– macchina, 29, 58
– – binario, 29
– – codice, 143
– – simbolico, 29, 58
– – sintassi, 58, 62
località, 280, 305
– spaziale, 281
– spaziale-temporale, 281
– temporale, 281
LSB, 325

M
macchina di Turing, 28
macchina di von Neumann, 28
macchina fotografica digitale, 361
– diagramma a blocchi, 362
macroistruzioni, o macro, 82
– espansione, 82
maxtermine, 416
meccanismo di collegamento, 70
– basato su registro, 71
– in pila implicito, 71
memoria, 4, 255
– a doppia porta, 169
– a semiconduttori, 257
– a sola lettura (ROM), 271
– a sola lettura programmabile e cancellabile elettricamente (EEPROM), 274
– a sola lettura programmabile e cancellabile (EPROM), 273
– a sola lettura programmabile (PROM), 272
– accesso diretto alla (DMA), 275, 288
– ad accesso diretto, 255, 310
– algoritmo di sostituzione, 282, 289, 305
– – a caso, 290
– – FIFO, 290
– – LRU, 289, 290, 305
– banco, 258, 268
– cache, 5, 30, 117, 151, 257
– capacità, 255, 259, 280
– cella, 258
– centrale, 4, 257
– – con cache, 295
– – – tempo medio di accesso, 295, 297
– chip, 258
– – circuito di lettura e scrittura, 258, 272
– – linea di parola, 258
– – linee di bit, 258
– – matrice di celle, 258, 259
– condivisa, 306, 391
– – blocco, 397, 398
– – directory, 398
– – modulo, 391, 392
– costo, 280

– di controllo, 192
– di massa, 5, 306
– dinamica, 262
– – asincrona, 262
– – banco, 270
– – rinfresco, 271
– – modulo, 270
– – – DIMM, 270
– – – SIMM, 270
– – SDRAM a frequenza doppia (DDR SDRAM), 268
– – sincrona (SDRAM), 265
– flash, 274
– – chiave, 275
– – scheda, 274, 362
– gerarchia, 255, 278, 295
– lettura, 38
– modo, 265
– – registro di, 266
– – veloce di pagina, 265
– pagina
– – area di, 302
– – fault di, 300, 304
– – hit di, 304
– permanente, 156
– persistente, 5, 271
– prestazione, 295
– – buffer di scrittura, 297
– – – scrittura differita, 298
– – – scrittura immediata, 297
– – cache non bloccante, 299
– – caricamento anticipato, 298
– – penalità di miss, 295, 296
– – tasso di hit, 295, 296
– – tasso di miss, 295
– primaria, 4, 300
– principale, 257
– protezione, 306
– RAM, 5, 257
– – dinamica (DRAM), 262
– – statica (SRAM), 259
– Rambus, 268
– rapporto costo/prestazione, 294
– scrittura, 38
– secondaria, 5, 257, 300
– statica, 259
– – banco, 269
– tempo di accesso, 5
– trasferimento di blocco, 266
– – banda, 267
– – diagramma temporale, 266
– – latenza, 266
– unità di gestione, 139, 300
– velocità, 256, 280
– – tempo di accesso, 256
– – tempo di ciclo, 257
– virtuale, 30, 139, 255, 257, 299
– – numero di pagina fisico, 303
– – tabella delle pagine, 302
– – – bit di controllo, 303, 305
– – – registro di base, 302, 306
– – tabella TLB, 303
– – – bit di controllo, 303
– volatile, 5
– volatilità, 259
microcontrollore, 359
– ARM, 364, 382
– – Thumb, 383
– diagramma a blocchi, 364, 365
– Freescale, 364, 382
– Intel, 364, 381
– nucleo processore, 363
microelettroniche, 257
microistruzione, 192
microprocessore, 31
microprogrammazione, 30
microroutine, 192
MIDI, 23
mintermine, 416
MMU, 139, 300, 303, 304
modi di indirizzamento, 49, 56
– assoluto, 49
– CISC, 90
– con autodecremento, 90
– con autoincremento, 90
– con base e indice, 55
– con base e spiazzamento, 57
– con base, indice e spiazzamento, 55
– con indice e spiazzamento, 52
– con post-base, 57
– con pre-base, 57
– con writeback, 57
– di registro, 49
– diretto, 49, 57
– immediato, 50
– indiretto, 50
– – da memoria, 57
– puntatore, 50
– relativo a contatore di programma, 91
– RISC, 49
– – valore a 32 bit, 88
moltiplicatore,
– a matrice, 330
– – con propagazione del riporto, 338, 339
– – con salvataggio del riporto, 338, 339
– – – albero CSA, 340
– – – riduttore 3-2, 340
– – – riduttore 4-2, 343
– completo, 330
– sequenziale, 331, 332
moltiplicazione, 329
– addizione con salvataggio del riporto (CSA), 337
– – albero CSA, 341
– – riduttore 4-2, 342
– in complemento a due, 333
– – algoritmo di Booth, 333
– ricodifica, 334
– – bit-pair, 336, 337
– – di Booth, 334, 335
– veloce, 344
MOS, 433
MSB, 325
MSI (scala di integrazione media), 442
multicalcolatore, 27, 399
– nodo, 399

– scambio di messaggi, 399
multiplatore, 458
multiprocessore, 27, 391
– nodi, 392
– NUMA, 392, 393
– programmazione parallela, 400
– – libreria POSIX threads (Pthreads), 401
– – routine di libreria, 400, 401
– rete di interconnessione, 393
– – anello, 394
– – – bidirezionale, 394
– – – gerarchia, 394
– – – singolo, 394
– – bus, 393
– – – a transazioni spezzate, 393
– – – arbitraggio, 393
– – crossbar, 395
– – maglia (mesh), 395
– – – toroidale, 396
– UMA, 392
multiprogrammazione, 159
multithreading hardware, 388
– coarse-grained, 388
– fine-grained, 388
– interleaved, 388

N

nastro magnetico, 317
– cartucce, 318
negazione, 412
NIOS II, 33, 42
– registri, 43
NMOS, 433
nome, 144
– esportato, 145
– esterno, 145
numeri frazionari, 16
– cifre significative, 16
– fattore di scala, 17
– in virgola fissa, 16
– in virgola mobile, 16
– precisione, 16
numeri interi, 10
– complemento a due, 10, 11, 12
– complemento a uno, 10
– naturali, 11
– segno e valore assoluto, 10
numero
– binario, 67
– esadecimale, 67
numero frazionario
– in virgola fissa
– – conversione da decimale a binario, 356
– in virgola mobile, 347
– – algoritmi, 351
– – – addizione algebrica, 352
– – – divisione, 352
– – – moltiplicazione, 352
– – – standard, 351
– – bit di guardia, 352
– – – sticky bit, 354
– – bit nascosto, 349
– – bit significativo, 348
– – fattore di scala, 348
– – – base, 348
– – – esponente, 348
– – fattore intero, 349
– – forma normale, 350
– – mantissa, 349
– – operazioni, 350
– – – standard, 350
– – precisione, 348
– – – doppia, 349
– – – singola, 348
– – rappresentazione, 348
– – segno, 348
– – troncamento, 353
– – – approssimazione distorta, 353
– – – approssimazioni senza distorsione, 353
– – – arrotondamento, 353
– – – di von Neumann, 353
– – – spezzamento, 353
– – – standard, 353
– – valori speciali, 350
– – – azzeramento graduale, 351
– – – forma non normale, 351
– – – infinito, 351
– – – NaN, 351
numero intero
– complemento a due, 321
– eccesso 1.023, 349
– eccesso 127, 349

O

operando
– a 64 bit, 100
– di destinazione, 59
– immediato, 59, 175, 178
– pseudoimmediato, 98
– sorgente, 59
ottimizzazioni, 147

P

pacchetto, 393
parallelismo
– di dati, 388
– di programmi, 399
parola
– allineamento, 37
– di controllo, 192
– indirizzamento, 37
– lunghezza, 34
parola di memoria, 4
– indirizzo, 4
– lettura, 4
– lunghezza, 4
– scrittura, 5
passaggio di parametri, 71
– per riferimento, 75
– per valore, 75
– tramite i registri, 71
– tramite la pila, 71
PC (contatore di programma), 44, 163
percorso dati, 182

periferica, 109, 234, 276
pila, 67
– base, 67
– cima, 67
– fondo, 67
– operazioni, 67
– – impila, 67
– – pop, 67
– – push, 67
– – spila, 67
– puntatore alla, 68
– verso di crescita, 67
– vuota, 67
pipeline, 166, 196
– nei processori CISC, 218
– – ColdFire, 219
– – Intel, 219
– numero di stadi, 210, 212
– prestazione, 210
– profondità, 212
pipelining, 26, 30, 195, 220
pixel, 23, 361
PMOS, 433
porta, 234
– AND, 411, 413
– di accesso per il collaudo, 385
– di I/O, 234
– – parallela, 234, 366
– – seriale, 234, 239, 240
– di ingresso, 235
– di uscita, 238
– NAND, 426
– NOR, 426
– NOT, 413
– OR, 413
– passante, 441
– seriale, 238
– tri-state, 188, 232, 440
– XOR, 412, 413
POS, 416
prestazione (prodotto di somme), 294, 402
– elaborazione parallela, 403
– – Legge di Amdahl, 403
– equazione di base, 210
– guadagno, 210, 295, 403
– penalità, 211
– – di cache miss, 211
– – di salto, 203, 204, 211
– – dipendenze di dato, 211
– throughput, 210
processo, 159, 387, 388
– assemblativo, 143
– di avvio, 156
– thread, 387, 388
processore, 3, 7, 163
– a cinque stadi, 197
– CISC, 94, 188, 190
– componenti hardware, 164, 169
– core, 391
– modo di eccezione, 140
– modo operativo, 139, 141
– – di sistema, 306
– – di utente, 306
– multicore, 26, 399
– organizzazione a cinque stadi, 172, 195
– percorso dati, 172, 195, 200
– registro di controllo, 152, 126, 134
– RISC, 94
– sezione
– – di esecuzione, 171
– – di prelievo, 174
– struttura a più stadi, 166
– su un singolo chip, 31
– superscalare, 195, 213
– – buffer di riordino, 217
– – eccezioni precise, 216
– – esecuzione, 214
– – – completamento, 216
– – – fuori ordine, 215
– – – speculativa, 215
– – organizzazione, 213
– – stazione di prenotazione, 215
– – throughput, 214
– – unità
– – – di commitment, 216
– – – di prelievo, 213
– – – di smistamento, 213, 217
– superscalari, 212
prodotto logico, 411
profondità di colore, 23
programma, 3, 29
– chiamante, 70
– chiamato, 70
– concorrenti, 158
– di I/O, 111, 113, 154
– – ARM, 135
– – CISC, 116, 117, 132
– – ColdFire, 137
– – controllo, 111
– – gestore di interruzioni, 130, 155
– – IA-32, 137
– – ISR, 120
– – NIOS, 135
– – RISC, 115, 117, 129
– di simulazione controllata, 138, 148
– esecuzione, 9
– oggetto, 58, 143
– – generazione, 148
– registro di stato, 122, 125, 152
– sorgente, 58, 143
PS, 121
pseudoistruzione, 52, 98
– Move, 52
– MoveImmediateAddress, 88
puntatore, 150, 155
– alla pila (SP), 68
– volatile, 151

Q

quantizzazione, 22, 23
– unità di, 23
quanto di tempo, 159, 388

R

RAID, 312
– data striping, 312

rapporto di compressione, 24
registro, 5, 453
– a scorrimento, 238, 454
– – di ricezione, 369
– – di trasmissione, 369
– asincrono, 266
– base, 55
– contatore di m-programma, 192
– contatore di programma (PC), 44, 163
– di collegamento, 70, 155, 372
– di controllo, 367, 369, 370
– di direzione dei dati, 366
– di I/O, 111, 152
– – controllo, 111, 122, 126
– – dati, 111, 126
– – stato, 111, 126
– di istruzione (IR), 7, 45, 163, 183
– – campi, 176, 183
– di programma, 8
– di stato, 91, 367, 368, 370
– di uso generale, 8
– di uso speciale, 8
– divisore del clock, 369
– indice, 52, 55
– interstadio, 172, 181
– IR, 7
– PC, 8
– tempo di hold o di mantenimento, 226
– tempo di setup o di impostazione, 225, 230
– universale, 456
– vettoriale, 389
rete
– combinatoria, 413
– di interconnessione, 3, 109, 221
– – a bus singolo, 222
– sequenziale, 443
RGB, 23
ricerca associativa, 287, 305
riporto
– circolare, 15
– in entrata, 12
– in ingresso, 322
– in uscita, 11, 322
– programmazione del, 322
– propagazione del, 13
RISC, 41, 93, 100
risoluzione, 23
ritardo, 439
– di porta, 325
– di propagazione, 439
– di salto, 203
– – posto del, 205
rivelazione di errore, 20, 315
routine, 69, 120, 147
– di gestione di memoria, 306
– di libreria, 147, 391
– di servizio delle interruzioni, 120, 158, 371
– di SO, 158, 160
– servizio d'interruzioni, 154
– – funzione intserv, 154
– – parola chiave interrupt, 154
RTN (notazione a trasferimento di registro), 39

S

salto
– buffer di destinazione, 209
– differito, 206
– predizione
– – accuratezza, 211
– – algoritmo, 207, 208
– – dinamica, 207
– – errata, 211, 215
– – statica, 206
schema di indirizzamento, 37, 255
scorrimento, 83
– aritmetico, 85
– logico, 83
segnale di condizione, 184, 186
segnale di controllo, 181
– ALU_op, 184
– B_selettore, 184, 187
– C_selettore, 182
– Contatore_abilita, 187
– extend, 183
– generatore di, 185
– INC_selettore, 185
– IR_abilita, 183, 184
– MA_selettore, 183
– MEM_lettura, 183, 184
– MEM_scrittura, 183, 184
– operazione in memoria completata MFC, 181, 183, 184, 187
– PC_abilita, 185, 187
– PC_selettore, 185
– RF_scrittura, 184, 186
– WMFC, 187
– Y_selettore, 182
sensore, 109, 360, 375
– con partitore di tensione, 377
– di posizione, 377
– di pressione, 378
– di temperatura, 377
– di velocità, 378
– ottico, 361, 378
– – CMOS, 361
– – DAC, 361osistema di I/O, 3
sistema embedded, 359
– progettazione, 383
sistema operativo (SO), 141, 143, 145, 155, 300, 304, 306, 387
– commutazione di contesto, 306, 388
– esecuzione parallela, 401
– multiprogrammato, 158
– uso delle interruzioni, 158
sistema reattivo, 359, 372
sistemi embedded, 274
sistemi su un chip, 382
SO. *Vedi* sistema operativo
software, 1
soglia, 263, 430, 438
somma, 411
– logica, 411
SOP (somma di prodotti), 414
sorgente, 44
sottoprogramma, 69
– annidamento, 70, 77

– collegamento, 81
sottrattore, 324
sottrazione, 12
– algebrica, 14
– in complemento a due, 324
SP (puntatore alla pila), 68
spazio, 113
– di indirizzamento, 113
– – separato, 115, 247
– – di sistema, 306
– – di utente, 306
– – PCI, 247
– – unificato, 247
– – unificazione, 113, 222
spiazzamento, 52
– di salto, 178
SSI (scala di integrazione piccola), 442
stallo, 198
– dipendenza di dato, 198, 200, 201
– inoltro di operandi, 199, 202
– limiti di risorse, 209
– ritardo di accesso alla memoria, 202
stringa, 4, 10
SVG, 23

T
tabella
– dei simboli, 144
– dei vettori d'interruzione, 124
– di transizione, 445
– di verità, 411
telemetria, 363
tempo di transizione, 439
thread, 400
– creazione, 400
– di controllo dell'esecuzione, 400
– sincronizzazione, 401
– – barriera, 401
trabocco, 14, 15, 92, 323, 350
– circuito di rilevamento, 324
traffico
– a raffica, 248
– in streaming, 253
– isocrono, 243, 245, 246, 252
– raffiche, 247
transistore
– a giunzione bipolare, 30
– a semiconduttori, 4
– laterale, 30
trasduttore, 375
trasmissione, 241
– a terminazione singola, 246, 251
– broadcast, 393
– isocrona, 252
– rumore, 246
– segnalazione differenziale, 246, 251, 252, 254, 268
– seriale, 241
– – asincrona, 241
– – sincrona, 241, 242
– Start-Stop, 241
trasparenza, 448

U
unità funzionale, 2, 164
– aritmetica, 164
– aritmetica-logica, 3, 5
– di calcolo, 190
– di comunicazione, 399
– di controllo, 3, 6, 185
– – cablato, 185
– – microprogrammato, 191, 192
– di elaborazione grafica (GPU), 391
– di I/O, 6, 112
– di ingresso, 2, 4
– di memoria, 4
– di uscita, 3, 6
– logica, 164
– memoria, 3

V
variabili volatili, 151
vettore, 388
– lunghezza del, 389
vettorizzazione, 389, 390
VLSI, 26, 30, 257, 442

gli esercizi di McGraw-Hill

ARCHITETTURA DEI CALCOLATORI

PROBLEMI RISOLTI E PROPOSTI

250 problemi con soluzioni sul sito internet

Istruzioni macchina • Operazioni di Input/Output
Software di supporto • Struttura del processore • Pipelining
Sistema di Input/Output • Sistema di memoria • Aritmetica binaria
Sistemi embedded • Elaborazione parallela e prestazioni
Elementi di logica della commutazione

McGraw Hill

1 Introduzione al calcolatore

Problemi risolti

Problema 1.1

Elencare i passi necessari a eseguire l'istruzione macchina:

Load R2, LOC

in termini di trasferimenti tra i componenti mostrati in Figura 1.2 e di alcuni semplici comandi di controllo. Il Paragrafo 1.3 mostra una panoramica dei passi necessari. Assumere che l'indirizzo della locazione di memoria contenente questa istruzione sia inizialmente nel registro PC.

Soluzione

I passi richiesti sono:

- mandare l'indirizzo della parola di istruzione dal registro PC alla memoria ed emettere il comando di controllo Read;
- aspettare fino a quando la parola richiesta sia stata prelevata dalla memoria, quindi caricarla nel registro IR, dove questa è interpretata (decodificata) dal circuito di controllo per determinare l'operazione da eseguire;
- incrementare i contenuti del registro PC per puntare alla prossima istruzione in memoria;
- mandare il valore dell'indirizzo LOC dall'istruzione nel registro IR alla memoria e fornire il comando di controllo Read;
- aspettare finché la parola richiesta sia stata prelevata dalla memoria, quindi caricarla nel registro R2.

Problema 1.2

Quantificare gli effetti sulle prestazioni che risultano dall'uso di una cache nel caso in cui un programma abbia un totale di 500 istruzioni, tra cui un ciclo di 100 istruzioni che sia eseguito 25 volte. Determinare il rapporto tra il tempo di esecuzione senza la cache e il tempo di esecuzione con la cache. Questo rapporto è chiamato *accelerazione* (*speedup*).

Assumere che gli accessi alla memoria principale richiedano 10 unità di tempo e gli accessi alla cache richiedano 1 unità di tempo. Inoltre facciamo le seguenti assunzioni ai fini di semplificare i calcoli e illustrare facilmente i vantaggi dell'uso della cache.

- Il tempo di esecuzione del programma è proporzionale al tempo totale necessario a prelevare istruzioni dalla memoria principale o dalla cache, trascurando gli accessi dovuti al prelievo degli operandi.
- Inizialmente tutte le istruzioni sono memorizzate nella memoria principale e la cache è vuota.
- La cache è sufficientemente grande da contenere tutte le istruzioni del ciclo.

Soluzione

Il tempo di esecuzione senza la cache è:

$$T = 400 \times 10 + 100 \times 10 \times 25 = 29.000$$

Il tempo di esecuzione con la cache è:

$$T_{cache} = 500 \times 10 + 100 \times 1 \times 24 = 7.400$$

Quindi l'accelerazione è:

$$T/T_{cache} = 3{,}92$$

Problema 1.3

Convertire le seguenti coppie di numeri decimali in numeri di 5 bit in complemento a due, quindi sommare e sottrarre ciascuna coppia. Indicare se c'è o meno trabocco in ciascun caso.

a. 7 e 13
b. −12 e 9

Soluzione

La conversione e le operazioni sono:

a. $7_{10} = 00111_2$ e $13_{10} = 01101_2$.
Sommando questi due numeri positivi otteniamo 10100, che è un numero negativo. Quindi c'è stato trabocco.
Per sottrarre i numeri, prima calcoliamo il complemento a due di 01101, che è 10011. Quindi sommiamo questo con 00111 ottenendo 11010, che è -6_{10}, ovvero il corretto risultato.

b. $-12_{10} = 10100_2$ e $9_{10} = 01001_2$.
Sommando questi due numeri otteniamo $11101 = -3_{10}$, il corretto risultato.
Per sottrarre i numeri, prima calcoliamo il complemento a due di 01001, che è 10111, quindi sommiamo i due numeri negativi 10100 e 10111, ottenendo 01011, che è un numero positivo. Quindi c'è stato trabocco.

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

1.1 F Ripetere il Problema risolto 1.1 per l'istruzione macchina:

Add R4, R2, R3

che è descritta nel Paragrafo 1.3.

1.2 F Ripetere il Problema risolto 1.1 per l'istruzione macchina:

Store R4, LOC

che è descritta nel Paragrafo 1.3.

1.3 M

a. Dare un breve sequenza di istruzioni macchina per il compito "sommare il contenuto della locazione di memoria A a quello della locazione B e porre il risultato nella locazione C". Le istruzioni:

Load R*i*, LOC

e:

Store R*i*, LOC

sono le sole istruzioni disponibili per trasferire dati tra la memoria e i registri generali. Le istruzioni di somma sono descritte nel Paragrafo 1.3. Non cambiare il contenuto di nessuna delle locazioni A e B.

b. Supporre che le istruzioni Move e Add siano disponibili con i formati:

Move Location1, Location2

e:

Add Location1, Location2

Queste istruzioni spostano o sommano alla prima locazione una copia dell'operando alla seconda locazione, sovrascrivendo l'operando originale alla prima locazione. Uno o entrambi gli operandi possono essere in memoria o in registri generali. È possibile usare meno istruzioni di questi tipi per ottenere l'obiettivo della parte **a**? Se sì, dare la sequenza.

1.4 M

a. Un programma consiste di un totale di 300 istruzioni e contiene un ciclo di 50 istruzioni che è eseguito 15 volte. Il processore contiene una cache, come descritto nel Paragrafo 1.2.2. Prelevare ed eseguire un'istruzione che si trova in memoria principale richiede 20 unità di tempo. Se l'istruzione è trovata nella cache, prelevare ed eseguire l'istruzione richiede solo 2 unità di tempo. Ignorando gli accessi dovuti al prelievo degli operandi, calcolare il rapporto tra il tempo di esecuzione del programma senza la cache e il tempo di esecuzione con la cache. Questo rapporto è chiamato *accelerazione* (*speedup*) dovuta all'uso della cache. Assumere che la cache sia inizialmente vuota, che sia sufficientemente grande da contenere il ciclo e che il programma inizi con tutte le istruzioni presenti in memoria principale.

b. Generalizzare la parte **a** sostituendo le costanti 300, 50, 15, 20 e 2 con le variabili w, x, y, m e c. Formulare un'espressione per l'accelerazione.

c. Per i valori $w = 300$, $x = 50$, $m = 20$ e $c = 2$ quale valore di y dà accelerazione 5?

d. Considerare la forma dell'espressione per l'accelerazione sviluppata nella parte **b**. Qual è il limite superiore sull'accelerazione al crescere del numero di iterazioni y del ciclo?

1.5 M

a. Una cache per processore è descritta nel Paragrafo 1.2.2. Supporre che il tempo di esecuzione per un programma sia proporzionale al tempo di prelievo delle istruzioni. Assumere che prelevare un'istruzione dalla cache richieda 1 unità di tempo, ma che prelevarla dalla memoria principale richieda 10 unità di tempo. Inoltre assumere che un'istruzione richiesta sia trovata nella cache con probabilità 0.96. Infine assumere che, se un'istruzione non è trovata nella cache, essa debba essere prima prelevata dalla memoria principale e inserita nella cache, poi prelevata dalla cache per essere eseguita. Calcolare il rapporto tra il tempo di esecuzione di un programma senza cache e il tempo di esecuzione con cache. Questo rapporto è chiamato *accelerazione* (*speedup*) dovuta alla presenza della cache.

b. Se la dimensione della cache è raddoppiata, assumere che la probabilità di non trovare una istruzione richiesta sia dimezzata. Ripetere la parte **a** per una cache con dimensione doppia.

1.6 F Estendere la Figura 1.4 per incorporare entrambe le possibilità di riporto in entrata (0 oppure 1) per ognuno dei quattro casi mostrati in figura. Specificare sia i bit somma sia il riporto in uscita per ognuno degli otto nuovi casi.

1.7 M Convertire le seguenti coppie di numeri decimali in numeri di 5 bit in complemento a due, successivamente sommarli. Valutare se c'è o meno trabocco in ciascun caso.

a. 4 e 11
b. 6 e 14
c. -13 e 12
d. -4 e 8
e. -2 e -9
f. -9 e -14

1.8 M Ripetere il Problema proposto 1.7 per l'operazione di sottrazione, dove il secondo numero di ciascuna coppia va sottratto dal primo numero. Valutare se c'è o meno trabocco in ciascun caso.

1.9 F Una locazione di memoria di un byte contiene la sequenza 01010011. Quale valore decimale rappresenta questa sequenza quando è interpretata come numero binario? Cosa rappresenta come codice ASCII?

1.10 F Un modo per riconoscere il trabocco quando si sommano due numeri in complemento a due è descritto alla fine del Paragrafo 1.4.1. Determinare come rilevare il trabocco quando si sottraggono due di tali numeri.

2 Istruzioni macchina

Problemi risolti

Problema 2.1

Si assuma che una stringa di caratteri ASCII sia immagazzinata in memoria a partire dall'indirizzo STRINGA. La stringa termina con il carattere di Ritorno Carrello (CR). Scrivere un programma in stile RISC per determinare la lunghezza della stringa e immagazzinarla nella locazione LUNGHEZZA.

Soluzione

La Figura P2.1 presenta un possibile programma. I caratteri nella stringa sono confrontati con CR (codice ASCII 0x0D), e un contatore viene incrementato fino a quando si raggiunge la fine della stringa (si veda la Figura 92.2).

Problema 2.2

Si voglia trovare il più piccolo numero in una lista di interi positivi a 32 bit. La parola all'indirizzo 1000 dovrà essere usata per contenere il valore del numero più piccolo dopo che questo sia stato trovato. La prossima parola contiene il numero di elementi, *n*, della lista. Le successive *n* parole contengono i numeri della lista. Il programma deve partire all'indirizzo 400. Scrivere un programma in stile RISC per trovare il numero più piccolo e includere le direttive di assemblatore necessarie per organizzare il programma e i dati specificati. Sebbene il programma dovrà essere in grado di gestire liste di diverse lunghezze, includere nel codice una piccola lista di dati d'esempio che consista di sette interi.

Soluzione

Il programma in Figura P2.2 svolge il compito richiesto. I commenti nel programma spiegano come è effettuato il compito.

```
        Move            R2, #STRINGA      R2 punta all'inizio della stringa
        Clear           R3                R3 è un contatore che viene azzerato
        Move            R4, #0x0D         Codice ASCII per Ritorno Carrello (CR)
CICLO:  LoadByte        R5, (R2)          Preleva il prossimo carattere
        Branch_if_[R5]=[R4] FATTO         Finito se il carattere è CR
        Add             R2, R2, #1        Incrementa il puntatore alla stringa
        Add             R3, R3, #1        Incrementa il contatore
        Branch          CICLO             Non ancora finito, reitera
FATTO:  Store           R3, LUNGHEZZA     Immagazzina il conto nella locazione
                                            LUNGHEZZA
```

Figura P2.1 Programma per il Problema risolto 2.1.

```
LISTA      EQU                 1000          Indirizzo iniziale della lista
           ORIGIN              400
           Move                R2, #LISTA    R2 punta all'inizio della lista
           Load                R3, 4(R2)     R3 è un contatore, inizializzato a n
           Add                 R4, R2, #8    R4 punta al primo numero
           Load                R5, (R4)      R5 contiene il numero più piccolo trovato finora
CICLO:     Subtract            R3, R3, #1    Decrementa il contatore
           Branch_if_[R3]=0    FATTO         Finito se R3 è uguale a 0
           Add                 R4, R4, #4    Incrementa il puntatore alla lista
           Load                R6, (R4)      Preleva il prossimo numero
           Branch_if_[R5]≤[R6] CICLO         Controlla se è stato trovato un numero più
                                             piccolo
           Move                R5, R6        Aggiorna il numero più piccolo trovato finora
           Branch              CICLO
FATTO:     Store               R5, (R2)      Immagazzina il numero più piccolo in MINIMO
           ORIGIN              1000
MINIMO:    RESERVE             4             Spazio per il numero più piccolo trovato
N:         DATAWORD            7             Numero di elementi della lista
ELEMENTI:  DATAWORD            4,5,3,6,1,8,2 Elementi della lista
           END
```

Figura P2.2 Programma per il Problema risolto 2.2.

Problema 2.3

Scrivere un programma in stile RISC che converta un intero decimale senza segno a n cifre in un numero binario. Il numero decimale è dato mediante n caratteri ASCII, analogamente al caso in cui il numero sia fornito digitandolo su una tastiera. La locazione di memoria N contiene n, la stringa ASCII inizia in DECIMALE e il numero convertito è immagazzinato in BINARIO. Non si richiede l'inclusione di questi dati nel programma.

Soluzione

Consideriamo un numero decimale di quattro cifre, $D = d_3d_2d_1d_0$. Il valore di questo numero è $[(d_3 \times 10 + d_2) \times 10 + d_1] \times 10 + d_0$. Questa rappresentazione del numero è la base per la tecnica di conversione usata nel programma di Figura P2.3. Notiamo che ciascun carattere ASCII viene convertito in una cifra in notazione decimale codificata in binario o BCD (*Binary Coded Decimal*) prima che sia usato nel calcolo. Assumiamo che il valore convertito possa essere rappresentato da non più di 32 bit.

Problema 2.4

Si consideri una matrice di numeri A(i,j), in cui l'indice delle righe i va da 0 a $n - 1$ e l'indice delle colonne j va da 0 a $m - 1$. La matrice è immagazzinata nella memoria di un calcolatore una riga dopo l'altra, con gli elementi di ogni riga che occupano m parole di locazioni consecutive. Si assuma, inoltre, che la memoria sia indirizzabile per byte e che la lunghezza della


```
           Load                R2, N         Inizializza il contatore R2 a n
           Move                R3, #DECIMALE R3 punta alle cifre ASCII
           Clear               R4            R4 conterrà il numero binario
CICLO:     LoadByte            R5, (R3)      Preleva la prossima cifra ASCII
           And                 R5, R5, #0x0F Forma la cifra BCD
           Add                 R4, R4, R5    Aggiungila al risultato intermedio
           Add                 R3, R3, #1    Incrementa il puntatore alle cifre
           Subtract            R2, R2, #1    Decrementa il contatore
           Branch_if_[R2]=0    FATTO
           Multiply            R4, R4, #10   Moltiplica per 10
           Branch              CICLO         Reitera se non ancora finito
FATTO:     Store               R4, BINARIO   Immagazzina il risultato nella locazione BINARIO
```

Figura P2.3 Programma per il Problema risolto 2.3.

parola sia di 32 bit. Scrivere un sottoprogramma in stile RISC per sommare la colonna x alla colonna y, elemento per elemento, inserendo gli elementi somma nella colonna y. Gli indici x e y sono passati al sottoprogramma nei registri R2 e R3. I parametri n e m sono passati al sottoprogramma nei registri R4 e R5, e l'indirizzo dell'elemento A(0,0) è passato nel registro R6.

Soluzione

Un possibile sottoprogramma è mostrato in Figura P2.4, che include un frammento di programma in cui si effettua la chiamata. Abbiamo assunto che i valori x, y, n e m siano immagazzinati nelle locazioni di memoria X, Y, N, M. Inoltre, gli elementi della matrice sono immagazzinati in parole consecutive che iniziano alla locazione MATRICE, che è l'indirizzo dell'elemento A(0,0). I commenti nel programma indicano lo scopo delle singole istruzioni (si veda la Figura P2.5b).

Problema 2.5

Si vuole *ordinare* in modo alfabetico crescente una *stringa di caratteri* (o di byte) in memoria. Si supponga che la stringa abbia lunghezza $n \geq 2$ e che gli elementi siano lettere maiuscole da A a Z codificate in ASCII (ovviamente nella stringa ci possono essere lettere ripetute). Dalla Tabella 1.1 si vede che il codice ASCII è a 7 bit (o 8 bit nella forma estesa) e che l'ordinamento alfabetico delle lettere (cioè A < B < ... < Z) coincide con quello numerico dei byte. Siccome il byte ha 8 bit mentre il codice ASCII ne ha solo 7, per accomodare il carattere ASCII nel byte si conviene di azzerarne l'ottavo bit, quello più significativo. In conclusione ordinare alfabeticamente la stringa significa semplicemente ordinarne numericamente gli elementi di tipo byte.

Si supponga, inoltre, che gli elementi della stringa stiano in byte consecutivi in una zona di memoria delimitata dagli indirizzi STRINGA e STRINGA + $n - 1$ (entrambi indirizzi di byte) e che la parola (a 32 bit) di indirizzo N contenga il valore n. L'ordinamento è fatto *in loco*, cioè la stringa ordinata va a sostituire quella originale che non viene conservata.

L'algoritmo di ordinamento è quello di *selezione diretta* (*straight selection sort*), forse il più semplice dei tanti esistenti (ma non il più efficiente). Funziona così: prima si sposta in fondo alla stringa un elemento massimo, cioè lo si permuta con quello che si trova all'indirizzo STRINGA + $n - 1$; poi si ripete il procedimento sul prefisso di lunghezza $n - 1$ spostando in fondo (al prefisso) un elemento massimo, cioè permutandolo con quello che si trova all'indirizzo STRIN-

```
        Load          R2, X          Carica il valore x
        Load          R3, Y          Carica il valore y
        Load          R4, N          Carica il valore n
        Load          R5, M          Carica il valore m
        Move          R6, #MATRICE   Carica l'indirizzo di A (0,0)
        Call          SUB
        prossima istruzione
        ⋮
SUB:    Subtract      SP, SP, #4
        Store         R7, (SP)       Salva in pila il registro R7
        LShiftL       R5, R5, #2     Determina la distanza in byte tra elementi
                                      consecutivi in una colonna
        Subtract      R3, R3, R2     Calcola y − x
        LShiftL       R3, R3, #2     Ottieni 4(y − x)
        LShiftL       R2, R2, #2     Calcola 4x
        Add           R6, R6, R2     R6 punta ad A(0,x)
        Add           R7, R6, R3     R7 punta ad A(0,y)
CICLO:  Load          R2, (R6)       Preleva il prossimo numero della colonna x
        Load          R3, (R7)       Preleva il prossimo numero della colonna y
        Add           R2, R2, R3     Somma i numeri e immagazzina il risultato
        Store         R2, (R7)
        Add           R6, R6, R5     Incrementa il puntatore alla colonna x
        Add           R7, R7, R5     Incrementa il puntatore alla colonna y
        Subtract      R4, R4, #1     Decrementa il contatore di riga
        Branch_if_[R4]>0  CICLO      Reitera se non ancora finito
        Load          R7, (SP)       Ripristina R7
        Add           SP, SP, #4
        Return                       Rientra al programma chiamante
```

Figura P2.4 Programma per il Problema risolto 2.4.

GA + n − 2; e così via finché tutta la stringa non sia stata ordinata. La Figura P2.5a mostra il programma di ordinamento codificato in linguaggio di alto livello C: la stringa è dichiarata come vettore di caratteri e l'algoritmo vero e proprio si articola in una coppia di cicli "for" annidati. Per ogni prefisso di stringa compreso tra gli estremi STRINGA[0] e STRINGA[p] (inclusi) l'elemento STRINGA[p] viene confrontato con tutti gli altri (nel prefisso), e se se ne trova uno maggiore li si scambia. È chiaro come, terminata la scansione del prefisso, l'elemento STRINGA[p] in fondo debba essere massimo. Siccome si ripete il procedimento per ogni prefisso, sempre più breve, alla fine la stringa è senz'altro tutta quanta ordinata in senso crescente.

La versione in C del programma scandisce a ritroso la stringa decrementando l'indice verso zero. Ciò viene fatto per semplificare il ciclo di scansione, perché per sapere quando sia concluso basta verificare se l'indice sia sceso a zero.

Scrivere un programma in stile CISC che implementa questo compito di ordinamento.

Soluzione

Si presenta un possibile programma in Figura P2.5b.


```
#define LUN 100          /* lunghezza max della stringa      */
char STRINGA[LUN];       /* stringa di caratteri (byte)      */
int N = LUN;             /* lunghezza della stringa          */
int p, q;                /* indici di scansione stringa      */
int TEMP;                /* var. temporanea di scambio       */
...
/* parte dell'algoritmo da tradurre in lin. mach.            */
/* scandisci la stringa dal fondo                            */
for (p = N − 1; p > 0; p = p − 1) {
  /* scandisci la stringa dalla posizione p − 1              */
  for (q = p − 1; q >= 0; q = q − 1) {
    /* risulta sempre che q < p, e pertanto                  */
    /* se elemento in pos. q > elemento in pos. p            */
    if (STRINGA[q] > STRINGA[p]) {
      /* scambia i due elementi in pos. p e q                */
      TEMP = STRINGA[q];
      STRINGA[q] = STRINGA[p];
      STRINGA[p] = TEMP;
    } /* if                                                  */
    /* decrementa indice q e torna al ciclo int.             */
  } /* for                                                   */
  /* decrementa indice p e torna al ciclo est.               */
} /* for                                                     */
```

Figura P2.5a Programma di ordinamento in linguaggio C per il Problema risolto 2.5.

```
        Move          R2, #STRINGA   Carica STRINGA nel registro di base R2
        Move          R3, N          Inizializza il registro R3, indice del ciclo esterno,
        Subtract      R3, #1          a p = n − 1
ESTERNO: Move         R4, R3         Inizializza il registro R4, indice del ciclo interno,
        Subtract      R4, #1          a q = p − 1
        MoveByte      R5, (R2,R3)    Carica STRINGA(p) in R5, che contiene
                                      il massimo attuale della sottolista
INTERNO: CompareByte  (R2,R4), R5    Se STRINGA(q) ≤ STRINGA(p)], salta a
        Branch ≤ 0    CONTINUA        CONTINUA senza scambiare
        MoveByte      R6, (R2,R4)    Altrimenti, scambia STRINGA(q) con
        MoveByte      (R2,R4), R5     STRINGA(p) e carica il nuovo massimo in R5
        MoveByte      (R2,R3), R6
        MoveByte      R5, R6         Il registro R6 serve come TEMP (fine scambio)
CONTINUA: Decrement   R4             Decrementa i registri indice R4 e R3, che servono
        Branch ≥ 0    INTERNO         anche come contatori di ciclo q e p,
        Decrement     R3              rispettivamente, e reitera se i cicli non sono finiti
        Branch > 0    ESTERNO
```

Figura P2.5b Un programma di ordinamento di byte per il Problema risolto 2.5.

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

2.1 F Data una sequenza binaria in una locazione di memoria, è possibile dire se questa sequenza rappresenti un'istruzione macchina o un numero?

2.2 F Si consideri un calcolatore che ha una memoria indirizzabile per byte organizzata in parole da 32 bit in base allo schema di indirizzamento crescente dei byte. Un programma legge i caratteri ASCII digitati su una tastiera e li immagazzina in byte di locazioni consecutive, a partire dalla locazione 1000. Mostrare il contenuto delle due parole di memoria alle locazioni 1000 e 1004 dopo che la parola "Computer" sia stata digitata.

2.3 F Ripetere il Problema proposto 2.2 per lo schema di indirizzamento decrescente dei byte.

2.4 F I registri R4 e R5 contengono i numeri decimali 2000 e 3000 prima che i seguenti modi di indirizzamento siano usati per accedere a un operando in memoria. Qual è l'indirizzo effettivo (EA) in ciascun caso?

a. 12(R4)
b. (R4,R5)
c. 28(R4,R5)
d. (R4)+
e. −(R4)

2.5 F Scrivere un programma in stile RISC che calcoli l'espressione:

$$\text{SOMMA} = 580 + 68400 + 80000$$

2.6 F Scrivere un programma in stile CISC per il compito del Problema proposto 2.5.

2.7 F Scrivere un programma in stile RISC che calcoli l'espressione:

$$\text{RISPOSTA} = A \times B + C \times D$$

2.8 F Scrivere un programma in stile CISC per il compito del Problema proposto 2.7.

2.9 M Riscrivere il ciclo di addizione nella Figura 2.8 in modo da accedere ai numeri della lista in ordine inverso; ovvero il primo numero a cui si accede è l'ultimo nella lista, e l'ultimo numero a cui si accede è alla locazione di memoria NUM1. Cercare di ottenere il modo più efficiente per determinare la terminazione del ciclo. Il nuovo ciclo eseguirà più velocemente del ciclo in Figura 2.8?

2.10 M La lista di voti per gli studenti mostrata in Figura 2.10 viene modificata per contenere j voti per altrettante prove per ciascuno studente. Assumere che ci siano n studenti. Scrivere un programma in stile RISC per calcolare le somme dei voti di tutti gli studenti per ciascuna prova e immagazzinare queste somme nelle parole di locazioni di memoria agli indirizzi SOMMA, SOMMA + 4, SOMMA + 8, ... Il numero di prove, j, è più grande del numero di registri del processore, quindi il tipo di programma mostrato in Figura 2.11 per il caso con 3 prove non può essere usato. Usare due cicli annidati. Il ciclo più interno dovrebbe accumulare la somma per una certa prova, e il ciclo più esterno dovrebbe essere iterato per scorrere il numero di prove, j. Assumere che l'area di memoria usata per immagazzinare le somme sia stata inizialmente azzerata.

2.11 M Scrivere un programma in stile RISC che trovi il numero di occorrenze di interi negativi in una lista di n interi a 32 bit e immagazzini il conteggio nella locazione NUMNEG. Il valore n è immagazzinato nella locazione di memoria N e il primo intero della lista è memorizzato nella locazione NUMERI. Includere le direttive di assemblatore necessarie e una lista di esempio che contenga sei numeri, alcuni dei quali negativi.

2.12 F Entrambi i seguenti frammenti di codice fanno sì che il valore 300 sia immagazzinato nella locazione 1000, ma in momenti diversi:

```
ORIGIN      1000
DATAWORD    300
```

e:

```
Move    R2, #1000
Move    R3, #300
Store   R3, (R2)
```

Spiegare la differenza.

2.13 F Scrivere un programma assemblativo nello stile della Figura 2.13 per il programma in Figura 2.11. Assumere lo schema di dati della Figura 2.10.

2.14 F Scrivere un programma in stile CISC per il compito nel Problema risolto 2.1. Può essere nella memoria al massimo un operando di un'istruzione.

2.15 F Scrivere un programma in stile CISC per il compito nel Problema risolto 2.2. Può essere nella memoria al massimo un operando di un'istruzione.

2.16 M Scrivere un programma in stile CISC per il compito nel Problema risolto 2.3. Può essere nella memoria al massimo un operando di un'istruzione.

2.17 M Scrivere un programma in stile CISC per il compito nel Problema risolto 2.4. Può essere nella memoria al massimo un operando di un'istruzione.

2.18 M Scrivere un programma in stile RISC per il compito nel Problema risolto 2.5.

2.19 F In un programma il registro R5 è usato per puntare alla cima della pila contenente numeri a 32 bit. Scrivere una sequenza di istruzioni usando i modi di indirizzamento con indice e spiazzamento, con autoincremento e con autodecremento per effettuare ciascuno dei seguenti compiti:

a. spilare i due elementi in cima alla pila, sommarli e quindi impilare il risultato;
b. copiare il quinto elemento dalla cima nel registro R3;
c. spilare dieci elementi dalla pila.

Per ciascun caso assumere che la pila contenga dieci o più elementi.

2.20 M Mostrare il contenuto della pila del processore e il contenuto del puntatore alla pila, SP, subito dopo che ognuna delle istruzioni sotto riportate sia stata eseguita nel programma in Figura 2.18. Assumere che [SP] = 1000 al Livello 1, prima che inizi l'esecuzione del programma chiamante.

a. La seconda istruzione Store nel sottoprogramma.
b. L'ultima istruzione Load nel sottoprogramma.
c. L'ultima istruzione Store nel programma chiamante.

2.21 M Si considerino le seguenti possibilità per conservare l'indirizzo di rientro da un sottoprogramma:

a. in un registro del processore;
b. in una locazione di memoria associata alla chiamata, così che una locazione diversa verrà usata quando un sottoprogramma sarà chiamato da posti diversi;
c. su una pila.

Quale di queste possibilità supporta annidamento di sottoprogrammi e quale supporta ricorsione dei sottoprogrammi (ovvero, un sottoprogramma che chiama se stesso)?

2.22 M In aggiunta alla pila del processore, in alcuni programmi può essere conveniente usare un'altra pila. Alla seconda pila è generalmente allocato un ammontare fissato di spazio di memoria. In questo caso è importante evitare di impilare un elemento quando la pila ha raggiunto la sua dimensione massima. Inoltre, è importante evitare di tentare di spilare un elemento da una pila vuota, a seguito di un errore di programmazione. Scrivere due brevi sottoprogrammi in stile RISC, chiamati IMPILASICURA e SPILASICURA, per impilare e spilare controllando le apposite condizioni per evitare questi due possibili errori. Assumere che l'elemento da impilare/spilare sia posizionato nel registro R2, e che il registro R5 serva da puntatore alla pila per questa pila dell'utente. La pila è piena se il suo elemento in cima è memorizzato nella locazione CIMA ed è vuota se l'ultimo elemento spilato era memorizzato nella locazione FONDO. I sottoprogrammi dovrebbero saltare a ERROREPIENO e ERROREVUOTO, rispettivamente, se si verificano i suddetti errori. Tutti gli elementi hanno come dimensione una parola, e la pila cresce verso locazioni con indirizzi più bassi.

2.23 M Ripetere il Problema proposto 2.22 per sottoprogrammi in stile CISC che possono usare i modi di indirizzamento con autoincremento e con autodecremento.

2.24 D Un'altra utile struttura dati simile alla pila è chiamata *coda FIFO* (*FIFO queue*). Si immagazzinano, o *accodano* (*enqueue*), e prelevano, o *scodano* (*dequeue*), i dati con disciplina "il primo a entrare è il primo a uscire" o FIFO (*First In First Out*). Quindi se si assume che una coda cresca nella direzione di indirizzi crescenti in memoria, che è la pratica comune, i nuovi dati saranno accodati dalla parte posteriore (indirizzo più alto) e scodati dalla parte anteriore (indirizzo più basso) della coda.

Ci sono due differenze importanti tra come una pila e una coda sono realizzate. Un estremo della pila è fissato (il fondo), mentre l'altro sale e scende quando i dati vengono impilati e spilati. Un singolo puntatore è necessario per puntare alla cima della pila in qualsiasi momento. D'altra parte, entrambi gli estremi di una coda si muovono verso indirizzi più alti quando i dati sono accodati dalla parte posteriore e scodati dalla parte anteriore. Quindi due puntatori sono necessari per tenere traccia dei due estremi della coda.

Si voglia realizzare in memoria una coda FIFO di byte, in modo da occupare una regione fissata di k byte. I puntatori necessari sono un puntatore IN e un puntatore OUT. Il puntatore

IN tiene traccia della locazione dove il prossimo byte dovrà essere accodato sul retro della coda, e il puntatore OUT tiene traccia della locazione contenente il prossimo byte che dovrà essere scodato dalla parte anteriore della coda.

a. A mano a mano che gli elementi sono accodati, essi sono posti a indirizzi via via più alti sino a raggiungere la fine della regione di memoria. Cosa accade dopo, quando si vuole accodare un nuovo elemento?
Suggerimento: per riutilizzare le locazioni di memoria liberate dalle operazioni di scodamento effettuate nel frattempo, si impieghi la regione di memoria assegnata come un *buffer circolare*, ovvero quando il puntatore IN raggiunge l'indirizzo di memoria più alto della regione, al successivo accodamento lo si riporta all'indirizzo più basso, e similmente con il puntatore OUT per le operazioni di scodamento.
b. Scegliere una definizione appropriata per i puntatori IN e OUT, indicando a cosa puntano nella struttura dati. Usare un semplice diagramma per illustrare la risposta.
c. Mostrare che se lo stato della coda è descritto solo dai due puntatori, impiegando la memoria assegnata come un buffer circolare, le situazioni quando la coda è completamente piena e quando è completamente vuota sono indistinguibili.
d. Quale condizione si dovrebbe aggiungere per risolvere il problema mostrato nel punto **c**?
e. Proporre una procedura per manipolare i due puntatori IN e OUT per accodare e scodare elementi.

2.25 M Si consideri la struttura a coda descritta nel Problema proposto 2.24. Scrivere i sottoprogrammi ACCODA e SCODA che trasferiscono dati tra un registro del processore e la coda. Fare attenzione a ispezionare e aggiornare lo stato della coda e i puntatori ogni volta che si tenti di effettuare o si porti a termine un'operazione sulla coda.

2.26 M Il calcolo del prodotto scalare è stato discusso nel Paragrafo 2.12.1. Questo tipo di calcolo può essere usato nel seguente compito di elaborazione di un segnale. Una sequenza temporale di segnali in ingresso IN(0), IN(1), IN(2), IN(3), ... è elaborata mediante un vettore peso a tre elementi [WT(0), WT(1), WT(2)] = (1/8, 1/4, 1/2) per produrre una sequenza temporale del segnale in uscita OUT(0), OUT(1), OUT(2), OUT(3), ... come segue:

$$\mathrm{OUT}(0) = \mathrm{WT}(0) \times \mathrm{IN}(0) + \mathrm{WT}(1) \times \mathrm{IN}(1) + \mathrm{WT}(2) \times \mathrm{IN}(2)$$

$$\mathrm{OUT}(1) = \mathrm{WT}(0) \times \mathrm{IN}(1) + \mathrm{WT}(1) \times \mathrm{IN}(2) + \mathrm{WT}(2) \times \mathrm{IN}(3)$$

$$\mathrm{OUT}(2) = \mathrm{WT}(0) \times \mathrm{IN}(2) + \mathrm{WT}(1) \times \mathrm{IN}(3) + \mathrm{WT}(2) \times \mathrm{IN}(4)$$

$$\mathrm{OUT}(3) = \mathrm{WT}(0) \times \mathrm{IN}(3) + \mathrm{WT}(1) \times \mathrm{IN}(4) + \mathrm{WT}(2) \times \mathrm{IN}(5)$$

Tutti i valori dei segnali e dei (reciproci dei) pesi sono numeri a 32 bit con segno. Pesi, valori in ingresso e in uscita sono immagazzinati nella memoria a partire dalle locazioni WT, IN e OUT, rispettivamente. Scrivere un programma in stile RISC per calcolare e memorizzare i primi n valori in uscita, dove n è memorizzato alla locazione N.

Suggerimento: Per fare le moltiplicazioni possono essere usati gli scorrimenti aritmetici verso destra.

2.27 M Scrivere un sottoprogramma MEMCOPY per copiare una sequenza di byte da un'area di memoria a un'altra. Il sottoprogramma dovrebbe accettare tre parametri di input mediante i registri per rappresentare l'indirizzo *da*, l'indirizzo *a*, e la *lunghezza* della sequenza da copiare. Le due aree possono sovrapporsi. In tutti i casi tranne uno, il sottoprogramma dovrebbe copiare i byte secondo l'ordine crescente degli indirizzi. Nel caso in cui l'indirizzo *a* cade dentro la sequenza dei byte da copiare, ovvero quando l'indirizzo *a* è tra l'indirizzo *da*

e *da* + *lunghezza* − *1*, il sottoprogramma deve copiare i byte secondo l'ordine decrescente degli indirizzi partendo dalla fine della sequenza di byte da copiare, per evitare di sovrascrivere i byte non ancora copiati.

2.28 M Scrivere un sottoprogramma MEMCOMP per effettuare un confronto byte per byte di due sequenze di byte nella memoria principale. Il sottoprogramma dovrebbe accettare tre parametri di input inseriti nei registri per rappresentare il *primo* indirizzo, il *secondo* indirizzo e la *lunghezza* delle sequenze da confrontare. Dovrebbe usare un registro per fornire quale valore di ritorno il conteggio del numero di confronti non corrispondenti.

2.29 M Scrivere un sottoprogramma chiamato ESCLAMA che accetta un singolo parametro in un registro che rappresenta l'indirizzo iniziale STRINGA nella memoria principale per una stringa di caratteri ASCII in byte consecutivi che rappresenta un insieme di frasi, con il carattere di controllo NUL (valore 0) alla fine della stringa. Il sottoprogramma dovrebbe scandire la stringa che inizia all'indirizzo STRINGA e sostituire ogni occorrenza di un punto (.) con un punto esclamativo (!).

2.30 M Scrivere un sottoprogramma chiamato MAIUSCOLE che accetta un parametro in un registro che rappresenta l'indirizzo iniziale STRINGA nella memoria principale per una stringa di caratteri ASCII in byte consecutivi, con il carattere di controllo NUL (valore 0) alla fine della stringa. Il sottoprogramma dovrebbe scandire la stringa che inizia all'indirizzo STRINGA e sostituire ogni occorrenza di una lettera minuscola (a-z) con la corrispondente lettera maiuscola (A-Z).

2.31 M Scrivere un sottoprogramma chiamato PAROLE che accetta un parametro in un registro che rappresenta l'indirizzo iniziale STRINGA nella memoria principale per una stringa di caratteri ASCII in byte consecutivi, con il carattere di controllo NUL (valore 0) alla fine della stringa. La stringa rappresenta testo in italiano con il carattere spazio tra le parole. Il sottoprogramma deve determinare il numero di parole nella stringa (escludendo i caratteri di punteggiatura). Il risultato deve essere restituito al programma chiamante in un registro.

2.32 D Scrivere un sottoprogramma chiamato INSERISCI che mette un numero nella posizione corretta in una lista di numeri positivi memorizzata in ordine di valori crescenti. Al sottoprogramma dovrebbero essere passati tre parametri in ingresso, mediante i registri del processore, per rappresentare l'indirizzo iniziale della lista ordinata di numeri, la lunghezza della lista e il nuovo valore da inserire nella lista. Il sottoprogramma dovrebbe individuare la posizione appropriata nella lista per il nuovo valore, quindi scorrere di una posizione verso l'alto tutti i numeri più grandi per creare lo spazio atto a memorizzare il nuovo valore nella lista.

2.33 D Scrivere un sottoprogramma chiamato INSERISCIORDINA che usa ripetutamente il sottoprogramma INSERISCI del Problema proposto 2.32 per ottenere da una lista non ordinata di numeri una nuova lista con gli stessi numeri in ordine crescente. Il sottoprogramma dovrebbe accettare tre parametri in ingresso, mediante i registri, per rappresentare l'indirizzo di partenza VECCHIALISTA per la sequenza di numeri non ordinata, la lunghezza della lista e l'indirizzo di partenza NUOVALISTA per la sequenza ordinata di numeri.

2.34 F Si considerino i programmi assemblativi generici per la somma di una lista di numeri proposti, rispettivamente in stile RISC e CISC, nelle Figure 2.8 e 2.26. Tenendo conto del confronto condotto nell'esempio mostrato nell'Approfondimento applicativo 2.6 fra caratteristiche delle ISA dei quattro processori considerati, produrre una stima approssimativa del numero di istruzioni di programmi NIOS II, ARM, ColdFire e IA-32 che concretamente realizzino i suddetti programmi assemblativi generici.

2.35 M Verificare la bontà della risposta al problema precedente producendo i quattro programmi concreti indicati nel problema.

2.36 F Estendere i programmi prodotti in risposta al problema precedente con direttive di assemblatore concrete, corrispondenti a quelle generiche impiegate nel programma mostrato in Figura 2.13, ma eventualmente aggiuntive se necessario.

2.37 M Si consideri il programma mostrato in Figura P2.2, quale soluzione al Problema risolto 2.2. Produrre innanzitutto un programma assemblativo generico che lo risolva in stile CISC, con il vincolo che non più di un operando di ciascuna istruzione sia collocato in memoria (si veda il Problema proposto 2.15), quindi riformulare i due programmi generici in quattro programmi NIOS II, ARM, ColdFire e IA-32, dotati delle necessarie direttive di assemblatore.

2.38 M Si considerino i programmi assemblativi generici per la ricerca di una stringa proposti nel Paragrafo 2.12.2, rispettivamente in stile RISC e CISC, nelle Figure 2.30 e 2.31. Tenendo conto del confronto condotto nell'esempio mostrato nell'Approfondimento applicativo 2.6 fra caratteristiche delle ISA dei quattro processori considerati, produrre una stima approssimativa del numero di istruzioni di programmi NIOS II, ARM, ColdFire e IA-32 che concretamente realizzino i suddetti programmi assemblativi generici.

2.39 D Verificare la bontà della risposta al problema precedente producendo i quattro programmi concreti indicati nel problema.

3 Operazioni di ingresso e uscita

Problemi risolti

Problema 3.1

Si assuma che la locazione di memoria BINARIO contenga una sequenza di 32 bit. Si vogliono mostrare questi bit come otto cifre esadecimali su uno schermo che ha l'interfaccia mostrata in Figura 3.3. Scrivere un programma che realizza questo compito.

Soluzione

Innanzitutto sarà necessario convertire la sequenza di 32 bit in cifre esadecimali che siano rappresentate come caratteri codificati in ASCII. Un modo semplice per effettuare la conversione è usare l'approccio di *ricerca in tabella* (*table-lookup*). Dovremo costruire una tabella con 16 elementi che fornisce il codice ASCII per ciascuna delle possibili cifre esadecimali. Quindi, per ciascun frammento di quattro bit della sequenza BINARIO, potremo trovare il carattere corrispondente nella tabella e memorizzarlo in un blocco di byte di memoria a partire dalla locazione ESA. Infine, gli otto caratteri a iniziare da ESA saranno inviati allo schermo.

Le Figure P3.1a e P3.1b forniscono i programmi in stile RISC e CISC, rispettivamente, per il compito richiesto. I commenti descrivono le singole attività svolte.

Problema 3.2

Si consideri il compito descritto nell'Esempio 3.1 del Paragrafo 3.2. Si assuma che il circuito del cronometro includa un contatore in avanti e indietro a 32 bit pilotato da un clock a 100 MHz. Il contatore può essere programmato per contare da un certo valore iniziale. L'interfaccia I/O del cronometro è mostrata in Figura P3.2a. Essa contiene quattro registri.

- TIM_STATUS indica lo stato corrente del cronometro, dove:
  - il bit TON è posto a 1 quando il cronometro sta girando;
  - il bit ZERO è posto a 1 quando il cronometro raggiunge il valore zero;
  - il bit TIRQ è posto a 1 quando il cronometro segnala una richiesta di interruzione, il che accade quando il contenuto del contatore raggiunge zero e le interruzioni del cronometro sono abilitate.

  L'operazione di lettura del registro di stato azzera automaticamente i bit ZERO e TIRQ.
- TIM_CONT controlla la modalità di lavoro, dove:
  - il bit UP è posto a 1 per fare in modo che il conteggio del contatore avvenga incrementando il suo contenuto; quando questo bit è posto a 0, il contenuto del contatore viene progressivamente decrementato;
  - il bit FREE è posto a 1 per ottenere una modalità di esecuzione continua, per cui il contatore è automaticamente ricaricato con il valore iniziale quando il suo contenuto raggiunge 0;
  - il bit RUN è posto a 1 per avviare il conteggio del contatore; è posto a 0 per fermare il contatore;
  - il bit TIE è posto a 1 per abilitare le interruzioni del cronometro.

	Load	R2, BINARIO	Carica il numero binario
	Move	R3, #8	R3 è un contatore di cifre che è posto a 8
	Move	R4, #ESA	R4 punta alle cifre esadecimali
CICLO:	RotateL	R2, R2, #4	Ruota le cifre di ordine più alto sulle posizioni di ordine più basso
	And	R5, R2, #0xF	Estrai la prossima cifra
	LoadByte	R6, TABELLA(R5)	Ottieni il codice ASCII della cifra e memorizzalo nella locazione del numero in ESA
	StoreByte	R6, (R4)	
	Subtract	R3, R3, #1	Decrementa il contatore di cifre
	Add	R4, R4, #1	Incrementa il puntatore alle cifre esadecimali
	Branch_if_[R3]>0	CICLO	Reitera se non è l'ultima cifra
DISPLAY:	Move	R3, #8	
	Move	R4, #ESA	
CICLO_DISP:	LoadByte	R5, DISP_STATUS	Attendi che lo schermo sia pronto
	And	R5, R5, #4	Controlla la condizione di stato DOUT
	Branch_if_[R5]=0	CICLO_DISP	
	LoadByte	R6, (R4)	Ottieni il prossimo carattere ASCII e invialo sullo schermo
	StoreByte	R6, DISP_DATA	
	Subtract	R3, R3, #1	Decrementa il contatore
	Add	R4, R4, #1	Incrementa il puntatore ai caratteri
	Branch_if_[R3]>0	CICLO_DISP	Reitera finché tutti i caratteri sono stati visualizzati
	prossima istruzione		
	ORIGIN	1000	
ESA:	RESERVE	8	Spazio per cifre codificate in ASCII
TABELLA:	DATABYTE	0x30,0x31,0x32,0x33	Tabella per la conversione in codice ASCII
	DATABYTE	0x34,0x35,0x36,0x37	
	DATABYTE	0x38,0x39,0x41,0x42	
	DATABYTE	0x43,0x44,0x45,0x46	

Figura P3.1a Un programma in stile RISC per il Problema risolto 3.1.

- TIM_INIT contiene il valore iniziale del contatore.
- TIM_COUNT contiene il valore corrente del contatore.

Scrivere un programma che realizzi il compito desiderato. Usare i registri di controllo del processore mostrati in Figura 3.7.

Soluzione

Per ottenere una richiesta di interruzione ogni dieci secondi, è necessario contare per 10^9 cicli di clock. Questo può essere realizzato scrivendo questo valore nel registro TIM_INIT, e quindi fare in modo che il contatore decrementi il suo valore e segnali un'interruzione quando il

```
            Move        R2, BINARIO        Carica il numero binario
            Move        R3, #8             R3 è un contatore di cifre che è posto a 8
            Move        R4, #ESA           R4 punta alle cifre esadecimali
CICLO:      RotateL     R2, #4             Ruota le cifre di ordine più alto sulle
                                             posizioni di ordine più basso
            Move        R5, R2
            And         R5, #0xF           Estrai la prossima cifra
            MoveByte    (R4)+, TABELLA(R5) Ottieni il codice ASCII per la cifra e
                                             memorizzalo nella locazione del numero
                                             in ESA
            Subtract    R3, #1             Decrementa il contatore di cifre
            Branch>0    CICLO              Reitera se non è l'ultima cifra
DISPLAY:    Move R3, #8
            Move        R4, #ESA
CICLO_DISP: TestBit     DISP_STATUS, #2    Attendi che lo schermo sia pronto
            Branch=0    CICLO_DISP
            MoveByte    DISP_DATA, (R4)+   Invia il prossimo carattere sullo schermo
            Subtract    R3, #1             Decrementa il contatore
            Branch>0    CICLO_DISP         Reitera finché tutti i caratteri sono stati
                                             visualizzati
            next instruction

            ORIGIN      1000
ESA:        RESERVE     8                  Spazio per cifre codificate in ASCII
TABELLA:    DATABYTE    0x30,0x31,0x32,0x33 Tabella per la conversione in codice ASCII
            DATABYTE    0x34,0x35,0x36,0x37
            DATABYTE    0x38,0x39,0x41,0x42
            DATABYTE    0x43,0x44,0x45,0x46
```

Figura P3.1b Un programma in stile CISC per il Problema risolto 3.1.

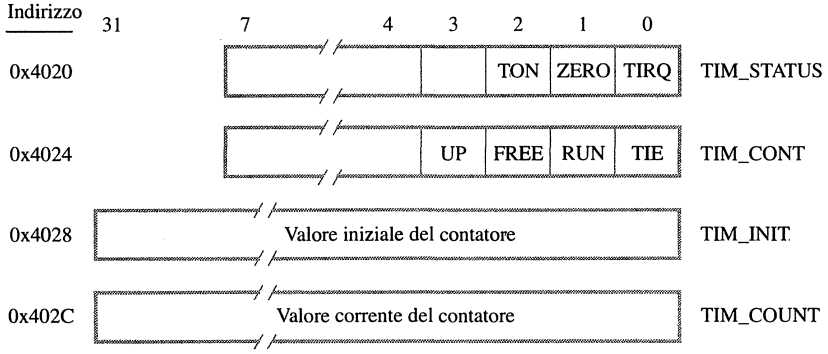


Figura P3.2a Registri nell'interfaccia del cronometro per il Problema risolto 3.2.

Gestore delle interruzioni

```
ILOC:       Subtract        SP, SP, #8         Salva i registri
            Store           LINK_reg, 4(SP)
            Store           R2, (SP)
            MoveControl     R2, IPENDING       Controlla il contenuto di IPENDING
            And             R2, R2, #8         Controlla se la richiesta viene dal
                                                 cronometro
            Branch_if_[R2]=0 PROSSIMO
            LoadByte        R2, TIM_STATUS     Azzera i bit TIRQ e ZERO
            Call            VISUALIZZA         Chiama la routine VISUALIZZA
PROSSIMO:   ...                                Controlla le interruzioni da altri dispositivi
                                                 di I/O
            Load            R2, (SP)           Ripristina i registri
            Load            LINK_reg, 4(SP)
            Add             SP, SP, #8
            Return-from-interrupt
```

Programma principale (Main)

```
INIZIO:     ...                                Inizializza i parametri per le ISR
            OrHigh          R2, R0, #0x3B9A    Prepara il valore iniziale del conteggio
            Or              R2, R2, #0xCA00
            Store           R2, TIM_INIT       Imposta il valore iniziale del conteggio
            Move            R2, #7             Avvia il cronometro e abilita
            StoreByte       R2, TIM_CONT         le interruzioni
            MoveControl     R2, IENABLE
            Or              R2, R2, #8         Abilita le interruzioni dal cronometro
            MoveControl     IENABLE, R2          nel registro di controllo del processore
            MoveControl     R2, PS
            Or              R2, R2, #1
            MoveControl     PS, R2             Poni a 1 il bit di abilitazione
                                                 delle interruzioni in PS
CALCOLA:    prossima istruzione
```

Figura P3.2b Un programma in stile RISC per il Problema risolto 3.2.

conteggio raggiunge zero. Il valore 10^9 può essere rappresentato dal numero esadecimale 3B9ACA00. Per ottenere l'operazione richiesta i bit FREE, RUN e TIE devono essere posti a 1, mentre il bit UP deve essere posto a 0.

Usando lo schema tracciato in Figura 3.9, possiamo realizzare il compito richiesto usando il programma in stile RISC mostrato in Figura P3.2b. Notiamo che il valore iniziale, che è un valore immediato a 32 bit, è caricato in R2 usando l'approccio spiegato nel Paragrafo 2.9.

La Figura P3.2c fornisce un programma in stile CISC che usa lo schema tracciato in Figura 3.11. In questo caso, l'operando immediato a 32 bit può essere specificato in un'istruzione singola.

Gestore delle interruzioni

ILOC:	Move	–(SP), R2	Salva i registri
	Move	–(SP), LINK_reg	
	MoveControl	R2, IPENDING	Controlla il contenuto di IPENDING
	TestBit	R2, #3	Controlla se la richiesta viene dal cronometro
	Branch=0	PROSSIMO	
	MoveByte	R2, TIM_STATUS	Azzera i bit TIRQ e ZERO
	Call	VISUALIZZA	Chiama la routine VISUALIZZA
PROSSIMO:	...		Controlla le interruzioni da altri dispositivi di I/O
	Move	LINK_reg, (SP)+	Ripristina i registri
	Move	R2, (SP)+	
	Return-from-interrupt		

Programma principale (Main)

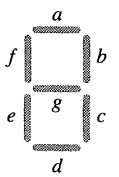
INIZIO:	...		Inizializza i parametri per le ISR
	Move	TIM_INIT, #0x3B9ACA00	Imposta il valore iniziale del conteggio
	MoveByte	TIM_CON, #7	Avvia il cronometro e abilita le interruzioni
	MoveControl	R2, IENABLE	
	Or	R2, #8	Abilita le interruzioni dal cronometro nel registro di controllo del processore
	MoveControl	IENABLE, R2	
	MoveControl	R2, PS	
	Or	R2, #1	
	MoveControl	PS, R2	Poni a 1 il bit di abilitazione delle interruzioni in PS
CALCOLA:	prossima istruzione		

Figura P3.2c Un programma in stile CISC per il Problema risolto 3.2.

Problema 3.3

Un dispositivo di uscita comunemente usato nei sistemi digitali è lo schermo a sette segmenti mostrato in Figura P3.3a. Il dispositivo consiste di sette segmenti indipendenti che possono essere illuminati applicandovi dei segnali elettrici. Si assuma che ciascun segmento venga illuminato quando vi si applica un valore logico 1. La figura mostra la sequenza di bit necessaria per visualizzare i numeri da 0 a 9.

Scrivere un programma che visualizzi i numeri rappresentati da un carattere in codifica ASCII immagazzinato in memoria alla locazione CIFRA di indirizzo 0x800. Assumere che lo schermo abbia un'interfaccia di I/O che consiste di un registro dati a 8 bit, SETTE, dove i segmenti da *a* a *g* sono connessi ai bit $SETTE_{6-0}$. Si ponga a 0 il bit $SETTE_7$. Inoltre, si assuma che l'indirizzo del registro SETTE sia 0x4030. Se il codice ASCII nella locazione



Numero	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Figura P3.3a Schermo a sette segmenti per il Problema risolto 3.3.

CIFRA rappresenta un carattere che non è un numero nell'intervallo da 0 a 9 allora lo schermo dovrebbe essere spento, ovvero tutti i segmenti saranno spenti.

Soluzione

Si può usare una tabella di ricerca per mantenere le sequenze di bit dei sette segmenti che corrispondono ai numeri da 0 a 9. La cifra codificata in ASCII è convertita in un numero di quattro bit che è usato come indice per la tabella, usando l'operazione AND. Inoltre, è necessario controllare che i quattro bit più significativi del codice ASCII siano 0011. Si noti che ciascuno dei tre indirizzi CIFRA, SETTE e TABELLA può essere rappresentato con 16 bit.

Le Figure P3.3b e 3.3c forniscono programmi in stile RISC e CISC, rispettivamente.

CIFRA	EQU	0x800	Locazione della cifra codificata in ASCII
SETTE	EQU	0x4030	Indirizzo dello schermo a 7 segmenti
	LoadByte	R2, CIFRA	Carica la cifra codificata in ASCII
	And	R3, R2, #0xF0	Estrai i bit più alti del codice ASCII
	And	R2, R2, #0x0F	Estrai il numero decimale
	Move	R4, #0x30	Controlla se i bit più alti del codice ASCII sono 0011
	Branch_if_[R3]=[R4]	ALTO	
	Move	R2, #0x0F	Non è una cifra, visualizza lo schermo vuoto
ALTO:	LoadByte	R5, TABELLA(R2)	Ottieni la sequenza di 7 segmenti
	StoreByte	R5, SETTE	Visualizza la cifra
	ORIGIN	0x1000	
TABELLA:	DATABYTE	0x7E,0x30,0x6D,0x79	Tabella che contiene le necessarie sequenze di 7 segmenti
	DATABYTE	0x33,0x5B,0x5F,0x70	
	DATABYTE	0x7F,0x7B,0x00,0x00	
	DATABYTE	0x00,0x00,0x00,0x00	

Figura P3.3b Un programma in stile RISC per il Problema risolto 3.3.

```
CIFRA    EQU        0x800                 Locazione della cifra codificata in ASCII
SETTE    EQU        0x4030                Indirizzo dello schermo a 7 segmenti
         Move       R2, CIFRA             Carica la cifra codificata in ASCII
         Move       R3, R2
         And        R3, #0xF0             Estrai i bit più alti del codice ASCII
         And        R2, #0x0F             Estrai il numero decimale
         CompareByte R3, #0x30            Controlla se i bit più alti del codice ASCII
                                           sono 0011
         Branch=0   ALTO
         Move       R2, #0x0F             Non è una cifra, visualizza lo schermo vuoto
ALTO:    MoveByte   SETTE, TABELLA(R2)    Visualizza la cifra

         ORIGIN     0x1000
TABELLA: DATABYTE   0x7E,0x30,0x6D,0x79   Tabella che contiene le necessarie sequenze
         DATABYTE   0x33,0x5B,0x5F,0x70    di 7 segmenti
         DATABYTE   0x7F,0x7B,0x00,0x00
         DATABYTE   0x00,0x00,0x00,0x00
```

Figura P3.3c Un programma in stile CISC per il Problema risolto 3.3.

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

3.1 F Il bit dello stato di ingresso in un circuito di interfaccia è posto a zero quando il registro dei dati di ingresso viene letto. Perché questo è importante?

3.2 F Scrivere un programma che visualizzi in formato esadecimale il contenuto di 10 byte in memoria centrale su una linea di uno schermo. I 10 byte sono in memoria a partire dalla locazione LOC e si mostrano due caratteri esadecimali per ciascun byte. I contenuti di byte successivi dovrebbero essere separati da uno spazio quando vengono mostrati sullo schermo.

3.3 F Qual è la differenza tra un sottoprogramma e una routine di servizio di interruzione?

3.4 F Nella prima istruzione And in Figura 3.4 il valore immediato 2 viene usato quando si controlla la variabile di condizione KIN, ma in Figura 3.5 si usa il valore immediato 1 nella prima istruzione TestBit quando si controlla la stessa variabile di condizione. Spiegare la differenza.

3.5 D Un calcolatore deve accettare caratteri dalle tastiere di 20 terminali. L'area di memoria principale che deve essere usata per immagazzinare i dati per ciascun terminale è riferita tramite un puntatore PNTR*n*, dove *n* va da 1 a 20. I dati in ingresso devono essere presi dai terminali mentre è in esecuzione un altro programma, PROG. Questo può essere realizzato in uno dei due modi seguenti.

a. Ogni T secondi, il programma PROG chiama un sottoprogramma di scansione SCANSIONE. Questo sottoprogramma controlla lo stato di ciascuno dei 20 terminali in sequenza e trasferisce i caratteri in ingresso in memoria. Quindi esso rientra a PROG.

b. Quando un carattere è pronto in un qualsiasi buffer di interfaccia dei terminali, viene generata una richiesta di interruzione. Questo causa l'esecuzione della routine INTERRUZIONE che controlla i registri di stato per trovare il primo carattere pronto, lo trasferisce e quindi rientra a PROG.

Scrivere i sottoprogrammi SCANSIONE e INTERRUZIONE. Sia c caratteri al secondo la frequenza massima di caratteri per un terminale e sia rc la frequenza media, con $r \leq 1$. Nel metodo **a.**, qual è il massimo valore di T per cui è ancora possibile garantire che nessun carattere in ingresso verrà perso? Qual è il corrispondente valore per il metodo **b.**? Stimare, in media, la percentuale di tempo passato a servire i terminali per i metodi **a.** e **b.**, per $c = 100$ caratteri al secondo e $r = 0{,}01$, 0,1, 0,5 e 1. Si assuma che SCANSIONE prenda 800ns per controllare tutti i 20 dispositivi e che un'interruzione da un dispositivo richieda 200ns per l'elaborazione.

3.6 F In Figura 3.9, il bit di abilitazione delle interruzioni in PS è assegnato per ultimo nella sezione INIZIO del programma Main. Perché? È importante l'ordine per le operazioni precedenti nella sezione INIZIO? Perché o perché no?

3.7 F Sebbene possano risultare pendenti più richieste di interruzioni, solo una richiesta verrà gestita per ciascun elemento in ILOC in Figura 3.9. Vero o falso? Spiegare.

3.8 F Un programma utente potrebbe controllare se un divisore è zero subito prima di ciascuna operazione di divisione e quindi eseguire azioni opportune senza invocare il sistema operativo. Dare motivazioni per cui questo potrebbe essere preferibile o no, rispetto a permettere la segnalazione di un'interruzione da eccezione su un'effettiva situazione di divisione per zero in un programma dell'utente.

3.9 M Si assuma che la locazione di memoria BINARIO contenga una sequenza di 16 bit. Si desidera visualizzare questi bit come una stringa di 0 e 1 su uno schermo che ha l'interfaccia mostrata in Figura 3.3. Scrivere un programma in stile RISC che realizzi questo compito.

3.10 M Scrivere un programma in stile CISC per il compito del Problema proposto 3.9.

3.11 F Modificare il programma in Figura P3.3b dove l'indirizzo di TABELLA sia 0x10100.

3.12 F Modificare il programma in Figura P3.3c dove l'indirizzo di TABELLA sia 0x10100.

3.13 M Usando lo schermo a sette segmenti di Figura P3.3a e i registri dell'interfaccia del cronometro di Figura P3.2a, scrivere un programma in stile RISC che faccia lampeggiare cifre decimali nella sequenza periodica 0, 1, 2, ..., 9, 0, ... Ciascuna cifra deve essere mostrata per un secondo. Si assuma che il contatore nel circuito del cronometro sia pilotato da un clock a 100 MHz.

3.14 M Scrivere un programma in stile CISC per il compito del Problema proposto 3.13.

3.15 D Usando due schermi a sette segmenti del tipo mostrato in Figura P3.3a e i registri dell'interfaccia del cronometro di Figura P3.2a, scrivere un programma in stile RISC che faccia lampeggiare la sequenza periodica di numeri 0, 1, 2, ..., 98, 99, 0, ... Ciascun numero deve essere mostrato per un secondo. Si assuma che il contatore nel circuito del cronometro sia pilotato da un clock a 100 MHz.

3.16 **D** Scrivere un programma in stile CISC per il compito del Problema proposto 3.15.

3.17 **D** Scrivere un programma in stile RISC che calcoli il tempo di un orologio da parete e mostri l'orario in ore (da 0 a 23) e minuti (da 0 a 59). Lo schermo consiste di quattro dispositivi a sette segmenti del tipo mostrato in Figura P3.3a. È disponibile un circuito per il cronometro che ha l'interfaccia mostrata in Figura P3.2a. Il suo contatore è pilotato da un clock a 100 MHz.

3.18 **D** Scrivere un programma in stile CISC per il compito del Problema proposto 3.17.

3.19 **M** Scrivere un programma in stile RISC che mostri il nome dell'utente con le lettere in ordine inverso. Il programma dovrebbe mostrare un simbolo di richiesta dati (*prompt*) e permettere all'utente di inserire i caratteri del suo nome da tastiera, seguiti dal ritorno carrello (CR). Il programma dovrebbe accettare una sequenza di caratteri e immagazzinarli nella memoria centrale. Dovrebbe quindi mostrare un messaggio per indicare che il nome dell'utente sarà mostrato al contrario, seguito dalla visualizzazione dei caratteri del nome dell'utente in ordine inverso.

3.20 **M** Scrivere un programma in stile CISC per il compito del Problema proposto 3.19.

3.21 **M** Scrivere un programma in stile RISC che determini se una parola inserita da un utente sulla tastiera sia palindroma, ovvero una parola che è la stessa quando i suoi caratteri sono scritti in ordine normale o inverso. Il programma dovrebbe mostrare un simbolo di richiesta dati e permettere all'utente di inserire i caratteri di una qualsiasi parola da tastiera, seguiti dal ritorno carrello (CR). Il programma dovrebbe leggere i caratteri e immagazzinarli nella memoria centrale. Dovrebbe quindi analizzare la parola per determinare se essa sia palindroma. Infine, dovrebbe mostrare un messaggio per indicare il risultato dell'analisi.

3.22 **M** Scrivere un programma in stile CISC per il compito del Problema proposto 3.21.

3.23 **D** Scrivere un programma in stile RISC che visualizzi una stringa di caratteri centrati orizzontalmente su una linea di 80 caratteri e racchiusi in un rettangolo, come mostrato qui:

```
+----------+
| Testo di esempio |
+----------+
```

La stringa di caratteri si trova in memoria centrale a partire dall'indirizzo STRINGA. Vi è un carattere di controllo NUL (valore 0) alla fine della stringa di caratteri. Se la stringa ha più di 78 caratteri (inclusi gli spazi), il programma dovrebbe troncare la stringa mostrata ai primi 78 caratteri. Il programma per determinare la lunghezza di una stringa di caratteri nel Problema risolto 2.1 può essere adattato come sottoprogramma per usarlo da parte del programma in questo problema. Si assuma che lo schermo abbia l'interfaccia mostrata in Figura 3.3.

3.24 **D** Scrivere un programma in stile CISC per il compito del Problema proposto 3.23.

3.25 **D** Scrivere un programma in stile RISC che mostri una lunga sequenza di testo codificato in caratteri ASCII con ritorno a capo automatico per entrare in linee di 80 caratteri.

Prima di mostrare la prossima parola, il programma deve determinare se rimane spazio sufficiente per essa sulla linea. Se no, la parola dovrebbe comparire all'inizio della prossima linea. Il processo di visualizzazione dovrebbe continuare fino a che il carattere di controllo NUL (valore 0) sia raggiunto alla fine della sequenza di caratteri da mostrare. Si assuma che la sequenza di caratteri non contenga caratteri di controllo a eccezione del carattere NUL alla fine, e quindi che le parole siano separate solo da un carattere spazio. Si assuma, inoltre, che lo schermo abbia l'interfaccia illustrata in Figura 3.3.

3.26 **D** Scrivere un programma in stile CISC per il compito del Problema propoto 3.25.

4 Software

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

4.1 F Scrivere un programma C che svolga le attività descritte nel Problema proposto 3.2.

4.2 M Scrivere un programma C che svolga le attività descritte nel Problema proposto 3.9.

4.3 M Scrivere un programma C che svolga le attività descritte nel Problema proposto 3.13.

4.4 D Scrivere un programma C che svolga le attività descritte nel Problema proposto 3.15.

4.5 D Scrivere un programma C che svolga le attività descritte nel Problema proposto 3.17.

4.6 D Scrivere un programma C che svolga le attività descritte nel Problema proposto 3.17, ma usando una routine di servizio dell'interruzione associata al cronometro.

4.7 D Si assuma che l'insieme delle istruzioni per un processore includa l'istruzione:

MultiplyAccumulate R*i*, R*j*, R*k*

che effettua l'operazione R*i* ← [R*i*] + [R*j*] × [R*k*] usando i registri del processore R*i*, R*j*, e R*k*. Tale istruzione è descritta nel Paragrafo 2.12.1.

Si assuma che il compilatore non usi questa istruzione quando genera il risultato in linguaggio assemblativo. Si assuma, poi, che ci siano tre variabili, X, Y e Z, definite come variabili globali in un programma C. Scrivere una funzione *mult_acc_XYZ* in linguaggio C che usi l'istruzione MultiplyAccumulate per calcolare X = X + Y × Z. Notare che le istruzioni in linguaggio assemblativo, generate dal compilatore, in questa funzione e nel programma chiamante possono usare i registri del processore per mantenere dati.

4.8 D Il Paragrafo 4.9.2 espone come i passi di ingresso e uscita di un insieme di programmi come quello mostrato in Figura 4.7 potrebbero essere sovrapposti per ridurre il tempo totale necessario a eseguirli. Si supponga che ciascuno dei sei intervalli di esecuzione delle routine di SO sia 1 unità di tempo, che ciascuna operazione su disco richieda 3 unità, che la stampa richieda 3 unità e che ciascun intervallo di esecuzione di un programma richieda 2 unità. Calcolare il migliore rapporto tra tempo di sovrapposizione e tempo di non sovrapposizione per una lunga sequenza di programmi. Ignorare i transitori iniziali e finali.

4.9 D Il Paragrafo 4.9.2 indica che l'elaborazione di un programma può essere sovrapposta a operazioni di ingresso o di uscita o di entrambi i tipi. Ignorando il tempo relativamente breve necessario per le routine di SO, qual è il migliore rapporto tra tempo di sovrapposizione e tempo di non sovrapposizione per completare l'esecuzione di un insieme di programmi, dove ogni programma ha circa uguale bilancio tra attività di ingresso, elaborazione e uscita?

4.10 M Nella discussione dei tre stati dei processi nel Paragrafo 4.9.3, sono state descritte le transizioni da Eseguibile a Esecuzione, da Esecuzione a Bloccato e da Bloccato a Eseguibile. Quali altre transizioni dirette tra questi stati sono possibili per un processo? Quali non lo sono? Spiegare sinteticamente ognuna delle scelte.

5 Struttura di base del processore

Problemi risolti

Problema 5.1

La Figura 5.11 mostra un'istruzione Add che è in esecuzione tramite cinque passi, ma nessuna azione di elaborazione viene svolta nel passo 4. Se si desidera eliminare questo passo, quali cambiamenti devono essere effettuati nel percorso dati in Figura 5.8 per renderlo possibile?

Soluzione

Il passo 4 può essere saltato inviando l'uscita dell'ALU in Figura 5.8 direttamente al registro RY. Questo può essere ottenuto aggiungendo uno o più ingressi al multiplatore MuxY e collegando questo ingresso all'uscita dell'ALU. Quindi, il risultato di un calcolo all'uscita dell'ALU viene caricato in entrambi i registri RZ e RY alla fine del passo 3. Per un'istruzione Add, o una qualsiasi altra istruzione di calcolo, il segnale di controllo RF_scrittura per il banco di registri può essere abilitato nel passo 4 per caricare il contenuto di RY nel banco di registri.

Problema 5.2

Si assuma che tutte le operazioni di accesso a memoria siano completate in un ciclo di clock in un processore che ha un clock a 1 GHz. Qual è la frequenza delle operazioni di accesso alla memoria se le istruzioni Load e Store costituiscono il 20%del conteggio dinamico delle istruzioni in un programma? (Il conteggio dinamico è il numero di esecuzioni di istruzioni, che include l'effetto di cicli, il quale può far sì che alcune istruzioni siano eseguite più di una volta.) Si assuma che tutte le istruzioni siano eseguite in 5 cicli di clock.

Soluzione

Vi è un accesso in memoria per prelevare ciascuna istruzione. Quindi il 20% delle istruzioni hanno un secondo accesso in memoria per leggere o scrivere un operando in memoria. In media, ogni istruzione ha 1,2 accessi in memoria ogni 5 cicli di clock. Quindi la frequenza di accessi in memoria è $(1{,}2/5) \times 10^9$, o 240 milioni di accessi al secondo.

Problema 5.3

Derivare le espressioni logiche per un circuito che confronta due numeri binari senza segno $X = x_2x_1x_0$ e $Y = y_2y_1y_0$ e genera tre uscite *XGY*, *XEY* e *XLY*. Una di queste uscite è posta a 1 per indicare che *X* è maggiore di, uguale a, o minore di *Y*, rispettivamente.

Soluzione

Per confrontare due numeri senza segno dovremo confrontare ciascuna locazione dei bit, a partire dal bit più significativo. Se $x_2 = 1$ e $y_2 = 0$, allora *X* è maggiore di *Y*. Se $x_2 = y_2$,

allora dovremo confrontare la locazione del bit più in basso ecc. Quindi, espressioni logiche per le tre uscite possono essere scritte come segue:

$$XGY = x_2\bar{y}_2 + \overline{(x_2 \oplus y_2)} \cdot (x_1\bar{y}_1 + \overline{(x_1 \oplus y_1)}x_0\bar{y}_0)$$

$$XEY = \overline{(x_2 \oplus y_2)} \cdot \overline{(x_1 \oplus y_1)} \cdot \overline{(x_0 \oplus y_0)}$$

$$XLY = \overline{XGY} + \overline{XEY}$$

Problema 5.4

Stilare la sequenza di azioni per un'istruzione di rientro da sottoprogramma in un processore RISC. Si assuma che l'indirizzo LINK del registro di uso generale in cui è immagazzinato l'indirizzo di rientro dal sottoprogramma venga fornito nel campo dell'istruzione collegato all'Indirizzo A del banco di registri (IR_{31-27}).

Soluzione

Quando un'istruzione viene caricata nell'IR, il contenuto del registro di uso generale il cui indirizzo è fornito dai bit IR_{31-27} viene letto e posto nel registro RA (si veda la Figura 5.18). Quindi, un'istruzione di rientro da sottoprogramma farà sì che il contenuto del registro LINK venga letto e posto nel registro RA. L'esecuzione procede come segue:

1. Indirizzo di memoria ← [PC], Leggi da memoria, Attesa MFC, IR ← Dati da memoria, PC ← [PC] + 4;
2. Decodifica istruzione, RA ← [LINK];
3. PC ← [RA];
4. Nessuna azione;
5. Nessuna azione.

Problema 5.5

Un processore ha la seguente struttura di interruzioni. Quando un'interruzione viene ricevuta, l'indirizzo di rientro da interruzione viene salvato in un registro di uso generale chiamato IRA. Il contenuto corrente del registro di stato del programma, PS, viene salvato in un registro speciale chiamato IPS, che non è un registro di uso generale. La routine di servizio dell'interruzione inizia all'indirizzo ILOC.

Si assuma che il processore esamini le interruzioni nell'ultimo passo di esecuzione di ogni istruzione. Se una richiesta di interruzione è presente e le interruzioni sono abilitate, la richiesta viene accettata. Anziché prelevare la successiva istruzione, il processore salva PC e PS e salta a ILOC. Fornire un'opportuna sequenza di passi per effettuare queste azioni. Quale hardware aggiuntivo è necessario nelle Figure da 5.18 a 5.20 a supporto dell'elaborazione delle interruzioni?

Soluzione

I primi due passi di esecuzione delle istruzioni, in cui un'istruzione viene prelevata e decodificata, non sono necessari nel caso di un'interruzione. Essi possono essere saltati, o non produrranno azioni se si desidera mantenere una sequenza di 5 passi. Il salvataggio del PC può essere fatto esattamente come per un'istruzione di chiamata a sottoprogramma. È necessario un altro ingresso a MuxC nella Figura 5.18 a cui collegare l'indirizzo del registro IRA. Per caricare l'indirizzo di partenza della routine di servizio dell'interruzione nel PC è necessario un ingresso aggiuntivo a MuxPC in Figura 5.20, a cui collegare il valore ILOC. I registri

PS e IPS dovrebbero essere collegati direttamente l'un l'altro per abilitare il trasferimento di dati fra loro. I passi di esecuzione richiesti sono:

3. PC-Temp ← [PC], PC ← ILOC, IPS ← [PS], Disabilita interruzioni;
4. RY ← [PC-Temp];
5. IRA ← [RY].

Le azioni inverse sono eseguite per un'istruzione di rientro da interruzione. Si veda il Problema proposto 5.8.

Problema 5.6

Il Problema risolto 5.5 illustra come i contenuti di PC e PS vengano salvati quando una richiesta di interruzione viene accettata. Per supportare l'annidamento di interruzioni, è necessario che una routine di servizio delle interruzioni salvi in pila questi registri, come descritto nel Paragrafo 3.2. Per far questo, il contenuto di PS, che viene salvato nel registro IPS al momento in cui l'interruzione viene accettata, deve essere trasferito in uno dei registri di uso generale, da cui può essere salvato in pila. Si assuma che due istruzioni speciali:

MoveControl R*i*, IPS

e:

MoveControl IPS, R*i*

siano disponibili per salvare e ripristinare il contenuto di IPS, rispettivamente. Suggerire modifiche dell'hardware nelle Figure 5.8 e 5.10 per realizzare queste istruzioni.

Soluzione

Una possibile organizzazione è mostrata in Figura P5.6. Per salvare il contenuto di IPS, la sua uscita è collegata a un ingresso aggiuntivo su MuxY. Quando se ne ripristina il contenuto, MuxIPS seleziona il registro RA.

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

5.1 M Il ritardo di propagazione attraverso il circuito combinatorio in Figura 5.2 è di 600 ps (picosecondi). I registri richiedono un tempo di impostazione (*setup time*, si veda l'Appendice A) di 50 ps, e il ritardo di propagazione massimo dall'ingresso del clock all'uscita Q è di 70 ps.

a. Qual è il periodo di clock minimo richiesto per il corretto funzionamento del circuito?
b. Si assuma che il circuito sia riorganizzato in tre stadi come in Figura 5.3, così che il circuito combinatorio in ciascuno stadio abbia un ritardo di 200 ps. Qual è il periodo di clock minimo in questo caso?

5.2 M Al momento in cui l'istruzione:

Load R6, 1000(R9)

viene prelevata, R6 e R9 contengono i valori 4.200 e 85.320, rispettivamente. La locazione di memoria 86.320 contiene 75.900. Mostrare i contenuti dei registri interstadi in Figura 5.8 durante ognuno dei 5 passi di esecuzione di questa istruzione.

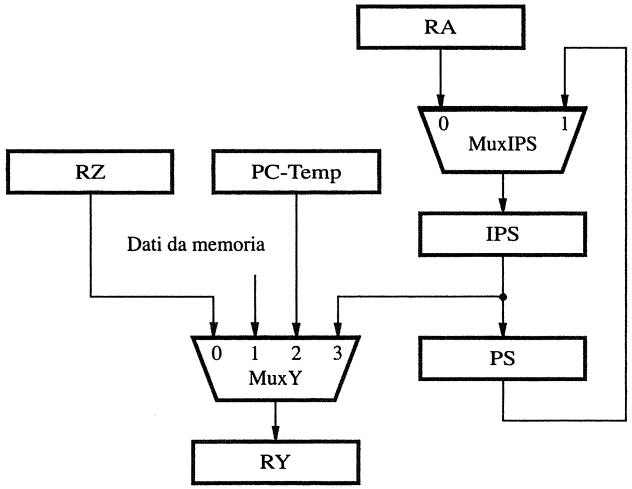


Figura P5.6 Connessione di IPS per il Problema risolto 5.6.

5.3 F La Figura 5.12 mostra i campi di bit assegnati agli indirizzi dei registri per gruppi diversi di istruzioni. Perché è importante usare le stesse posizioni dei campi per tutte le istruzioni?

5.4 M A un certo punto nell'esecuzione di un programma, i registri R4, R6 e R7 contengono i valori 1.000, 7.500 e 2.500, rispettivamente. Mostrare i contenuti dei registri RA, RB, RZ, RY e R6 nella Figura 5.8 durante i passi da 3 a 5 di esecuzione dell'istruzione:

Subtract R6, R4, R7

e anche durante il passo 1 dell'istruzione che viene successivamente prelevata.

5.5 M L'istruzione:

And R4, R4, R8

è immagazzinata nella locazione 0x37C00 in memoria. Nel momento in cui quest'istruzione viene prelevata, i registri R4 e R8 contengono i valori 0x1000 e 0xB2500, rispettivamente. Determinare i valori nei registri PC, R4, RA, RM, RZ e RY delle Figure 5.8 e 5.10 in ciascun ciclo di clock durante l'esecuzione dell'istruzione, e anche nel primo ciclo di clock della successiva istruzione.

5.6 D Modificare le espressioni date nel Problema risolto 5.3 per confrontare due numeri a 4 bit con segno nella rappresentazione in complemento a due.

5.7 F Le istruzioni di chiamata a sottoprogramma descritte nel Capitolo 2 usano sempre lo stesso registro di uso generale, LINK, per immagazzinare l'indirizzo di rientro. Perciò l'indirizzo del registro di rientro non viene incluso nell'istruzione. Tuttavia l'indirizzo LINK viene incluso nei bit IR_{31-27} delle istruzioni di rientro da sottoprogramma (si veda il Paragrafo 5.4.1 e il Problema risolto 5.4). Perché le due istruzioni vengono gestite differentemente?

5.8 M Stilare la sequenza di esecuzione per l'istruzione di rientro da interruzione per un processore che ha la struttura di interruzioni data nel Problema risolto 5.5. Si assuma che l'indirizzo del registro IRA sia codificato nei bit IR_{31-27} dell'istruzione.

5.9 D Si consideri un insieme di istruzioni in cui la codifica delle istruzioni sia tale che gli indirizzi dei registri per istruzioni differenti non siano sempre nelle stesse locazioni di bit. Che effetto avrebbe ciò sui passi di esecuzione delle istruzioni? Cosa si potrebbe fare per mantenere una sequenza di esecuzione a cinque passi in questo caso? Si assuma la stessa struttura hardware di Figura 5.8.

5.10 M Si assuma che gli operandi immediati occupino i bit IR_{21-6} dell'istruzione. Il valore immediato viene esteso con segno a 32 bit nelle istruzioni aritmetiche, come Add, ed esteso con zeri in istruzioni logiche, come OR. Progettare una realizzazione adatta al blocco Immediato in Figura 5.9.

5.11 M Un processore RISC che usa la sequenza di cinque passi nella Figura 5.4 è pilotato da un clock a 1 GHz. Le statistiche per le istruzioni in un programma di grandi dimensioni sono come segue:

Salto	20%
Load	20%
Store	10%
Istruzioni di calcolo	50%

Stimare la frequenza di esecuzione di istruzioni in ciascuno dei seguenti casi.

a. L'accesso alla memoria viene sempre completato in un ciclo di clock.
b. Le operazioni di prelievo vengono completate in un ciclo di clock nel 90% dei casi e in 4 cicli di clock nel restante 10%. In media, l'accesso in memoria agli operandi di un'istruzione Load o Store viene completato in 3 cicli di clock.

5.12 F L'esecuzione di istruzioni di calcolo segue il modello fornito in Figura 5.11 per l'istruzione Add, in cui nessuna azione di elaborazione viene effettuata al passo 4. Si consideri un programma che ha le statistiche per le istruzioni fornite nel Problema proposto 5.11. Stimare l'incremento della frequenza di esecuzione delle istruzioni se questo passo venisse eliminato, assumendo che tutti i passi di esecuzione vengano completati in un ciclo di clock.

5.13 D La Figura 5.16 mostra che il passo 3 di un'istruzione di salto condizionato può fornire come risultato che un nuovo valore sia caricato nel PC. Nei processori con pipeline, è utile determinare il risultato di un salto condizionato il prima possibile nella sequenza di esecuzione. Quali cambiamenti dell'hardware sarebbero necessari per rendere possibile lo spostamento delle azioni nel passo 3 al passo 2? Esaminare tutte le azioni coinvolte in questi due passi e mostrare quali azioni possano essere svolte in parallelo e quali debbano essere completate sequenzialmente.

5.14 M Le istruzioni di un calcolatore sono codificate come mostrato in Figura 5.12. Quando un valore immediato è fornito in un'istruzione, esso deve essere esteso a un valore a 32 bit. Si assuma che il valore immediato sia usato in tre modi diversi:

a. un valore a 16 bit con segno da usare in operazioni aritmetiche;
b. un valore a 16 bit esteso con zeri a sinistra da usare in operazioni logiche;
c. un valore a 26 bit esteso con 2 zeri a destra e con i 4 bit di ordine più alto del PC aggiunti a sinistra da usare in istruzioni di chiamata a sottoprogramma.

Mostrare una realizzazione del blocco Immediato nella Figura 5.19 che possa effettuare le estensioni richieste.

5.15 F Si è visto come tutte le istruzioni in stile RISC possono essere eseguite usando i passi in Figura 5.4 sull'hardware a più stadi di Figura 5.8. I modi di indirizzamento con autoincremento e autodecremento non sono inclusi in un insieme di istruzioni in stile RISC. Spiegare perché l'istruzione:

Load R3, (R5)+

non possa essere eseguita sull'hardware in Figura 5.8.

5.16 F Il Paragrafo 2.9 descrive come le due istruzioni Or e OrHigh possono essere usate per caricare un valore a 32 bit in un registro. Quale funzionalità aggiuntiva è necessaria nel percorso dati del processore per implementare l'istruzione OrHigh? Stilare la sequenza di azioni necessarie a prelevare ed eseguire l'istruzione.

5.17 F Durante il passo 1 dell'elaborazione di istruzioni, viene iniziata un'operazione di lettura dalla memoria, Read, per prelevare un'istruzione alla locazione 0x46000. Tuttavia, poiché l'istruzione non viene trovata nella cache, l'operazione Read ritarda e il segnale MFC non diventa attivo fino al quarto ciclo di clock. Si assuma che il ritardo sia gestito come descritto nel Paragrafo 5.6.2. Mostrare il contenuto del PC durante ciascuno dei quattro cicli di clock del passo 1 e anche durante il passo 2.

5.18 M Stilare la sequenza di passi necessari a prelevare ed eseguire le due istruzioni speciali:

MoveControl R*i*, IPS

e:

MoveControl IPS, R*i*

usate nel Problema risolto 5.6.

5.19 D Quali sono le differenze essenziali fra le strutture dell'hardware nelle Figure 5.8 e 5.22? Illustrare la risposta identificando le difficoltà che si incontrerebbero se si tentasse di eseguire l'istruzione:

Subtract LOC, R5

sull'hardware in Figura 5.8. Questa istruzione effettua l'operazione:

$$LOC \leftarrow [LOC] - [R5]$$

in cui LOC è una locazione di memoria il cui indirizzo viene fornito come seconda parola di un'istruzione di due parole.

5.20 M Si considerino le azioni necessarie ad eseguire le istruzioni fornite nel Paragrafo 5.4.1. Derivare le espressioni logiche per generare i segnali C_selettore, MA_selettore e Y_selettore nelle Figure 5.18 e 5.19 per queste istruzioni.

5.21 F Perché è necessario includere entrambe WMFC e MFC nell'espressione logica per Contatore_abilita fornita nel Paragrafo 5.6.2?

5.22 F Spiegare cosa succederebbe se la variabile MFC venisse omessa dall'espressione per PC_abilita fornita nel Paragrafo 5.6.2.

5.23 M Derivare le espressioni logiche per generare i segnali PC_selettore e INC_selettore mostrati in Figura 5.20, considerando le azioni necessarie quando si eseguono le seguenti istruzioni:

- Branch: tutte le istruzioni di salto, con spiazzamento di salto a 16 bit fornito nell'istruzione;

- Call_Register: un'istruzione di chiamata a sottoprogramma con l'indirizzo del sottoprogramma fornito in un registro di uso generale;
- Altre: tutte le altre istruzioni che non coinvolgono salti.

5.24 M Un processore microprogrammato ha i seguenti parametri. La generazione dell'indirizzo di inizio della microroutine di un'istruzione richiede 2,1 ns e la lettura di una microistruzione dalla memoria di microprogramma richiede 1,5 ns. Lo svolgimento di un'operazione nell'ALU richiede un massimo di 2,2 ns e l'accesso alla memoria cache richiede 1,7 ns. Si assuma che tutte le istruzioni e i dati siano nella cache.

a. Determinare il tempo minimo necessario per ciascuno dei passi nella Figura 5.26.
b. Ignorando tutti gli altri ritardi, qual è il ciclo di clock più breve che può essere usato per questo processore?

5.25 M Stilare la sequenza di passi necessari per prelevare ed eseguire l'istruzione:

Load R3, (R5)+

sul processore di Figura 5.24. Si assuma che gli operandi siano a 32 bit.

5.26 M Si consideri un processore in stile CISC che salva l'indirizzo di rientro da sottoprogramma sulla pila del processore invece che nel registro predefinito LINK. Stilare la sequenza di azioni necessarie per eseguire un'istruzione Call_Register sul processore di Figura 5.24.

6 Introduzione al pipelining

Problemi risolti

Problema 6.1

Si consideri l'esecuzione in pipeline della seguente sequenza di istruzioni:

Add	R4, R3, R2
Or	R7, R6, R5
Subtract	R8, R7, R4

Inizialmente, i registri R2 e R3 contengono 4 e 8, rispettivamente. I registri R5 e R6 contengono 128 e 2, rispettivamente. Si assuma che la pipeline fornisca percorsi di inoltro all'ALU dai registri RY e RZ in Figura 5.8. La prima istruzione viene prelevata nel ciclo 1 e le rimanenti istruzioni vengono prelevate nei cicli successivi.

Disegnare un diagramma simile alla Figura 6.1 per mostrare l'esecuzione in pipeline di queste istruzioni assumendo che il processore usi l'inoltro di operandi. Quindi, con riferimento alla Figura 5.8, descrivere i contenuti dei registri RY e RZ durante i cicli da 4 a 7.

Soluzione

Vi sono dipendenze di dato che coinvolgono i registri R4 e R7. L'istruzione Subtract ha bisogno dei nuovi valori di questi registri prima che essi siano scritti nel banco di registri. Quindi quei valori devono essere inoltrati agli ingressi dell'ALU quando l'istruzione Subtract è nello stadio Elaborazione della pipeline. La Figura P6.1 mostra l'esecuzione con inoltro. Una freccia rappresenta l'inoltro del nuovo valore del registro R7 dal registro RZ, l'altra rappresenta l'inoltro del nuovo valore del registro R4 dal registro RY.

Per i contenuti dei registri RY e RZ durante i cicli da 4 a 7, la risposta è fornita dalla descrizione che segue.

- Usando i valori iniziali per i registri R2 e R3, l'istruzione Add genera il risultato 12 nel ciclo 3. Quel risultato è disponibile nel registro RZ durante il ciclo 4. Il valore nel registro RY durante il ciclo 4 è il risultato dell'istruzione non specificata che precede l'istruzione Add.
- Nel ciclo 4, l'istruzione Or genera il risultato 130. Quel risultato è posto nel registro RZ e sarà disponibile durante il ciclo 5. Il risultato 12 dell'istruzione Add è nel registro RY durante il ciclo 5.
- Nel ciclo 5, l'istruzione Subtract è nello stadio Elaborazione. Per generare un risultato corretto, l'inoltro viene usato per fornire il valore 130 nel registro RY e il valore 12 nel registro RZ. Il risultato dall'ALU è 130 − 12 = 118. Questo risultato è disponibile nel registro RZ durante il ciclo 6. Il risultato dell'istruzione Or, 130, è nel registro RY durante il ciclo 6.
- Nel ciclo 6, l'istruzione Subtract è nello stadio Memoria. L'istruzione non specificata che segue l'istruzione Subtract sta generando un risultato nello stadio Elaborazione. Nel ciclo 7, il risultato dell'operazione non specificata è nel registro RZ, e il risultato dell'istruzione Subtract è nel registro RY.

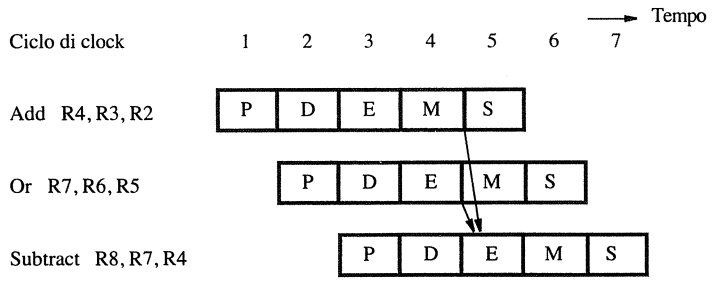


Figura P6.1 Esecuzione in pipeline di istruzioni per il Problema risolto 6.1.

Problema 6.2

Si assuma che il 20% del conteggio dinamico delle istruzioni eseguite per un programma siano istruzioni di salto. Non vi sono stalli della pipeline dovuti a dipendenze di dato. La predizione statica di salto viene usata con l'assunto che il salto non venga effettuato.

a. Determinare i tempi di esecuzione per i due casi: quando il 30% dei salti siano effettuati; quando il 70% dei salti siano effettuati.

b. Determinare l'accelerazione (*speedup*) per una caso relativamente all'altro. Esprimere l'accelerazione come percentuale relativa a 1.

Soluzione

Il Paragrafo 6.8.1 descrive il calcolo di $\delta_{\text{penalità_salto}}$ per considerare l'effetto delle penalità di salto.

a. Il valore di $\delta_{\text{penalità_salto}}$ è 0,20 × 0,30 = 0,06 per il primo caso e 0,20 × 0,70 = 0,14 per il secondo caso. Usando $S = 1 + \delta_{\text{penalità_salto}}$, il tempo di esecuzione per il primo caso è $(1{,}06 \times N)/R$ e $(1{,}14 \times N)/R$ per il secondo caso.

b. Poiché il tempo di esecuzione del primo caso è minore, il miglioramento di prestazioni espresso come percentuale di accelerazione è:

$$\left(\frac{1{,}14}{1{,}06} - 1\right) \times 100 = 7{,}5\%$$

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

6.1 M Considerare le seguenti istruzioni agli indirizzi di memoria indicati:

1000	Add	R3, R2, #20
1004	Subtract	R5, R4, #3
1008	And	R6, R4, #0x3A
1012	Add	R7, R2, R4

Inizialmente i registri R2 e R4 contengono 2.000 e 50, rispettivamente. Queste istruzioni vengono eseguite in un calcolatore che ha una pipeline a cinque stadi come mostrato in Figura

6.2. La prima istruzione viene prelevata nel ciclo di clock 1 e le rimanenti istruzioni sono prelevate nei cicli successivi.

a. Disegnare un diagramma simile alla Figura 6.1 che rappresenti il flusso delle istruzioni attraverso la pipeline. Descrivere le operazioni svolte da ciascuno stadio della pipeline durante i cicli di clock da 1 a 8.

b. Con riferimento alle Figure 5.8 e 5.9, descrivere i contenuti dei registri IR, PC, RA, RB, RY e RZ nella pipeline durante i cicli da 2 a 8.

6.2 M Ripetere il Problema proposto 6.1 per il seguente programma:

```
1000  Add      R3, R2, #20
1004  Subtract R5, R4, #3
1008  And      R6, R3, #0x3A
1012  Add      R7, R2, R4
```

Si assuma che la pipeline fornisca percorsi per l'inoltro all'ALU dai registri RY e RZ in Figura 5.8 e che il processore usi l'inoltro di operandi.

6.3 M Si consideri il ciclo nel programma di Figura 2.8. Si assuma che sia eseguito in una pipeline a cinque stadi con percorsi di inoltro all'ALU dai registri RY e RZ in Figura 5.8. Si assuma che la pipeline usi la predizione statica di salto con l'assunto che il salto non avvenga. Disegnare un diagramma simile alla Figura 6.1 per l'esecuzione di due iterazioni successive del ciclo.

6.4 D Ripetere il Problema proposto 6.3, ma prima riordinare le istruzioni per ottimizzare le prestazioni come farebbe il compilatore.

6.5 D Ripetere il Problema proposto 6.3 per una pipeline che usi il salto differito con un posto del ritardo. Riordinare le istruzioni come necessario per migliorare le prestazioni.

6.6 M Il percorso di inoltro in Figura 6.5 permette che il contenuto del registro RZ sia usato direttamente in un'operazione dell'ALU. Il risultato di tale operazione è immagazzinato nel registro RZ, rimpiazzando il suo contenuto precedente. Questo problema riguarda il tracciamento dei contenuti del registro RZ su più cicli. Si considerino le due istruzioni:

```
I1   Add      R3, R2, R1
I2   LShiftL  R3, R3, #1
```

Mentre l'istruzione I_1 viene prelevata nel ciclo 1, un'istruzione precedentemente prelevata sta effettuando un'operazione con l'ALU che fornisce il risultato 17. Quindi, mentre l'istruzione I_1 viene decodificata nel ciclo 2, un'altra istruzione precedentemente prelevata sta effettuando un'operazione con l'ALU che fornisce il risultato 198. Inoltre durante il ciclo 2, i registri R1, R2 e R3 contengono i valori 30, 100 e 45 rispettivamente. Usando queste informazioni, disegnare un diagramma temporale che mostri il contenuto del registro RZ durante i cicli da 2 a 5.

6.7 M Si assuma che il 20% del conteggio dinamico delle istruzioni eseguite per un programma siano istruzioni di salto. Si usi il salto differito, con un posto del ritardo. Si assuma che non ci siano stalli causati da altri fattori. Dapprima, determinare un'espressione per il tempo di esecuzione espresso in cicli qualora tutti i posti del ritardo siano occupati con istruzioni NOP. Quindi determinare un'altra espressione che rifletta il tempo di esecuzione con il 70% di posti di ritardo occupati con istruzioni utili dal compilatore ottimizzante. Da queste espressioni, determinare il contributo del compilatore all'incremento delle prestazioni, espresso come percentuale di accelerazione.

6.8 D Ripetere il Problema proposto 6.7, ma questa volta per un processore con pipeline con due posti del ritardo per salto. L'output del compilatore ottimizzante è tale da far sì che il primo posto di ritardo venga occupato con un'istruzione utile il 70% delle volte, ma il secondo posto venga occupato con un'istruzione utile solo il 10% delle volte.

Confrontare il tempo di esecuzione derivante dal codice prodotto dal compilatore ottimizzante per questo caso con quello prodotto dal compilatore ottimizzante del Problema proposto 6.7. Si assuma che i due processori abbiano la stessa frequenza di clock. Indicare quale combinazione di processore e compilatore è più veloce, e determinare di quale percentuale di accelerazione è più veloce.

6.9 D Si assuma che il 20% del conteggio dinamico delle istruzioni eseguite per un programma siano istruzioni di salto. Si assuma inoltre che il 75% dei salti siano realmente effettuati. Il programma viene eseguito in due processori diversi che hanno la stessa frequenza di clock. Uno usa la predizione statica dei salti con l'assunto di salto non effettuato. L'altro usa la predizione dinamica di salti basata sugli stati in Figura 6.12a. Il buffer di destinazione di salto viene usato nel modo descritto nel Paragrafo 6.6.4.

a. Senza stalli della pipeline dovuti ad altre cause, quale deve essere la minima accuratezza di predizione per il processore che usa la predizione dinamica di salti per avere almeno le stesse prestazioni del processore che usa la predizione di salto statica?

b. Se l'effettiva accuratezza di predizione dinamica risulta essere del 90%, qual è l'accelerazione relativa rispetto all'uso della predizione statica?

6.10 M Si richiede della logica di controllo aggiuntiva nella pipeline per inoltrare il valore del registro RZ come mostrato in Figura 6.5. Quali specifiche condizioni deve valutare questa logica aggiuntiva per determinare le impostazioni dei multiplatori che alimentano gli ingressi dell'ALU nello stadio Elaborazione della pipeline?

6.11 M Ripetere il Problema proposto 6.10 per le condizioni specifiche relative all'inoltro del contenuto del registro RY in Figura 5.8 ai multiplatori che alimentano gli ingressi dell'ALU.

6.12 D Come continuazione dei Problemi proposti 6.10 e 6.11 considerare la sequenza di istruzioni:

```
Add      R3, R2, R1
Subtract R3, R5, R4
Or       R8, R3, #1
```

Si descriva il modo in cui l'inoltro deve essere gestito per questa situazione. Come dovrebbero essere modificate le condizioni sviluppate nei Problemi 6.10 e 6.11?

6.13 M Si consideri un programma che consiste di quattro istruzioni di accesso alla memoria e quattro istruzioni aritmetiche. Si assuma che non ci siano dipendenze di dato tra le istruzioni. Due versioni di questo programma vengono eseguite sul processore superscalare mostrato in Figura 6.13. La prima versione ha le quattro istruzioni di accesso alla memoria in sequenza, seguite dalle quattro istruzioni aritmetiche. La seconda versione ha le istruzioni di accesso alla memoria alternate con le istruzioni aritmetiche. Disegnare due diagrammi simili alla Figura 6.14 per confrontare l'esecuzione di queste due versioni del programma.

6.14 F Si assuma che un programma non contenga istruzioni di salto. Il programma viene eseguito sul processore superscalare mostrato in Figura 6.13. Qual è il miglior tempo di esecuzione espresso in cicli che si può ottenere se il misto di istruzioni consiste del 75% di istruzioni aritmetiche e del 25% di istruzioni di accesso alla memoria? Qual è il confronto di questo tempo rispetto al miglior tempo di esecuzione sul processore più semplice in Figura 6.2 con la stessa frequenza di clock?

6.15 M Si ripeta il Problema proposto 6.14 per trovare i migliori tempi di esecuzione possibili per i processori nelle Figure 6.2 e 6.13, assumendo che il misto di istruzioni consista per il 15% di istruzioni di salto, con salti mai effettuati, per il 65% di istruzioni aritmetiche, e per il 20% di istruzioni di accesso alla memoria. Si assuma un'accuratezza di predizione del 100% per tutte le istruzioni di salto.

6.16 F Si consideri un processore che usi lo schema di predizione di salto rappresentato in Figura 6.12*b*. L'insieme di istruzioni per il processore viene migliorato con una caratteristica che abilita il compilatore a specificare lo stato iniziale della predizione come PS o PNS per ciascuna istruzione di salto. Questo stato iniziale viene usato dal processore a tempo di esecuzione quando l'informazione sull'istruzione di salto non viene trovata nel buffer di destinazione di salto. Discutere come il compilatore dovrebbe usare questa caratteristica quando genera il codice per i seguenti casi:

a. un ciclo con un'istruzione di salto condizionato alla fine per saltare all'inizio del ciclo;
b. un ciclo con un'istruzione di salto condizionato all'inizio del ciclo per uscire dal ciclo, e un salto incondizionato alla fine del ciclo per saltare all'inizio.

6.17 M Si assuma che un processore abbia la caratteristica descritta nel Problema proposto 6.16 per specificare lo stato iniziale della predizione per istruzioni di salto. Si consideri un'espressione della forma:

IF A>B THEN A = A + 1 ELSE B = B + 1

a. Generare il codice in linguaggio assemblativo per l'espressione qui sopra.
b. In assenza di qualsiasi altra informazione, discutere come il compilatore dovrebbe specificare lo stato iniziale della predizione per istruzioni di salto nel codice assemblativo.
c. Uno studio del comportamento a tempo di esecuzione del programma contenente l'espressione qui sopra rivela che il valore della variabile A è spesso maggiore del valore della variabile B. Se questa informazione fosse resa disponibile al compilatore, discutere come ne sarebbe influenzata la predizione iniziale per le istruzioni di salto.

6.18 M Si consideri un'espressione della forma:

IF A>B THEN A = A + 1 ELSE B = B + 1

a. Si consideri un processore che abbia l'organizzazione in pipeline mostrata in Figura 6.2, con predizione statica di salto e assunto di salto non effettuato. Scrivere il codice in linguaggio assemblativo per l'espressione qui sopra. Disegnare diagrammi simili alla Figura 6.1 per mostrare l'esecuzione in pipeline delle istruzioni per differenti decisioni di salto e determinare i tempi di esecuzione espressi in cicli.
b. Si assuma adesso che venga usato il salto differito. Scrivere il codice in linguaggio assemblativo per l'espressione qui sopra. Disegnare i diagrammi che mostrano l'esecuzione in pipeline delle istruzioni per differenti decisioni di salto e confrontare i tempi di esecuzione espressi in cicli con i tempi per il caso precedente.

7 Sistema di ingresso e uscita

Problemi risolti

Problema 7.1

Il bus di I/O di un calcolatore usa il protocollo sincrono mostrato in Figura 7.4. Il ritardo massimo di propagazione su questo bus è 4 ns. Il master del bus necessita di 1,5 ns per porre un indirizzo sulle linee dell'indirizzo. I dispositivi slave necessitano di 3 ns per decodificare l'indirizzo e un massimo di 5 ns per porre i dati richiesti sulle linee dati. I registri di ingresso collegati al bus hanno un tempo di impostazione minimo di 1 ns. Si assuma che il clock del bus abbia un ciclo di lavoro del 50%; ovvero, le fasi di clock alta e bassa hanno durata uguale. Qual è la massima frequenza di clock di questo bus?

Soluzione

Il tempo minimo per la fase alta del clock è il tempo necessario perché l'indirizzo arrivi e venga decodificato dallo slave, ovvero 1,5 + 4 + 3 = 8,5 ns. Il tempo minimo per la fase bassa del clock è il tempo necessario allo slave per porre i dati sul bus e al master per caricare i dati in un registro, ovvero 5 + 4 + 1 = 10 ns. Quindi, il periodo di clock minimo è $2 \times 10 = 20$ ns, e la massima frequenza di clock è 50 MHz.

Problema 7.2

Un arbitro riceve tre segnali di richiesta, R1, R2, R3, e genera tre segnali di concessione, G1, G2, G3. La richiesta R1 ha la priorità più alta e la richiesta R3 la priorità più bassa. Un esempio di funzionamento di tale arbitro è fornito in Figura 7.9. Tracciare un diagramma degli stati che descriva il comportamento di questo arbitro.

Soluzione

Un diagramma degli stati è mostrato in Figura P7.2. L'arbitro inizia nello stato di inattività, A. Quando uno o più segnali di richiesta vengono asseriti, l'arbitro transisce a uno dei tre stati, B, C o D, secondo quale delle richieste attive ha più alta priorità. Quando entra in un nuovo stato, asserisce il corrispondente segnale di concessione. L'arbitro rimane in quello stato finché il dispositivo servito non rimuove la sua richiesta, momento in cui l'arbitro ritorna allo stato A. Una volta rientrato nello stato A, risponderà a qualsiasi richiesta attiva in quel momento, o aspetterà che una nuova richiesta venga asserita.

Problema 7.3

Progettare un circuito d'interfaccia di uscita per un bus sincrono che usi il protocollo di Figura 7.4. Quando i dati vengono scritti nel registro dati di questa interfaccia, l'interfaccia manda un impulso con un'ampiezza di un ciclo di clock su una linea chiamata Nuovo-dato. Questo impulso segnala al dispositivo di uscita collegato all'interfaccia la disponibilità di un nuovo dato.

Soluzione

Tutti gli eventi in un circuito sincrono sono pilotati da un segnale di clock. Un possibile circuito per l'interfaccia è mostrato in Figura P7.3. Il segnale Scrittura-dati abilita il registro

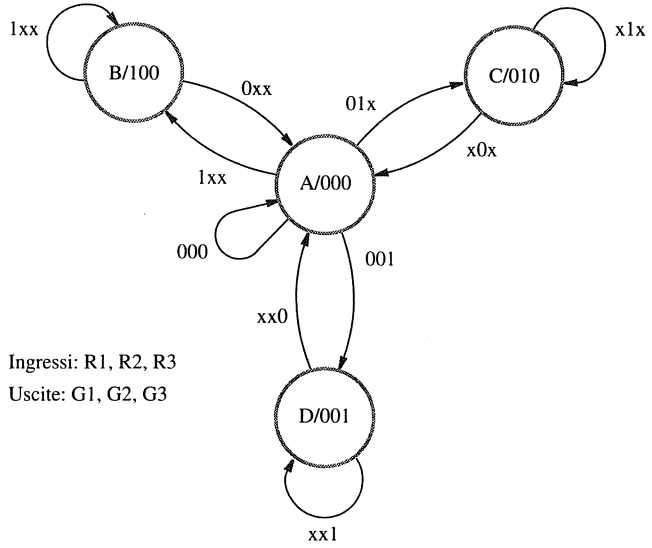


Figura P7.2 Diagramma di stato per il Problema risolto 7.2.

dati, nel quale i dati vengono caricati sul fronte del clock alla fine del ciclo di clock. Al contempo, il flip-flop Nuovo-dato viene posto a 1. La connessione di ritorno dall'uscita $\overline{Q}$ del flip-flop azzera il flip-flop sul successivo fronte del clock.

Problema 7.4

Tracciare un diagramma degli stati per una macchina a stati finiti (FSM) che rappresenti il comportamento del circuito di controllo dell'handshake in Figura 7.14.

Soluzione

Un diagramma degli stati è mostrato in Figura P7.4. Il circuito inizia nello stato A, con il dispositivo dello schermo pronto a ricevere nuovi dati. Quindi, Nuovo-dato = 0 e DOUT = 1. Un'operazione Write fa cambiare a 1 il segnale Scrivi-dato. Questo provoca la transizione allo stato B della macchina a stati, e le sue uscite cambiano a 10. La macchina resta nello stato B finché Pronto non va a 0, indicando che lo schermo ha riconosciuto la disponibilità di nuovi dati. Quando ciò accade, la macchina transisce allo stato C per aspettare che lo schermo divenga nuovamente pronto. Deve anche aspettare che Scrivi-dato ritorni a zero, se non lo ha ancora fatto.

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

7.1 F Il bit di stato d'ingresso in un circuito d'interfaccia, che indica che nuovi dati sono disponibili, viene azzerato non appena il registro di dati di ingresso viene letto. Perché questo è importante?

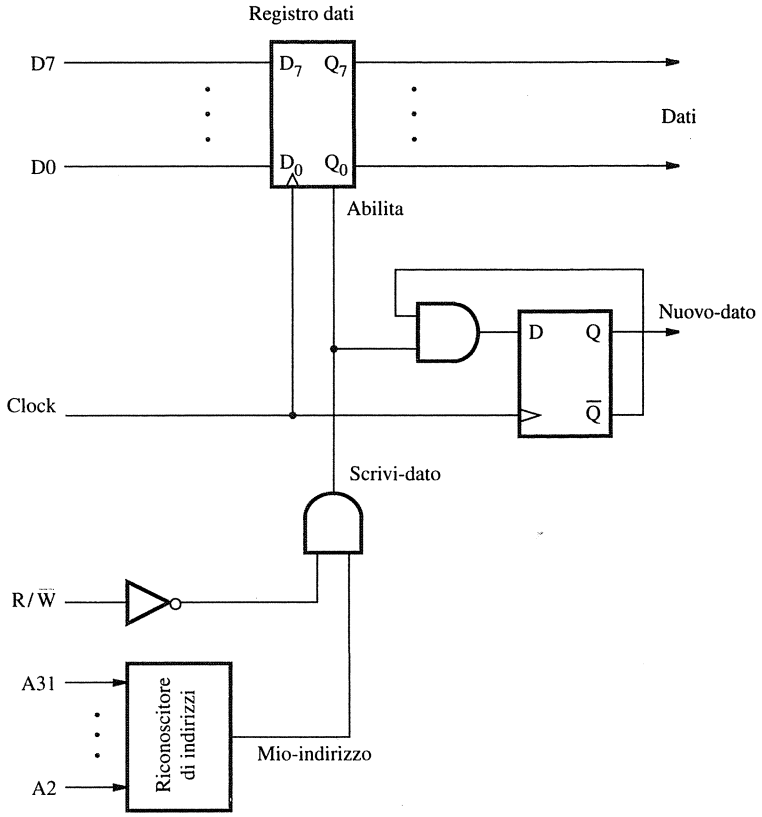


Figura P7.3 Un circuito sincrono d'interfaccia di uscita per il Problema risolto 7.3.

7.2 F Il bus degli indirizzi di un calcolatore ha 16 linee di indirizzo, $A_{15\text{-}0}$. Se l'indirizzo esadecimale assegnato a un dispositivo è 7CA4 e il decodificatore dell'indirizzo per quel dispositivo ignora le linee A_8 e A_9, quali sono tutti gli indirizzi a cui questo dispositivo risponde?

7.3 M Un processore ha sette linee di richiesta di interruzione, da INTR1 a INTR7. La linea INTR7 ha la priorità più alta e INTR1 la priorità più bassa. Progettare un circuito di codifica della priorità che generi un codice a 3 bit che rappresenti la richiesta con la priorità più alta.

7.4 M Le Figure 7.4, 7.5 e 7.6 mostrano tre protocolli per il trasferimento di dati tra un master e uno slave. Cosa accade in ciascun caso se il dispositivo indirizzato non risponde a causa di un malfunzionamento durante un'operazione Read? Quali problemi ciò causerebbe e quali sono i possibili rimedi?

7.5 F Nel diagramma di sincronizzazione in Figura 7.5, il processore mantiene l'indirizzo sul bus finché non riceve una risposta dal dispositivo. Questo è necessario? Quali aggiunte sono necessarie sul dispositivo se il processore invia un indirizzo per un ciclo soltanto?

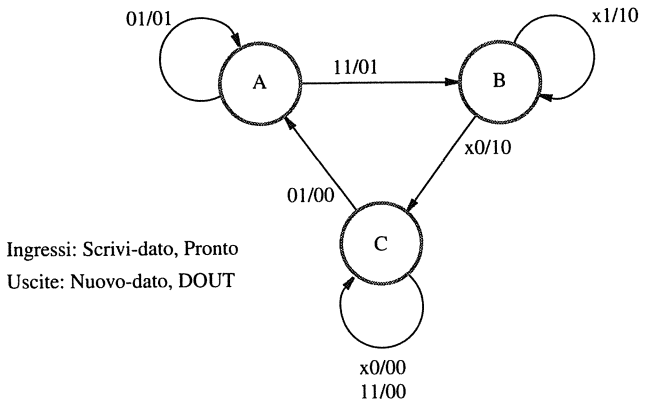


Figura P7.4 Diagramma di stato per il Problema risolto 7.4.

7.6 F Come viene influenzato il diagramma di sincronizzazione in Figura 7.6 quando cresce la distanza tra processore e dispositivo di I/O? Come si tiene conto della maggiore distanza nel caso di Figura 7.4?

7.7 F Si consideri un bus sincrono che operi conformemente al diagramma di sincronizzazione in Figura 7.5. Il bus e il circuito d'interfaccia collegato a esso hanno i seguenti parametri:

Ritardo del pilota del bus	2 ns
Ritardo di propagazione sul bus	da 5 a 10 ns
Ritardo del decodificatore di indirizzi	6 ns
Tempo di prelievo dei dati richiesti	da 0 a 25 ns
Tempo di impostazione	1,5 ns

a. Qual è la massima velocità del clock a cui questo bus può operare?
b. Quanti cicli di clock sono necessari per completare un'operazione di ingresso?

7.8 M Si consideri il protocollo del bus asincrono mostrato in Figura 7.6. Usando gli stessi parametri del Problema proposto 7.7, quali sono i tempi minimo e massimo per completare un trasferimento sul bus? Si conceda 1 ns per lo sfasamento temporale.

7.9 M Il protocollo del bus asincrono in Figura 7.6 usa un handshake pieno, in cui il master mantiene asserito un segnale Master-pronto finché non riceve Slave-pronto, lo slave mantiene Slave-pronto asserito finché Master-pronto non diventa inattivo e così via. Si consideri un protocollo alternativo in cui ciascuno di questi segnali sia un impulso di durata fissa di 4 ns. I dispositivi agiscono solo sul fronte di salita dell'impulso. Usando gli stessi parametri del Problema proposto 7.7, qual è il tempo minimo e massimo per completare un trasferimento sul bus?

7.10 M Nell'esempio di protocollo di arbitraggio illustrato in Figura 7.9, il master che riceve una concessione del bus mantiene la sua linea di richiesta nello stato asserito finché non è pronto a rilasciare l'uso del bus. Si assuma che sia disponibile una linea chiamata Occupato, che viene asserita dal master che sta usando il bus. L'arbitro concede il bus solo quando Occupato è inattiva. Quando un master riceve una concessione, asserisce Occupato e rimuove la sua richiesta, e in risposta l'arbitro rimuove la concessione. Il master disattiva Occupato quando ha finito di usare il bus. Tracciare un diagramma di sincronizzazione equivalente alla Figura 7.9 per questo modo di funzionamento.

7.11 M Modificare il diagramma degli stati fornito nel Problema risolto 7.2 per il modo di funzionamento descritto nel Problema proposto 7.10.

7.12 D L'arbitro del Problema risolto 7.2 controlla l'accesso a una risorsa condivisa. Non permette prelazione (*preemption*): ciò significa che se una richiesta ad alta priorità viene ricevuta dopo la concessione a una richiesta a bassa priorità, essa deve aspettare che il servizio del dispositivo che sta correntemente usando la risorsa condivisa venga completato. In alcuni casi è desiderabile permettere la prelazione per servire più rapidamente un dispositivo ad alta priorità. I dispositivi in questo sistema devono essere capaci di fermarsi e rilasciare l'uso della risorsa condivisa quando l'arbitro chiede loro di farlo. Questo deve essere fatto in modo sicuro. A un dispositivo che sta usando la risorsa deve essere permesso di raggiungere un punto sicuro in cui il servizio possa essere terminato. Questo dispositivo segnalerebbe quindi all'arbitro la cessazione del suo uso della risorsa.

a. Suggerire un'opportuna modifica al protocollo di segnalazione che permetta al servizio in esecuzione di terminare in modo sicuro.
b. Modificare il diagramma degli stati dell'arbitro per realizzare la revisione del protocollo come sopra.

7.13 F Un arbitro controlla l'accesso a una risorsa condivisa. Esso usa uno schema di priorità a rotazione (*round-robin*) per rispondere alle richieste sulle linee da R1 a R4. Inizialmente, R1 ha la priorità più alta e R4 la priorità più bassa. Dopo che una richiesta su una delle linee riceve il servizio, quella linea va al livello di priorità più basso e la prossima linea della sequenza diventa la linea a più alta priorità. Per esempio, dopo che R2 sia stata servita, l'ordine di priorità, partendo con il più alto, diventa R3, R4, R1, R2. Quale sarà la sequenza di concessioni per la seguente sequenza di richieste: R3, R1, R4, R2? Si assuma che le ultime tre richieste arrivino mentre la prima richiesta viene servita.

7.14 F Si consideri un arbitro che usa lo schema di priorità descritto nel Problema proposto 7.13. Determinare cosa accade se un dispositivo richiede ripetutamente il servizio. Confrontare il comportamento di questo arbitro con l'uso dello schema di priorità fisse.

7.15 F Formulare l'espressione logica per un decodificatore di indirizzi che riconosca l'indirizzo esadecimale a 16 bit FA68.

7.16 M Un impianto industriale usa vari sensori per monitorare temperatura, pressione e altri fattori. Ciascun sensore include un commutatore che passa alla posizione ON quando il parametro corrispondente eccede un limite prefissato. Bisogna collegare otto di tali sensori al bus di un calcolatore a 16 bit. Progettare un'interfaccia appropriata per abilitare la lettura simultanea dello stato di tutti gli otto commutatori come singolo byte. Si assuma che il bus sia sincrono e che usi la sequenza di sincronizzazione di Figura 7.4.

7.17 **F** Il protocollo del bus di Figura 7.4 specifica che il dispositivo slave dovrebbe inviare i suoi dati solo nella seconda fase del clock.

a. È possibile che un dispositivo possa riconoscere il suo indirizzo e sia pronto a mandare dati prima. Perché non è lecito farlo? Il processore riceverebbe dati erronei?
b. Insorgerebbe qualche altro problema?

7.18 **M** I dati sono posti in una piccola memoria in un'interfaccia di ingresso collegata a un bus sincrono che usa il protocollo di Figura 7.5. Le operazioni Read e Write sul bus sono indicate da una linea di Comando detta R/$\overline{W}$. La velocità della memoria è tale che sono richiesti due cicli di clock per leggere i dati dalla memoria. Progettare un circuito che generi la risposta Slave-pronto di questa interfaccia.

7.19 **F** Ciascuno dei due segnali DEVSEL# e TRDY# del protocollo PCI in Figura 7.19 rappresenta una risposta dall'iniziatore. In che cosa differiscono le funzioni di questi due segnali?

7.20 **F** Si consideri il trasferimento di dati mostrato in Figura 7.19 per il bus PCI. Come gestirebbe questo protocollo del bus una situazione in cui all'obiettivo occorresse un ritardo di due cicli di clock fra le parole 2 e 3?

7.21 **F** Tracciare un diagramma di sincronizzazione per trasferire tre parole a un dispositivo di uscita collegato al bus PCI.

8 Sistema di memoria

Problemi risolti

Problema 8.1

Descrivere una struttura simile a quella in Figura 8.10 per una memoria 8M × 32 usando chip di memoria 512K × 8.

Soluzione

La struttura richiesta è essenzialmente la stessa di quella in Figura 8.10, eccetto che sono necessarie 16 righe, ognuna con quattro chip 512 × 8. Le linee di indirizzo A_{18-0} dovrebbero essere connesse a tutti i chip. Le linee di indirizzo A_{22-19} dovrebbero essere connesse a un decodificatore a 4 bit per selezionare una delle 16 righe.

Problema 8.2

Un calcolatore usa indirizzi di memoria a 32 bit e ha una memoria principale di 1 Gbyte. Esso ha una cache di 4 Kbyte organizzata secondo lo schema associativo a gruppi, con 4 blocchi per gruppo e 64 byte per blocco.

a. Calcolare il numero di bit in ciascuno dei campi Etichetta, Gruppo e Spiazzamento dell'indirizzo di memoria.
b. Assumere che la cache sia inizialmente vuota. Supporre che il processore prelevi 1088 parole di quattro byte ciascuna da locazioni di memoria successive a partire dalla locazione 0. Questa sequenza di prelievi viene reiterata per altre nove volte. Se la cache è 10 volte più veloce della memoria, stimare il guadagno di prestazione risultante dall'uso della cache. Si assuma l'uso dell'algoritmo LRU per rimpiazzare i blocchi.

Soluzione

Indirizzi consecutivi si riferiscono a byte.

a. Un blocco ha 64 byte, quindi il campo Spiazzamento è di 6 bit. Con 4 × 64 = 256 byte in un gruppo, ci sono 4 K/256 = 16 gruppi, il che richiede un campo Gruppo di 4 bit. Questo lascia 32 − 4 − 6 = 22 bit per il campo Etichetta.
b. Le 1088 parole costituiscono 68 blocchi, che occupano i blocchi da 0 a 67 nella memoria. La cache ha spazio per 64 blocchi. Quindi dopo che i blocchi 0, 1, 2, ..., 63 sono stati letti dalla memoria e posti nella cache al primo passo, la cache è piena. Ai successivi quattro blocchi, numerati da 64 a 67, corrispondono i gruppi 0, 1, 2, e 3. Ciascuno di questi sostituirà il blocco nella cache meno recentemente usato nel suo gruppo, che è il blocco nella posizione 0 del gruppo. Durante il secondo passo, il blocco di memoria 0 deve essere ricaricato nel gruppo 0 della cache, poiché è stato sovrascritto dal blocco 64. Rimpiazzerà il blocco meno recentemente usato del gruppo 0 in quel momento, che è il blocco nella posizione 1 del gruppo. Quindi, i blocchi di memoria 1, 2, e 3 rimpiazzeranno il blocco in posizione 1 dei gruppi 1, 2 e 3 nella cache, rispettivamente. I blocchi di memoria da 4 a 15 saranno trovati nella cache. I blocchi di memoria da 16 a 19, che erano

nelle posizioni 1 dei gruppi da 0 a 3, adesso saranno stati sovrascritti, e saranno ricaricati nella posizione 2 di questi gruppi.

A mano a mano che l'esecuzione va avanti, tutti i blocchi di memoria che occupano i primi quattro dei 16 gruppi di cache saranno sempre sovrascritti prima che possano essere usati in un passo successivo. I blocchi di memoria 0, 16, 32, 48, e 64 si rimpiazzano continuamente a vicenda poiché competono per le 4 posizioni di blocco nel gruppo 0 della cache. La stessa cosa accade nel gruppo 1 della cache (blocchi di memoria 1, 17, 33, 49, 65), nel gruppo 2 della cache (blocchi di memoria 2, 18, 34, 50, 66), e gruppo 3 della cache (blocchi di memoria 3, 19, 35, 51, 67). I blocchi di memoria che occupano gli ultimi 12 gruppi (i gruppi da 4 a 15) vengono prelevati una volta nel primo passo e rimangono nella cache per i prossimi 9 passi.

In sintesi, al primo passo, tutti i 68 blocchi del ciclo sono prelevati dalla memoria. In ciascuno dei successivi 9 passi, 48 blocchi vengono trovati nei gruppi da 4 a 15 della cache, e i rimanenti 20 blocchi devono essere prelevati dalla memoria. Sia τ il tempo di accesso alla cache. Quindi:

$$\begin{aligned}\text{Guadagno di prestazione} &= \frac{\text{Tempo senza cache}}{\text{Tempo con cache}} = \\ &= \frac{10\times 68\times 10\tau}{1\times 68\times 11\tau + 9(20\times 11\tau + 48\tau)} = 2{,}15\end{aligned}$$

Questo esempio illustra una debolezza dell'algoritmo LRU durante l'esecuzione di cicli di programma. Si veda il Problema proposto 8.9 per le prestazioni di un algoritmo alternativo in questo caso.

Problema 8.3

Si supponga che un calcolatore abbia un processore con due cache L1, una per le istruzioni e una per i dati, e una cache L2. Sia τ il tempo di accesso per le due cache L1. Le penalità di miss sono approssimativamente 15τ per trasferire un blocco da L2 a L1 e 100τ per trasferire un blocco dalla memoria principale a L2. Per l'obiettivo di questo problema, si assuma che i tassi di hit siano gli stessi per istruzioni e dati e che i tassi di hit nelle cache L1 e L2 siano 0,96 e 0,80, rispettivamente.

a. Quale frazione di accessi risulta in miss in entrambe le cache L1 e L2 e quindi richiede un accesso alla memoria principale?
b. Qual è il tempo medio di accesso visto dal processore?
c. Supponendo che la cache L2 abbia un tasso di hit ideale, cioè pari a 1, di quale fattore si ridurrebbe il tempo medio di accesso alla memoria visto dal processore?
d. Si consideri il seguente cambiamento alla gerarchia di memoria: la cache L2 è rimossa e la dimensione delle cache L1 è accresciuta così da dimezzarne il tasso di miss. Qual è il tempo medio di accesso alla memoria visto dal processore in tal caso?

Soluzione

Il tempo medio di accesso alla memoria con un livello di cache è dato nel Paragrafo 8.7.1 dall'equazione (8.1):

$$\Delta T_{accesso} = hH + (1 - h)M$$

Con cache L1 e L2, il tempo medio di accesso alla memoria è dato nel Paragrafo 8.7.2 dall'equazione (8.2):

$$\Delta T_{accesso} = h_1H_1 + (1 - h_1)[h_2H_2 + (1 - h_2)M]$$

a. La frazione di accessi alla memoria che risultano in miss in entrambe le cache L1 e L2 è:

$$(1 - h_1)(1 - h_2) = (1 - 0{,}96)(1 - 0{,}80) = 0{,}008$$

b. Il tempo medio di accesso alla memoria usando due livelli di cache è:

$$\Delta T_{accesso} = 0{,}96\tau + 0{,}04(0{,}80 \times 15\tau + 0{,}20 \times 100\tau) = 2{,}24\tau$$

c. Senza miss nella cache L2, abbiamo:

$$\Delta T_{accesso}(\text{ideale}) = 0{,}96\tau + 0{,}04 \times 15\tau = 1{,}56\tau$$

Quindi:

$$\frac{\Delta T_{accesso}(\text{reale})}{\Delta T_{accesso}(\text{ideale})} = \frac{2{,}24\tau}{1{,}56\tau} = 1{,}44$$

d. Con cache L1 più grandi e senza la cache L2, il tempo di accesso è:

$$\Delta T_{accesso} = 0{,}98\tau + 0{,}02 \times 100\tau = 2{,}98\tau$$

Problema 8.4

Una matrice 1.024 × 1.024 di numeri a 32 bit deve essere normalizzata come segue. Per ciascuna colonna viene trovato l'elemento più grande e tutti gli elementi della colonna vengono divisi per il suo valore. Si assuma che ogni pagina nella memoria virtuale consista di 4 Kbyte, e che 1 Mbyte della memoria principale siano allocati per immagazzinare dati della matrice durante questo calcolo. Si assuma che siano necessari 10 ms per caricare una pagina dal disco alla memoria principale quando si verifica un fault di pagina.

a. Si assuma che la matrice sia elaborata una colonna alla volta. Quanti fault di pagina si verificano e quanto tempo occorre per completare il processo di normalizzazione se gli elementi della matrice sono immagazzinati in ordine di colonna nella memoria virtuale?
b. Ripetere la parte **a.** assumendo che gli elementi siano immagazzinati in ordine di riga.
c. Proporre un modo alternativo di elaborare la matrice per ridurre il numero di fault di pagina quando la matrice è immagazzinata in memoria in ordine di riga. Stimare il numero di fault di pagina e il tempo richiesto dalla soluzione proposta.

Soluzione

Ciascun numero a 32 bit occupa 4 byte. Quindi ogni pagina contiene 1.024 numeri. C'è spazio per 256 pagine nella parte di 1 Mbyte della memoria principale che viene allocata per immagazzinare dati durante il calcolo.

a. Ciascuna colonna è immagazzinata in una pagina; si ha un fault di pagina per portare ciascuna colonna nella memoria principale, per un totale di 1.024 fault di pagina.
Tempo di calcolo = 1.024 × 10 ms = 10,24 s.
b. Elaborare ciascuna colonna richiede due passi, il primo per trovare l'elemento più grande e il secondo per effettuare la normalizzazione. Quando si elabora la prima colonna, ciascun accesso a un elemento provoca un fault di pagina che porta tutti gli elementi della

riga corrispondente nella memoria principale. Dopo che 256 elementi sono stati esaminati, la memoria principale è piena. L'accesso ai successivi 256 elementi provoca fault di pagina che rimpiazzano tutti i dati in memoria, e il processo si ripete. Quindi un fault di pagina si verifica per ciascun accesso a ciascun elemento nella matrice.
Tempo di calcolo = 2 × 1.024 × 1.024 × 10 ms = 20.972 s = 5,8 ore.

c. Un'alternativa più efficiente per questa disposizione di dati è completare il primo passo solo per un quarto di ciascuna colonna per tutte le colonne, quindi elaborare il secondo quarto e così via. Il secondo passo è gestito allo stesso modo. In questo caso, ciascun passo attraverso la matrice provoca 1.024 fault di pagina, per un totale di 2.048.
Tempo di calcolo = 2.048 × 10 ms = 20,48 s.

Questo esempio illustra come può crescere in modo notevole il numero di fault di pagina in alcuni casi in cui la dimensione della memoria è insufficiente per l'applicazione. Questo comportamento è chiamato *thrashing*.

Problema 8.5

Si consideri una lunga sequenza di accessi al disco con un tempo medio di posizionamento di 6 ms e una latenza rotazionale di 3 ms. La dimensione media dei blocchi a cui si accede è 8 Kbyte. Il tasso di trasferimento di dati dal disco è 34 Mbyte/s.

a. Assumendo che i blocchi di dati siano collocati in modo casuale sul disco, stimare la percentuale media del tempo totale dovuta a operazioni di posizionamento e ritardi di rotazione.

b. Ripetere la parte **a.** per la situazione in cui gli accessi al disco siano organizzati in modo tale che nel 90% dei casi il prossimo accesso sarà a un blocco di dati sullo stesso cilindro.

Soluzione

Per trasferire un blocco di dati occorrono 8K/34Ms = 0,23 ms.

a. Il tempo totale necessario per accedere a ciascun blocco è 6 + 3 + 0,23 = 9,23 ms. La frazione di tempo impiegata per posizionamento e latenza è 9/9,23 = 0,97 = 97%.

b. Nel 90% dei casi si ha solo il ritardo di rotazione. Quindi il tempo medio di accesso a un blocco è 0,9 × 3 + 0,1 × 9 + 0,23 = 3,83 ms. La frazione di tempo impiegata per posizionamento e latenza è 3,6/3,83 = 0,94 = 94%.

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

8.1 M Si consideri la cella di memoria dinamica di Figura 8.6. Si assuma che $C = 30$ femtofarad (10^{-15} F) e che la corrente che si perde attraverso il transistore è circa 0,25 picoampere (10^{-12} A). La tensione sul capacitore a piena carica è 1,5 V. La cella deve essere rinfrescata prima che questa tensione scenda sotto 0,9 V. Stimare la frequenza di rinfresco minima.

8.2 M Si consideri una memoria principale costruita con chip SDRAM. I dati sono trasferiti a raffica come mostrato in Figura 8.9, eccetto che la lunghezza della raffica è 8. Si assuma che 32 bit di dati siano trasferiti in parallelo. Se si usa un clock a 400 MHz, stimare quanto tempo è necessario per trasferire:

a. 32 byte di dati.
b. 64 byte di dati.

Qual è la latenza in ciascun caso?

8.3 F Descrivere una struttura simile a quella in Figura 8.10 per una memoria 16M × 32 usando chip di memoria 1M × 4.

8.4 F Esprimere una valutazione critica della seguente affermazione: "L'uso di un processore più veloce fornisce un corrispondente incremento delle prestazioni di un calcolatore anche se la velocità della memoria principale rimane la stessa."

8.5 M La memoria di un calcolatore è indirizzabile a byte, e la lunghezza della parola è 32 bit. Un programma consiste di due cicli innestati: un ciclo interno piccolo e un ciclo esterno più grande. La struttura generale del programma è data in Figura P8.5. Gli indirizzi decimali di memoria mostrati delimitano la collocazione dei due cicli e l'inizio e la fine del programma. Tutte le locazioni di memoria nelle varie sezioni del programma, 8-52, 56-136, 14-240 e così via, contengono istruzioni da eseguire in sequenza lineare. Il programma deve essere eseguito su un calcolatore che ha una cache di istruzioni a corrispondenza diretta (si veda la Figura 8.16) con i seguenti parametri:

Dimensione della cache 1K byte

Dimensione di blocco 128 byte

La penalità di miss nella cache delle istruzioni è 80τ, dove τ è il tempo di accesso alla cache. Calcolare il tempo totale necessario per prelevare le istruzioni durante l'esecuzione del programma in Figura P8.5.

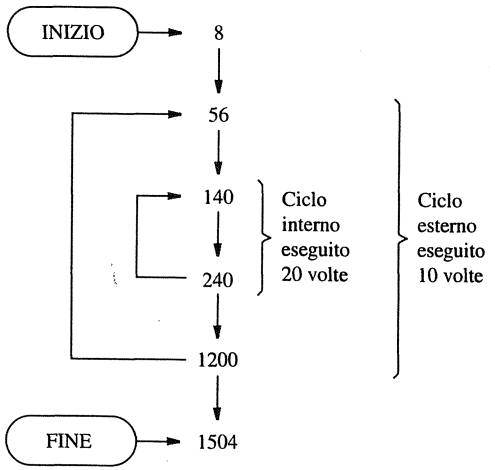
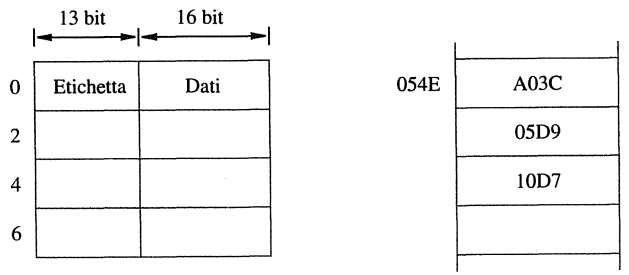


Figura P8.5 Struttura di un programma per il Problema proposto 8.5.



(a) Cache (b) Memoria principale

Figura P8.6 Contenuti della cache e della memoria principale nel Problema proposto 8.6.

8.6 M Un calcolatore con lunghezza di parola di 16 bit ha una cache a corrispondenza diretta, usata sia per istruzioni sia per dati. Gli indirizzi di memoria sono lunghi 16 bit, e la memoria è indirizzabile a byte. La cache è piccola a scopo illustrativo, contiene solo quattro parole da 16 bit. Ogni parola costituisce un blocco della cache e ha associata un'etichetta da 13 bit, come mostrato in Figura P8.6a. Si accede alle parole nella cache usando i 3 bit di più basso ordine dell'indirizzo. Quando si ha miss durante un'operazione Read, sia per un'istruzione sia per un operando dati, la parola richiesta viene letta dalla memoria principale e inviata al processore. Al contempo, viene copiata nella cache e il suo numero di blocco viene immagazzinato nell'etichetta associata. Si consideri il seguente breve ciclo, in cui ogni istruzione è lunga 16 bit:

```
CICLO:  Add        R0, (R1)+
        Decrement  R2
        BNE        CICLO
```

Si assuma che, prima di entrare in questo ciclo, i registri R0, R1 e R2 contengano rispettivamente 0, 0×054E e 3. Si assuma, inoltre, che la memoria principale contenga i dati mostrati in Figura P8.6b, dove tutti i valori sono in notazione esadecimale. Il ciclo inizia alla locazione CICLO = 02EC. Si usa il modo di indirizzamento con autoincremento nell'istruzione Add per accedere ai numeri successivi nella lista da 3 numeri e sommarli nel registro R0. Il registro contatore, R2, è decrementato fino a raggiungere 0, al qual punto si esce dal ciclo.

a. Partendo con una cache vuota, mostrare il contenuto della cache, includendo le etichette, alla fine di ciascuna passata del ciclo.
b. Si assuma che i tempi di accesso alla cache e alla memoria siano τ e 10τ, rispettivamente. Calcolare il tempo di esecuzione per ciascuna passata, tenendo conto solo dei tempi di accesso alla memoria.

8.7 M Ripetere il Problema proposto 8.6 assumendo che solo le istruzioni siano memorizzate nella cache. Gli operandi dati sono prelevati direttamente dalla memoria principale e non copiati nella cache. Perché questa scelta porta a un'esecuzione più veloce rispetto a quando sia istruzioni sia dati sono caricati nella cache?

8.8 F Una cache con indirizzamento associativo a gruppi consiste di un totale di 64 blocchi, divisi in gruppi di 4. La memoria principale contiene 4096 blocchi, ciascuno dei quali consta di 32 parole. Assumendo uno spazio degli indirizzi a 32 bit indirizzabile a byte, quanti bit ci sono in ciascuno dei campi Etichetta, Gruppo e Spiazzamento?

8.9 M Si consideri la cache nel Problema risolto 8.3. Si assuma che quando un blocco debba essere trasferito dalla memoria principale e il gruppo corrispondente nella cache sia pieno, il nuovo blocco rimpiazzi il blocco usato più di recente in questo gruppo. Derivare la soluzione per la parte **b.** in questo caso.

8.10 D Il Paragrafo 8.6.3 illustra l'effetto di schemi di indirizzamento di cache differenti, usando il programma in Figura 8.20. Si supponga che questo programma venga modificato così che nel secondo ciclo gli elementi siano trattati nello stesso ordine del primo ciclo; ovvero il controllo per il secondo ciclo sia specificato da:

for q := 0 **to** 9 **do**

Derivare le equivalenti delle Figure da 8.21 a 8.23 per questo programma. Quali conclusioni possono essere tratte da questo esercizio?

8.11 M Un calcolatore indirizzabile a byte ha una piccola cache di dati capace di tenere otto parole da 32 bit. Ciascun blocco della cache consiste di una parola da 32 bit. Quando un certo programma viene eseguito, il processore legge i dati sequenzialmente dai seguenti indirizzi esadecimali:

200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4

Questa sequenza di letture è ripetuta in quattro iterazioni di un ciclo.

a. Si assuma che la cache sia inizialmente vuota. Mostrare il contenuto della cache alla fine di ciascuna passata del ciclo se si usa una cache a corrispondenza diretta e calcolare il tasso di hit.
b. Ripetere la parte **a.** per una cache a corrispondenza associativa che usa l'algoritmo di sostituzione LRU.
c. Ripetere la parte **a.** per una cache associativa a gruppi con quattro vie.

8.12 M Ripetere il Problema proposto 8.11 assumendo che ciascun blocco della cache consista di due parole da 32 bit. Per la parte **c.** usare una cache associativa a gruppi con due vie che usi l'algoritmo di sostituzione LRU.

8.13 F La dimensione del blocco di cache in molti calcolatori va da 32 a 128 byte. Quali sarebbero i principali vantaggi e svantaggi nello scegliere una dimensione dei blocchi maggiore o minore, lasciando inalterata la dimensione della cache?

8.14 M Un calcolatore ha due livelli di cache L1 e L2. Tracciare due grafici per il tempo medio di accesso alla memoria (asse y) in funzione del tasso di hit h_1 (asse x) per i due valori $h_2 = 0{,}75$ e $h_2 = 0{,}85$. Si usino i valori 0,90, 0,92, 0,94 e 0,96 per h_1. Si assuma che le penalità di miss siano rispettivamente 15τ e 100τ per le cache L1 e L2, dove τ è il tempo di accesso alle cache L1.

8.15 F Si consideri la cache a due livelli descritta nel Problema risolto 8.4. Il tempo medio di accesso è dato nella soluzione della parte **b.** dell'esempio, pari a $2{,}24\tau$. Quale valore per h_1 sarebbe necessario per ridurre $\Delta T_{accesso}$ a $1{,}5\tau$, se tutti gli altri parametri rimangono gli stessi dell'esempio? Si può ottenere lo stesso risultato migliorando il tasso di hit di L2?

8.16 F Si consideri la seguente analogia per il concetto di cache. Un tecnico va in una casa per riparare il sistema di riscaldamento, portando con sé una cassetta che contiene un certo numero di utensili che ha usato recentemente in lavori simili. Egli usa questi utensili ripetutamente, finché non raggiunge un punto in cui sono necessari altri utensili. È probabile che abbia gli utensili richiesti nel suo automezzo parcheggiato davanti alla casa. Tuttavia, se gli utensili necessari non sono nell'automezzo, deve andare a prenderli nella sua officina. Si supponga che la cassetta, l'automezzo e l'officina corrispondano, rispettivamente, alla cache L1, alla cache L2 e alla memoria principale di un calcolatore. Quanto è accurata questa analogia? Discuterne le caratteristiche corrette e quelle non corrette.

8.17 F Lo scopo dell'uso di una cache L2 è di ridurre la penalità di miss della cache L1 e quindi di ridurre il tempo di accesso alla memoria visto dal processore. Un'alternativa è quella di incrementare la dimensione della cache L1 per incrementare il suo tasso di hit. Cosa limita l'utilità di questo approccio?

8.18 M Elaborare una critica dell'assunzione fatta nell'Esempio 8.1, nel Paragrafo 8.7.1, che la penalità di miss sia la stessa per gli accessi in lettura e in scrittura. Si considerino entrambi i casi di modo di scrittura immediata e di scrittura differita, come descritti nel Paragrafo 8.6, nel formulare la risposta.

8.19 M Si consideri un calcolatore nel quale le pagine disponibili nella memoria fisica sono divise fra diversi programmi delle applicazioni. Il sistema operativo monitora l'attività di trasferimento di pagine e aggiusta dinamicamente il numero di pagine allocate ai vari programmi. Suggerire una strategia appropriata che il sistema operativo possa usare per minimizzare il tasso complessivo di trasferimenti di pagine.

8.20 M In un calcolatore con memoria virtuale, l'esecuzione di un'istruzione può essere interrotta da un fault di pagina. Quali informazioni di stato sono da salvare cosicché l'istruzione possa poi essere ripresa? Si noti che trasferire una nuova pagina nella memoria principale comporta un trasferimento in DMA, che richiede l'esecuzione di altre istruzioni. È più semplice abbandonare l'istruzione interrotta e poi rieseguirla daccapo? È fattibile?

8.21 F Quando un programma genera un riferimento a una pagina che non risiede nella memoria principale fisica, l'esecuzione del programma viene sospesa finché la pagina richiesta non viene caricata nella memoria principale dal disco. Quali difficoltà potrebbero insorgere quando un'istruzione in una pagina ha un operando in una pagina differente? Quali funzionalità deve avere il processore per gestire questa situazione?

8.22 M Un'unità a disco ha 24 superfici di registrazione, un totale di 14.000 cilindri e una media di 400 settori per traccia. Ciascun settore contiene 512 byte di dati.

a. Qual è il massimo numero di byte immagazzinabili in questa unità?
b. Qual è il tasso di trasferimento dati in byte per secondo alla velocità di rotazione di 7.200 rpm?
c. Usando una parola da 32 bit, suggerire uno schema appropriato per specificare l'indirizzo su disco.

8.23 M Si consideri una lunga sequenza di accessi a un disco con 8 ms di tempo medio di posizionamento, 3 ms di latenza rotazionale e un tasso di trasferimento di dati di 60 Mbyte/s. La dimensione media dei blocchi a cui si accede è 64 Kbyte. Si assuma che ciascun blocco di dati sia immagazzinato in settori contigui.

a. Assumendo che i blocchi siano collocati in modo casuale sul disco, stimare la percentuale media del tempo totale occupato da operazioni di posizionamento e ritardi di rotazione.

b. Si supponga che 20 blocchi siano trasferiti in sequenza da cilindri adiacenti, riducendo il tempo di posizionamento a 1 ms, e che i blocchi siano collocati in modo casuale su questi cilindri. Qual è il tempo di trasferimento totale?

8.24 M Il tempo medio di posizionamento e la latenza di rotazione in un sistema a disco sono 6 ms e 3 ms, rispettivamente. Il tasso di trasferimento dati da o verso il disco è 30 Mbyte/s, e tutti gli accessi al disco sono per 8 Kbyte di dati, immagazzinati in settori contigui. I blocchi di dati sono immagazzinati in locazioni casuali sul disco. Il controllore del disco ha un buffer di 8 Kbyte. Il controllore del disco, il processore e la memoria principale sono tutti collegati a un singolo bus. La larghezza del bus dati è di 32 bit e un singolo trasferimento sul bus da o verso la memoria principale richiede 10 nanosecondi.

a. Qual è il massimo numero di unità disco che possono simultaneamente trasferire dati da e verso la memoria principale?
b. Quale percentuale di accessi alla memoria principale sono usati da un'unità disco, in media, su un lungo periodo di tempo durante cui viene effettuata una sequenza di trasferimenti indipendenti di 8 Kbyte?

8.25 M I dischi magnetici sono usati come memoria secondaria per programmi e file di dati nella maggior parte dei sistemi con memoria virtuale. Quali parametri del disco dovrebbero influenzare la scelta della dimensione di pagina?

9 Aritmetica

Problemi risolti

Problema 9.1

Quante porte logiche sono necessarie per costruire l'addizionatore a 4 bit con anticipo del riporto mostrato in Figura 9.4?

Soluzione

Ciascuna cella B richiede 3 porte come mostrato in Figura 9.4a. Quindi, sono necessarie 12 porte per tutte le quattro celle B. I riporti c_1, c_2, c_3 e c_4 forniti dalla logica di anticipo del riporto richiedono rispettivamente 2, 3, 4 e 5 porte, concordemente alle quattro espressioni logiche nel Paragrafo 9.2.1. La logica di anticipo del riporto produce anche G_0^I, usando 4 porte, e P_0^I, usando 1 porta, come mostrato anche nel Paragrafo 9.2.1. Quindi sono necessarie in totale 19 porte per realizzare la logica di anticipo del riporto.

L'addizionatore a 4 bit completo richiede 12 + 19 = 31 porte, con un fan-in massimo di 5.

Problema 9.2

Assumendo una rappresentazione di numeri in complemento a 2 da 6 bit, moltiplicare il moltiplicando A = 110101 per il moltiplicatore B = 011011 usando entrambi gli algoritmi di Booth, normale e con ricodifica bit-pair, seguendo lo schema usato in Figura 9.15.

Soluzione

Le moltiplicazioni sono svolte come segue:

a. Algoritmo di Booth normale

						1	1	0	1	0	1
					×	+1	0	−1	+1	0	−1
0	0	0	0	0	0	0	0	1	0	1	1
										0	
1	1	1	1	1	1	0	1	0	1		
0	0	0	0	0	1	0	1	1			
							0				
1	1	1	0	1	0	1					
1	1	1	0	1	1	0	1	0	1	1	1

b. Algoritmo di Booth con ricodifica bit-pair

						1	1	0	1	0	1
					×		+2		−1		−1
0	0	0	0	0	0	0	0	1	0	1	1
0	0	0	0	0	0	1	0	1	1		
1	1	1	0	1	0	1					
1	1	1	0	1	1	0	1	0	1	1	1

Problema 9.3

Quanti livelli di riduttori 4-2 sono necessari per ridurre k addendi a 2 in un albero di riduzione? Quanti livelli sono necessari con riduttori 3-2?

Soluzione

Sia L il numero di livelli.

Per riduttori 4-2, abbiamo:

$$k(1/2)^L = 2$$

Prendendo i logaritmi in base 2 di ciascun lato di questa equazione, abbiamo:

$$\log_2 k - L = 1$$

ovvero:

$$L = \log_2 k - 1$$

Per riduttori 3-2, abbiamo:

$$k(2/3)^L = 2$$

Come sopra, prendendo i logaritmi in base 2, abbiamo:

$$\begin{aligned} \log_2 k + L(\log_2 2 - \log_2 3) &= \log_2 2 \\ \log_2 k + L(1 - 1{,}59) &= 1 \\ L &= (1 - \log_2 k)/(-0{,}59) \\ L &= 1{,}7 \log_2 k - 1{,}7 \end{aligned}$$

Queste espressioni sono solo approssimazioni (per difetto), a meno che il numero di addendi in ingresso a ciascun livello sia un multiplo di 4 nel caso di riduzione 4-2, o un multiplo di 3 nel caso di riduzione 3-2.

Problema 9.4

Convertire la frazione decimale 0,1 a una frazione binaria. Se la conversione non è esatta, dare l'approssimazione della frazione binaria a 8 bit dopo la virgola binaria usando ciascuno dei metodi di troncamento discussi nel Paragrafo 9.7.2.

Soluzione

Si usa il metodo di conversione dato nel Paragrafo 9.8. Moltiplicando la frazione decimale 0,1 per 2 ripetutamente, come mostrato in Figura 9.29, si genera la sequenza di bit 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, ... alla sinistra della virgola decimale, che continua indefinitamente, ripetendo la parte periodica 0, 0, 1, 1. Quindi la conversione non è esatta.

- Il troncamento tramite spezzamento dà 0,00011001.
- Il troncamento tramite arrotondamento di von Neumann dà 0,00011001.
- Il troncamento tramite arrotondamento dà 0,00011010.

Problema 9.5

Si consideri il seguente formato di rappresentazione di numeri in virgola mobile a 12 bit, comodo per lavorare su esercizi numerici. Il primo bit è il segno del numero. I successivi cinque

bit rappresentano un esponente in eccesso 15 per il fattore di scala, che ha 2 come base implicita. Gli ultimi sei bit rappresentano la mantissa.

Effettuare operazioni di sottrazione e moltiplicazione sugli operandi:

$A =$	0	10001	011011
$B =$	1	01111	101010

che rappresentano i numeri:

$$A = 1{,}011011 \times 2^2$$

e:

$$B = -1{,}101010 \times 2^0$$

Soluzione

Le operazioni richieste si effettuano come segue.

- Sottrazione.
  In accordo all'algoritmo di addizione algebrica nel Paragrafo 9.7.1, effettuiamo i seguenti quattro passi.
  1. Far scorrere la mantissa di B a destra di due posizioni di bit, ricavando 0,01101010.
  2. Porre l'esponente del risultato a 10001.
  3. Sottrarre il fattore intero di B da quello di A sommandoli, poiché B è negativo, ricavando il fattore intero provvisorio del risultato

	1	,	0	1	1	0	1	1	0	0
+	0	,	0	1	1	0	1	0	1	0
	1	,	1	1	0	1	0	1	1	0

  e porre il segno del risultato a 0 (positivo).
  4. Il risultato è in forma normalizzata, ma la mantissa va troncata a sei bit. Se questo viene fatto con arrotondamento, i due bit da rimuovere rappresentano un caso di pareggio, così arrotondiamo al numero pari più vicino aggiungendo 1, ottenendo un fattore intero di 1,110110. La risposta è:

$A - B =$	0	10001	110110

- Moltiplicazione.
  In accordo all'algoritmo di moltiplicazione nel Paragrafo 9.7.1, effettuiamo i seguenti tre passi.
  1. Addizionare gli esponenti e sottrarre 15 per ottenere 10001 come esponente provvisorio del risultato.
  2. Moltiplicare i fattori interi per ottenere 10,010110101110 come fattore intero provvisorio del risultato. Il segno del risultato è posto a 1 (negativo).
  3. Normalizzare il risultato provvisorio facendo scorrere la mantissa a destra di una posizione. Quindi sommare 1 all'esponente per ottenere 10010 come esponente del risultato. Troncare la mantissa a sei bit arrotondando per ottenere la risposta.

$A \times B =$	0	10010	001011

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

9.1 M Un *semiaddizionatore* (*half adder*) è un circuito logico combinatorio che ha due ingressi, x e y, e due uscite, s e c, che sono rispettivamente la somma e il riporto in uscita risultanti dall'addizione di x e y.

a. Progettare un semiaddizionatore tramite un circuito a due livelli AND-OR.
b. Mostrare come realizzare un addizionatore completo, quale mostrato in Figura 9.2a, usando due semiaddizionatori e necessarie porte logiche esterne.
c. Confrontare il più lungo percorso di ritardo logico attraverso la rete ottenuta nella parte **b.** a quello del ritardo della rete dell'addizionatore mostrata in Figura 9.2a.

9.2 M I metodi di rappresentazione binaria in complemento a uno e complemento a due sono casi speciali di tecniche di rappresentazione in complemento a $(b - 1)$ e complemento a b in sistemi di numerazione in base b. Per esempio, consideriamo il sistema decimale. I valori +526, −526, +70 e −70 hanno rappresentazioni a 4 cifre di intero con segno in ciascuno dei due sistemi di complemento, come mostrato in Figura P9.2. Il complemento a 9 è formato prendendo in ciascuna posizione il complemento a 9 della cifra.

Il complemento a 10 è formato aggiungendo 1 al complemento a 9. In ciascuna delle precedenti due rappresentazioni, la cifra più a sinistra è 0 per un numero positivo e 9 per un numero negativo.

Si consideri ora il sistema in base 3 (ternario), in cui il numero senza segno a 5 cifre $t_4t_3t_2t_1t_0$ ha il valore $t_4 \times 3^4 + t_3 \times 3^3 + t_2 \times 3^2 + t_1 \times 3^1 + t_0 \times 3^0$, con $0 \leq t_i \leq 2$. Esprimere i numeri con segno del sistema ternario +11011, −10222, +2120, −1212, +10 e −201 come numeri ternari con segno a 6 cifre nel sistema in complemento a 3.

9.3 M Rappresentare ciascuno dei valori decimali 56, −37, 122 e −123 come numeri con segno a 6 cifre nel formato ternario in complemento a 3, effettuare addizioni e sottrazioni su essi in tutte le possibili combinazioni di coppia e determinare se avviene o meno un trabocco aritmetico per ciascuna delle operazioni fatte (si veda il Problema proposto 9.2 per una definizione del sistema numerico ternario e, per la conversione di interi da decimale a ternario, si usi una tecnica analoga a quella data nel Paragrafo 9.8).

9.4 M Un addizionatore modulo 10 è necessario per sommare cifre BCD. L'addizione modulo 10 di due cifre BCD, $A = A_3A_2A_1A_0$ e $B = B_3B_2B_1B_0$, può essere ottenuta come segue:

Rappresentazione	Esempi			
Segno e valore assoluto	+526	−526	+70	−70
Complemento a 9	0526	9473	0070	9929
Complemento a 10	0526	9474	0070	9930

Figura P9.2 Numeri con segno in base 10 del Problema proposto 9.2.

sommare A a B (addizione binaria), quindi, se il risultato è un codice illegale, cioè maggiore o uguale a 10_{10}, sommare 6_{10} (ignorando il trabocco di questa addizione).

a. Quando si ha riporto in uscita uguale a 1?
b. Mostrare che questo algoritmo produce risultati corretti per:
  1. $A = 0101$ e $B = 0110$
  2. $A = 0011$ e $B = 0100$
c. Progettare un addizionatore di cifre BCD usando un addizionatore binario a 4 bit e le necessarie porte logiche esterne. Gli ingressi sono $A_3A_2A_1A_0$, $B_3B_2B_1B_0$ e un riporto in ingresso, le uscite sono le cifre di somma $S_3S_2S_1S_0$ e il riporto in uscita. Una cascata di tali blocchi forma un addizionatore BCD a propagazione di riporto.

9.5 F Mostrare che l'espressione logica $c_n \oplus c_{n-1}$ è un indicatore corretto di trabocco nell'addizione di interi in complemento a due usando una tabella di verità appropriata.

9.6 F Usare parti appropriate della soluzione nel Problema risolto 9.1 per calcolare quante porte logiche sono necessarie per costruire l'addizionatore a 16 bit con anticipo del riporto mostrato in Figura 9.5.

9.7 M In questo problema si investigano addizionatori con anticipo del riporto e i loro ritardi.

a. Progettare un addizionatore a 64 bit che usa quattro degli addizionatori a 16 bit con anticipo del riporto mostrati in Figura 9.5 assieme a circuiti logici aggiuntivi per generare c_{16}, c_{32}, c_{48} e c_{64} da c_0 e dalle variabili G_i^{II} e P_i^{II} mostrate nella figura. Qual è la relazione dei circuiti aggiuntivi con i circuiti logici di anticipo del riporto nella figura?
b. Mostrare che il ritardo attraverso l'addizionatore a 64 bit è di 12 ritardi di porta per s_{63} e 7 ritardi di porta per c_{64}, come asserito alla fine del Paragrafo 9.2.1.
c. Confrontare i ritardi di porta per produrre s_{31} e c_{32} nell'addizionatore a 64 bit della parte **a.** con i ritardi di porta per le stesse variabili nell'addizionatore a 32 bit costruito dalla cascata di due addizionatori a 16 bit, come discusso nel Paragrafo 9.2.1.

9.8 M Mostrare che il ritardo di caso peggiore attraverso una matrice $n \times n$ del tipo mostrato in Figura 9.6b è $6(n - 1) - 1$ ritardi di porta, come asserito nel Paragrafo 9.3.1.

9.9 F Moltiplicare ciascuna delle seguenti coppie di numeri con segno in complemento a due usando l'algoritmo di Booth. In ciascun caso, assumere che A sia il moltiplicando e B sia il moltiplicatore.

a. $A = 010111$ e $B = 110110$
b. $A = 110011$ e $B = 101100$
c. $A = 001111$ e $B = 001111$

9.10 M Ripetere il Problema proposto 9.9 usando la ricodifica bit-pair del moltiplicatore.

9.11 M Indicare in generale come modificare il diagramma di circuito in Figura 9.7a per realizzare la moltiplicazione di numeri a n bit in complemento a due con uso dell'algoritmo di Booth, specificando in modo chiaro gli ingressi e le uscite del Sequenzializzatore del controllo e ogni altro cambiamento necessario attorno all'addizionatore e al registro A.

9.12 M Estendere la tabella in Figura 9.14b a 16 righe, indicando come ricodificare tre bit del moltiplicatore: $i + 2$, $i + 1$ e i. Si possono generare tutte le versioni richieste del moltiplicando selezionate alla posizione i facendo scorrere e/o negando il moltiplicando M? Se no, quali versioni non possono essere generate in questo modo, e per quali casi sono richieste?

9.13 M Se il prodotto di due numeri a n bit nella rappresentazione in complemento a due è rappresentabile con n bit, si può usare l'algoritmo di moltiplicazione manuale mostrato in Figura 9.6a direttamente, trattando i bit di segno come gli altri bit. Provarlo per ciascuna delle seguenti coppie di numeri con segno a 4 bit:

a. moltiplicando = 1110 e moltiplicatore = 1101
b. moltiplicando = 0010 e moltiplicatore = 1110

Perché questo funziona correttamente?

9.14 D Si vuole usare un'unità aritmetica per interi che può eseguire l'addizione e la moltiplicazione di numeri senza segno a 16 bit, per moltiplicare due numeri senza segno a 32 bit. Tutti gli operandi, i risultati intermedi e i risultati finali sono in registri a 16 bit etichettati da R_0 a R_{15}. Il moltiplicatore hardware moltiplica il contenuto di R_i (moltiplicando) per R_j (moltiplicatore) e pone il prodotto di lunghezza doppia a 32 bit nei registri R_j e R_{j+1}, con la metà di ordine più basso in R_j. Quando $j = i - 1$, il prodotto riscrive entrambi gli operandi. L'addizionatore hardware somma i contenuti di R_i e R_j e pone il risultato in R_j. Il riporto in ingresso a un'operazione Add è 0 e il riporto in ingresso a un'operazione Add-con-riporto è il contenuto di una variabile C indicatore di riporto (bit di esito). Il riporto in uscita dall'addizionatore è sempre posto in C.

Specificare i passi di una procedura per moltiplicare due operandi a 32 bit nei registri R_1, R_0 e R_3, R_2, con le metà di ordine alto prima, depositando il prodotto di 64 bit nei registri R_{15}, R_{14}, R_{13} e R_{12}. Si può usare qualsiasi registro da R_{11} a R_4 per valori intermedi, se necessario. Ciascun passo nella procedura può essere un'operazione di moltiplicazione, addizione o trasferimento fra registri.

9.15 M In questo problema si studia il ritardo del moltiplicatore a matrice.

a. Calcolare il ritardo, in termini di ritardi dei blocchi addizionatori completi, nel produrre il bit di prodotto p_7 in ciascuno dei moltiplicatori a matrice 4×4 in Figura 9.16. Ignorare il ritardo della porta AND per generare tutti i prodotti m_iq_j all'inizio.
b. Sviluppare le espressioni del ritardo per ciascuna delle matrici in Figura 9.16 in funzione di n per il caso $n \times n$, come un'estensione della parte **a.** del problema. Quindi usare queste espressioni per calcolare il ritardo per il caso 32×32 per ciascuna matrice.

9.16 M In questo problema si analizza la profondità dell'albero di riduzione con salvataggio del riporto.

a. Quanti livelli di riduzione 3-2 sono necessari per ridurre 16 addendi a 2 usando uno schema simile a quello mostrato in Figura 9.19?
b. Ripetere la parte **a.** per ridurre 32 addendi a 2 per mostrare che l'asserto di 8 livelli nel Paragrafo 9.5.3 è corretto.
c. Confrontare le risposte esatte nelle parti **a.** e **b.** con i risultati ottenuti usando l'approssimazione sviluppata nel Problema risolto 9.3.

9.17 M Nei Paragrafi 9.5.3 e 9.5.4 è stata descritta la riduzione degli addendi usando riduttori 3-2 e 4-2. È anche possibile fare riduzioni 7-3 a ciascun livello di riduzione. Quando rimangono solo tre addendi, si fa una riduzione 3-2, seguita dall'addizione dei due addendi finali.

a. Quanti livelli di riduzione 7-3 sono necessari per ridurre 32 addendi a tre? Si confronti questo con i sette livelli necessari per ridurre 32 addendi a tre quando si usano riduzioni 3-2.
b. Il Problema risolto 9.3 mostra che sono necessari $\log_2 k - 1$ livelli di riduzione 4-2 per ridurre k addendi a 2 in un albero di riduzione. Quanti livelli di riduzione 7-3 sono necessari per ridurre k addendi a 3?

9.18 M Mostrare come realizzare un riduttore 4-2 usando due riduttori 3-2. La tabella di verità per questa realizzazione è differente da quella mostrata in Figura 9.21.

9.19 F Usando metodi manuali, esgeuire le operazioni $A \times B$ e $A \div B$ sui numeri a 5 bit senza segno $A = 10101$ e $B = 00101$.

9.20 M Mostrare come le operazioni di moltiplicazione e divisione nel Problema proposto 9.19 siano rispettivamente eseguite dall'hardware nelle Figure 9.7a e 9.23, costruendo diagrammi simili a quelli nelle Figure 9.7b e 9.25.

9.21 D Nel Paragrafo 9.7 è stato usato il formato di dimensione pratica a 32 bit dello standard IEEE per i numeri in virgola mobile. Qui si adotta un formato più corto che mantiene tutti i concetti pertinenti ma è meglio gestibile per elaborare esercizi numerici. Si consideri che i numeri a virgola mobile siano rappresentati in un formato a 12 bit come mostrato in Figura P9.21. Il fattore di scala ha una base implicita di 2 e un esponente a 5 bit in eccesso 15, con i due valori estremi 0 e 31 usati per indicare rispettivamente lo zero esatto e l'infinito. La mantissa a 6 bit è normalizzata come nel formato IEEE, con un 1 implicito a sinistra della virgola binaria.

a. Rappresentare i numeri +1,7, −0,012, +19 e 1/8 in questo formato.
b. Quali sono i numeri più piccoli e quelli più grandi rappresentabili in questo formato?
c. Come si confronta l'intervallo calcolato nella parte **b.** con l'intervallo di un intero con segno a 12 bit e di una frazione dell'unità (in virgola fissa) con segno a 12 bit?
d. Eseguire le operazioni Addizione, Sottrazione, Moltiplicazione e Divisione sugli operandi:

$A =$	0	10000	011011
$B =$	1	01110	101010

9.22 D Si considerino numeri a 16 bit in virgola mobile in un formato simile a quello discusso nel Problema proposto 9.21, con esponente a 6 bit e mantissa a 9 bit. La base del fattore di scala è 2 e l'esponente è rappresentato nel formato in eccesso 31.

a. Sommare i numeri A e B, rappresentati come segue:

$A =$	0	100001	111111110
$B =$	1	011111	001010101

Dare la risposta in forma normalizzata. Si ricordi che un 1 implicito è alla sinistra della virgola binaria ma non è incluso nei formati di A e B. Si usi l'arrotondamento come metodo di troncamento per produrre la mantissa finale.

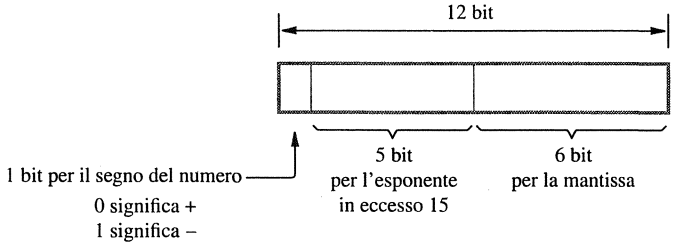


Figura P9.21 Formato di virgola mobile adottato nel Problema proposto 9.21.

b. Usando i numeri decimali w, x, y e z, esprimere l'ampiezza del valore più grande e del più piccolo (non nullo) rappresentabili nel precedente formato in virgola mobile normalizzato, con riferimento alla seguente forma:

$$\text{valore più grande} = w \times 2^x$$
$$\text{valore più piccolo} = y \times 2^{-z}$$

9.23 M In che modo la rappresentazione in eccesso x dell'esponente del fattore di scala nella rappresentazione dei numeri in virgola mobile di Figura 9.26a facilita il confronto dei valori di due numeri in virgola mobile? (Suggerimento: assumere che sia disponibile una rete logica combinatoria che confronti i valori di due interi senza segno a 32 bit. Usare questa rete, assieme alle necessarie porte logiche esterne, per progettare la rete richiesta per confrontare i numeri in virgola mobile.).

9.24 D Nel punto **a.** del Problema proposto 9.21, la conversione dei semplici numeri decimali al formato binario in virgola mobile è facile. Tuttavia, se i numeri decimali sono dati nel formato in virgola mobile, la conversione non è facile poiché non è possibile convertire separatamente la mantissa e l'esponente del fattore di scala, in quanto $10^x = 2^y$ non ammette che entrambi x e y siano interi non nulli. Si supponga che in un calcolatore sia memorizzata una tabella di numeri binari in virgola mobile t_i, tale che $t_i = 10^{x_i}$ per x_i nell'intervallo di esponenti rappresentabili del formato decimale in virgola mobile. Dare una procedura generale per convertire un dato numero decimale in virgola mobile al formato binario in virgola mobile. Si possono usare sia le istruzioni aritmetiche sugli interi sia quelle in virgola mobile disponibili nel calcolatore.

9.25 D Costruire un esempio per mostrare che sono necessari tre bit di guardia per produrre la risposta corretta quando si sottraggono due numeri positivi.

9.26 M Derivare espressioni logiche che specifichino le uscite Add/Sub e S_R della rete combinatoria CONTROLLO in Figura 9.28.

9.27 M Se il fan-in di porta è limitato a quattro, come si può realizzare lo SHIFTER in Figura 9.28 in modo combinatorio?

9.28 M Progettare una rete di porte logiche che realizzi il multiplatore MUX in Figura 9.28.

9.29 M Mettere in relazione la struttura della rete SCAMBIO in Figura 9.28 con la soluzione trovata per il Problema proposto 9.28.

9.30 M Come si può realizzare il rilevatore di zeri iniziali in Figura 9.28 in modo combinatorio?

9.31 M L'addizionatore-sottrattore di mantissa in Figura 9.28 opera su numeri binari positivi, senza segno, e deve produrre un risultato con segno e valore assoluto. Nella discussione che accompagna la Figura 9.28 si asserisce che l'aritmetica in complemento a uno è conveniente per via del formato richiesto degli operandi in ingresso e in uscita. Nella somma algebrica di due numeri in complemento a uno, il riporto in uscita dalla posizione del segno deve essere sommato al risultato per ottenere la corretta risposta con segno. Questa è chiamata *correzione da riporto circolare*. Si considerino i due esempi in Figura P9.31, che illustrano l'addizione usando codifiche a 4 bit con segno di operandi e risultati nel sistema in complemento a uno.

```
(3)        0 0 1 1          (6)        0 1 1 0
+(-5)   + [0] 1 0 0 1 1 0 0 0   +(-3)   + [1] 1 0 1 1 0 0 0 0
-2         | 1 1 0 1          3          | 0 0 1 0
           |------> 0                    |------> 1
           1 1 0 1                       0 0 1 1
```

Figura P9.31 Addizione in complemento a uno usata nel Problema proposto 9.31.

Il sistema aritmetico in complemento a uno è conveniente quando si deve generare un risultato in segno e valore assoluto, poiché un numero negativo nella notazione in complemento a uno può essere convertito alla forma con segno e valore assoluto complementando i bit a destra della posizione del bit di segno. Usando l'aritmetica in complemento a due, l'addizione di +1 è necessaria per convertire un valore negativo alla notazione con segno e valore assoluto. Se si usa un addizionatore con anticipo del riporto, è possibile incorporare nella logica di anticipo del riporto l'operazione di correzione da riporto circolare richiesta dall'aritmetica in complemento a uno. Con questa discussione come guida, dare la progettazione completa dell'addizionatore-sottrattore in complemento a uno richiesto in Figura 9.28.

9.32 M La rappresentazione in virgola fissa di frazioni binarie con segno in complemento a due è discussa nel Paragrafo 1.4.2.

a. Esprimere i valori decimali 0,5, −0,123, −0,75 e −0,1 come frazioni dell'unità a 6 bit con segno. Si veda il Paragrafo 9.8 per la conversione di frazioni da decimale a binario (con parte intera 0 implicita).
b. Qual è l'errore massimo di rappresentazione, *e*, in cui si incorre nell'usare solo 5 bit significativi dopo la virgola binaria?
c. Calcolare il numero di bit necessari dopo la virgola binaria cosicché l'errore massimo di rappresentazione *e* sia rispettivamente meno di 0,1, 0,01 o 0,001.

9.33 F Quali delle quattro risposte a 6 bit al punto **a.** del Problema proposto 9.32 non sono esatte? Per ciascuno di questi casi, dare i tre valori a 6 bit che corrispondono ai tre tipi di troncamento definiti nel Paragrafo 9.7.2.

10 Sistemi embedded

Problemi risolti

Problema 10.1

Si consideri la seguente attività. Un microcontrollore è utilizzato per monitorare lo stato di alcuni dispositivi meccanici. Le informazioni sullo stato sono disponibili per mezzo di segnali binari forniti su quattro linee. I 16 valori possibili dello stato devono essere visualizzati come una cifra esadecimale su uno schermo a sette segmenti del tipo mostrato in Figura 3.17.

L'operazione desiderata può essere ottenuta utilizzando l'interfaccia parallela mostrata nelle Figure da 10.4 a 10.6. Si assuma che le quattro linee di ingresso siano collegate ai piedini della porta A e che le sette linee di dati dello schermo a sette segmenti siano collegate alla porta B. Quindi, i registri di direzione dei dati, PADIR e PBDIR, devono configurare le porte A e B rispettivamente come di ingresso e di uscita. I dati in ingresso possono essere letti direttamente dai piedini della porta A. Scrivere un programma C che configuri le porte e realizzi l'operazione descritta.

Soluzione

La Figura P10.1 presenta un possibile programma. Si usano dichiarazioni *define* per associare le costanti degli indirizzi richiesti ai nomi simbolici dei puntatori.

Si noti che il puntatore PAIN è dichiarato *volatile*. Ciò è necessario poiché il programma si limita a leggere il contenuto della locazione corrispondente, tuttavia non scrive dati in essa, né le associa un valore specifico. Un compilatore ottimizzante potrebbe rimuovere le istruzioni di programma che sembrino non avere alcun impatto. Ciò include le istruzioni che coinvolgono variabili i cui valori non mutano mai. Poiché il contenuto del registro PAIN cambia per fonti esterne al programma, è essenziale informare il compilatore di questo fatto. Il compilatore non rimuoverà le istruzioni che contengono variabili che sono state dichiarate volatili.

Si utilizza una tabella per tradurre una cifra esadecimale in una corrispondente figura di sette segmenti. Il programma usa un ciclo continuo per leggere e visualizzare i nuovi dati. In un'applicazione reale, non è probabile che si utilizzi un ciclo in questo modo poiché sarebbero in gioco altre attività. Qui si usa il ciclo continuo solo per semplicità illustrativa.

Problema 10.2

Si consideri adesso il caso di un dispositivo di I/O in grado di inviare informazione bit per bit in un formato seriale che può essere gestito dall'interfaccia seriale mostrata nelle Figure 10.7 e 10.8. Il dispositivo invia 8 bit d'informazione che devono essere visualizzati come due cifre esadecimali su due schermi a 7 segmenti del tipo in Figura 3.17. Il dispositivo di I/O è collegato all'interfaccia seriale, e gli schermi a 7 segmenti alle porte parallele A e B. Quindi, sia A sia B devono essere configurate come porte di uscita. Scrivere due programmi C che configurino le porte e realizzino l'operazione descritta, rispettivamente usando la scansione e le interruzioni per la lettura dei dati di ingresso.

```
/* Definisci indirizzi dei registri */
#define  PAIN    (volatile unsigned char *) 0xFFFFFFF0
#define  PADIR   (volatile unsigned char *) 0xFFFFFFF2
#define  PBOUT   (volatile unsigned char *) 0xFFFFFFF4
#define  PBDIR   (volatile unsigned char *) 0xFFFFFFF5
#define  PCONT   (volatile unsigned char *) 0xFFFFFFF7

/* Tabella di conversione da esadecimale a 7 segmenti */
unsigned char table[16] = { 0x40, 0x79, 0x24, 0x30, 0x19, 0x12,
0x02, 0x78, 0x00, 0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E };
unsigned int current_value;

void main()
{
    /* Inizializza porte A e B */
    *PADIR = 0x0;          /* Configura la porta A come ingresso. */
    *PBDIR = 0xFF;         /* Configura la porta B come uscita. */
    *PCONT = 0x0;          /* Leggi gli ingressi direttamente dai piedini. */

    /* Leggi e visualizza i dati. */
    while (1)              /* Ciclo continuo. */
    {
        current_value = *PAIN & 0x0F;      /* Leggi l'ingresso dalla Porta A. */
        *PBOUT = table[current_value];     /* Invia il carattere alla Porta B. */
    }
}
```

Figura P10.1 Programma C per il Problema risolto 10.1.

Soluzione

Un possibile programma è presentato in Figura P10.2a. Si usa il metodo di scansione per determinare quando siano disponibili nuovi dati nel buffer di ricezione dell'interfaccia seriale. Il bit $SSTAT_0$ serve da bit di stato che viene interrogato. Come descritto nel Paragrafo 10.3.2, questo bit è azzerato quando i dati vengono letti dal buffer.

La Figura P10.2b mostra come si possono usare le interruzioni per accedere a nuovi dati. Si ricordi dal Paragrafo 10.3.4 che il vettore di interruzione IRQ è alla locazione di memoria 0x20. L'indirizzo della routine di servizio dell'interruzione è posto in questa locazione. Il bit $SCONT_4$ è posto a 1 per abilitare la ricezione delle interruzioni. Per far sì che il processore risponda alle interruzioni IRQ, il bit 6 nel registro di stato del processore, PSR, deve essere posto a 1. Poiché PSR non è una locazione nello spazio indirizzabile, è necessario utilizzare la direttiva *asm* per l'inserimento in-linea dell'istruzione assemblativa:

```
MoveControl    PSR, #0x40
```

Inoltre, è necessario includere l'istruzione di rientro da interruzione nel programma oggetto per assicurare un corretto ritorno al programma interrotto. Il compilatore inserirà questa istruzione poiché la definizione della funzione *intserv* comprende la parola chiave *interrupt*. Questo modo di gestire interruzioni in un linguaggio di alto livello è spiegato nel Paragrafo 4.6.


```
/* Definisci indirizzi dei registri */
#define  RBUF    (volatile unsigned char *) 0xFFFFFFE0
#define  SSTAT   (volatile unsigned char *) 0xFFFFFFE2
#define  PAOUT   (volatile unsigned char *) 0xFFFFFFF1
#define  PADIR   (volatile unsigned char *) 0xFFFFFFF2
#define  PBOUT   (volatile unsigned char *) 0xFFFFFFF4
#define  PBDIR   (volatile unsigned char *) 0xFFFFFFF5

/* Tabella di conversione da esadecimale a 7 segmenti */
unsigned char table[16] = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12,
0x02, 0x78, 0x00, 0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E};
unsigned int current_value, low_digit, high_digit;

void main()
{
    /* Inizializza le porte parallele */
    *PADIR = 0xFF;         /* Configura la porta A come uscita. */
    *PBDIR = 0xFF;         /* Configura la porta B come uscita. */

    /* Leggi e visualizza i dati. */
    while (1)              /* Ciclo continuo. */
    {
        while ((*SSTAT & 0x1) == 0);       /* Attesa di nuovi dati. */
        current_value = *RBUF;             /* Leggi il valore a 8 bit. */
        low_digit = current_value & 0x0F;
        high_digit = (current_value > > 4) & 0x0F;
        *PAOUT = table[low_digit];         /* Invia le due cifre */
        *PBOUT = table[high_digit];        /* allo schermo a 7 segmenti. */
    }
}
```

Figura P10.2a Programma C per il Problema risolto 10.2 che usa la scansione per leggere i dati di ingresso.

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

10.1 M Nel Problema risolto 10.2, si è ipotizzato che il dispositivo di I/O sia in grado di inviare dati a 8 bit in modo seriale. Si consideri ora un dispositivo simile che usi otto linee per inviare i dati in parallelo. Pertanto, una delle porte parallele del microcontrollore deve essere utilizzata per ricevere i dati. Questo lascia solo una porta parallela per visualizzare due cifre esadecimali che rappresentano i dati. Quindi si può usare un solo schermo a 7 segmenti del tipo mostrato in Figura 3.17. Per rendere l'informazione ricevuta, si può usare un singolo dispositivo visualizzando le cifre una dopo l'altra. Si faccia sì che si visualizzi la cifra più significativa per un secondo, seguita da una pausa di un secondo, quindi si visualizzi la seconda cifra per un secondo. Dopo aver visualizzato la seconda cifra, lo schermo dovrebbe mostrare un "trattino" per due secondi. Questa sequenza è ripetuta continuamente, per riflettere i dati ricevuti più di recente. Usare il temporizzatore in Figura 10.9, assumendo che esso

```
#define  RBUF   (volatile unsigned char *) 0xFFFFFFE0
#define  SCONT  (volatile unsigned char *) 0xFFFFFFE3
#define  PAOUT  (volatile unsigned char *) 0xFFFFFFF1
#define  PADIR  (volatile unsigned char *) 0xFFFFFFF2
#define  PBOUT  (volatile unsigned char *) 0xFFFFFFF4
#define  PBDIR  (volatile unsigned char *) 0xFFFFFFF5
#define  IVECT  (volatile unsigned int *) 0x20

/* Tabella di conversione da esadecimale a 7 segmenti */
unsigned char table[16] = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12,
0x02, 0x78, 0x00, 0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E};
unsigned int current_value, low_digit, high_digit;

interrupt void intserv();

void main()
{
    /* Inizializza la porta parallela */
    *PADIR = 0xFF;                      /* Configura la porta A come uscita. */
    *PBDIR = 0xFF;                      /* Configura la porta B come uscita. */

    /* Inizializza il meccanismo di interruzione */
    *IVECT = (unsigned int *) &intserv;  /* Imposta il vettore delle interruzioni. */
    asm ("MoveControl PSR, #0x40");     /* Rispondi a interruzioni IRQ. */
    *SCONT = 0x10;                      /* Abilita le interruzioni dal ricevitore. */

    while (1);                          /* Ciclo continuo. */
}

/* Routine di servizio delle interruzioni */
interrupt void intserv()
{

    current_value = *RBUF;              /* Leggi il valore a 8 bit. */
    low_digit = current_value & 0x0F;
    high_digit = (current_value > > 4) & 0x0F;
    *PAOUT = table[low_digit];          /* Invia le due cifre */
    *PBOUT = table[high_digit];         /* allo schermo a 7 segmenti. */
}
```

Figura P10.2b Programma C per il Problema risolto 10.2 che usa le interruzioni per leggere i dati di ingresso.

sia pilotato da un clock a 100 MHz. Mostrare le connessioni hardware necessarie e stilare un programma per realizzare il compito richiesto. Usare la scansione per controllare lo stato del temporizzatore.

10.2 M Risolvere il Problema proposto 10.1 usando le interruzioni del temporizzatore.

10.3 M Il microcontrollore descritto nel Paragrafo 10.3 viene utilizzato per controllare un sistema dotato di un motore che può funzionare a due velocità, alta e bassa. Un sensore associato al motore usa una singola linea di segnale per indicare la velocità attuale del motore. Per la velocità alta, il sensore trasmette un segnale continuo a onda quadra alla frequenza di 100 kHz. Per la velocità bassa trasmette un segnale a 50 kHz. Non trasmette alcun segnale (ovvero un valore logico 0 costante) se il motore non è in funzione. Il microcontrollore usa uno schermo a 7 segmenti del tipo mostrato in Figura 3.17 per visualizzare lo stato del motore come F (velocità alta, *fast*), S (velocità bassa, *slow*) o 0. Mostrare le connessioni hardware necessarie e stilare un programma per la realizzazione del compito richiesto. Si usi il temporizzatore in Figura 10.9 per generare gli intervalli di tempo necessari a misurare la frequenza del segnale inviato dal sensore. Si supponga che il temporizzatore sia pilotato da un clock a 100 MHz. Si usi la scansione per controllare lo stato del temporizzatore.

10.4 M Risolvere il Problema proposto 10.3 usando le interruzioni del temporizzatore.

10.5 D Il microcontrollore nel Paragrafo 10.3 viene utilizzato per ricevere numeri decimali sulla sua porta seriale. Ciascun numero è composto da due cifre codificate come caratteri ASCII. Per distinguere numeri successivi a due cifre si usa un carattere delimitatore H. Pertanto, se due numeri in successione sono 43 e 28, la sequenza ricevuta sarà H43H28. Ogni numero deve essere visualizzato su due schermi a 7 segmenti collegati alle porte parallele A e B. Il carattere di delimitazione non deve essere visualizzato. Il numero visualizzato dovrebbe cambiare solo quando entrambe le cifre del numero successivo sono state ricevute. Mostrare le connessioni necessarie per realizzare questa funzione. Etichettare i segmenti dello schermo, come indicato in Figura 3.17. Stilare un programma per eseguire il compito richiesto. Si usi l'interfaccia seriale in Figura 10.8. Si usi la scansione per rilevare l'arrivo di ciascun carattere ASCII.

10.6 D Risolvere il Problema proposto 10.5 utilizzando le interruzioni per rilevare l'arrivo di ciascun carattere ASCII.

10.7 D Il microcontrollore nel Paragrafo 10.3 viene utilizzato per ricevere numeri decimali sulla sua porta seriale. Ogni numero è costituito da quattro cifre codificate in caratteri ASCII. Per distinguere numeri successivi a quattro cifre si usa un carattere delimitatore H. Quindi, se due numeri in successione sono 2143 e 6292, la sequenza ricevuta sarà H2143H6292. Ogni numero va mostrato su quattro unità di visualizzazione a 7 segmenti. Si supponga che ogni unità abbia associato un circuito decodificatore da BCD a 7 segmenti, come mostrato in Figura P10.7. Mostrare le connessioni al microcontrollore necessarie. Stilare

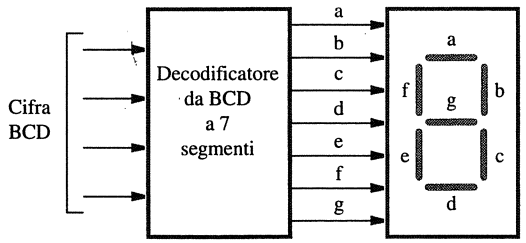


Figura P10.7 Schermo a 7 segmenti con decodificatore BCD per il Problema proposto 10.7.

un programma per eseguire il compito richiesto. Si usi l'interfaccia seriale in Figura 10.8. Usare la scansione per rilevare l'arrivo di ciascun carattere ASCII.

10.8 D Risolvere il Problema proposto 10.7 utilizzando le interruzioni per rilevare l'arrivo di ciascun carattere ASCII.

10.9 D Ripetere il Problema proposto 10.7 assumendo che ciascuna unità a 7 segmenti abbia un registro a 7 bit associato, invece di un decodificatore da BCD a 7 segmenti. Il registro ha un ingresso di controllo *Load*, che fa sì che i 7 bit di dati siano caricati nel registro quando *Load* = 1. Ciascun bit del registro pilota un segmento dell'unità a cui è associato. La Figura P10.9 mostra la disposizione registro-schermo. Disporre le connessioni di uscita del microcontrollore in modo che la porta parallela A fornisca i dati a tutte e quattro le unità.

10.10 D Risolvere il Problema proposto 10.9 usando le interruzioni per rilevare l'arrivo di ciascun carattere ASCII.

10.11 F Modificare il cronometro di riflessi presentato nel Paragrafo 10.4 supponendo che la persona da esaminare risponderà sempre in meno di un secondo. Quindi, il tempo di reazione trascorso può essere visualizzato usando solo due cifre che rappresentano i centesimi di secondo. Collegare i due schermi a 7 segmenti alla porta A e modificare il programma in Figura 10.11 per realizzare l'operazione desiderata.

10.12 M In Figura 10.10, lo schermo a 7 segmenti per ciascuna cifra incorpora un decodificatore da BCD a 7 segmenti, quindi il microcontrollore fornisce un codice BCD a 4 bit per ogni cifra da visualizzare. Si supponga che in luogo del decodificatore ciascuno schermo a 7 segmenti abbia un registro a 7 bit con un ingresso di controllo *Load*, in modo tale che i sette bit di dati siano caricati nel registro quando *Load* = 1. Ogni bit nel registro pilota un segmento dell'unità di visualizzazione associata. La Figura P10.9 mostra la disposizione registro-schermo. Modificare il programma in Figura 10.11 per l'uso con questo circuito registro-schermo.

10.13 M Si utilizzi il microcontrollore nel Paragrafo 10.3 per generare una visualizzazione dell'orario del giorno. L'orario (in ore e minuti) deve essere mostrato su quattro unità LCD a 7 segmenti. Si supponga che ogni unità LCD abbia associato un decodificatore da

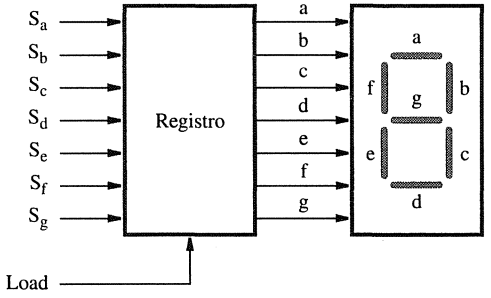


Figura P10.9 Schermo a 7 segmenti con registro per il Problema proposto 10.9.

BCD a 7 segmenti, come mostrato in Figura P10.7. Si supponga, inoltre, di usare un clock a 100 MHz. Mostrare i collegamenti hardware necessari e scrivere un programma adatto.

10.14 M Ripetere il Problema proposto 10.13 assumendo che ogni unità LCD a 7 segmenti abbia associato un registro, come mostrato in Figura P10.9.

10.15 F In un sistema realizzato su singolo chip, il processore e la memoria principale risiedono sullo stesso chip. C'è bisogno di una cache in questo sistema? Spiegare.

11 Elaborazione parallela e prestazioni

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

11.1 M Si supponga che un trasferimento su bus duri T secondi e il tempo di accesso alla memoria sia $4T$ secondi. Una richiesta di lettura su un bus convenzionale richiede quindi $6T$ secondi per completare. Quanti bus convenzionali sono necessari per eguagliare o superare il throughput effettivo di un bus a transazioni spezzate che opera con gli stessi ritardi di tempo? Si considerino, ora, solo le richieste di lettura, si ignorino i conflitti di memoria e si supponga che tutti i moduli di memoria siano collegati a tutti i bus nel caso di più bus. Il risultato aumenta o diminuisce se il tempo di accesso alla memoria aumenta?

11.2 M Si supponga che un calcolo comprenda $k + 1$ attività distinte. Al fine di preparare un programma per il calcolo desiderato, ciascuna di queste attività è stata scritta come una funzione nel linguaggio C. Le $k + 1$ funzioni vengono etichettate T0(), T1(), ... , Tk(). Ogni funzione richiede τ unità di tempo per l'esecuzione. A causa di dipendenze di dati, le funzioni da T1() a Tk() devono essere eseguite dopo la funzione T0(). Non vi sono dipendenze di dati fra le funzioni da T1() a Tk().

a. Usando le funzioni date, scrivere un programma C da eseguire su un singolo processore.
b. Scrivere un programma C equivalente da eseguire su k processori.
c. Ricavare un'espressione per il guadagno di prestazione ideale per il programma nella parte **b.** relativamente al programma nella parte **a.**

11.3 D Quali sono gli argomenti a favore e contro l'invalidazione e l'aggiornamento come strategie per mantenere la coerenza di cache?

11.4 D Gli approcci memoria condivisa e scambio di messaggi supportano entrambi l'esecuzione simultanea di attività che interagiscono fra loro. Quale di questi due approcci può emulare il funzionamento dell'altro più facilmente? Giustificare brevemente la risposta.

11.5 F Con riferimento alla Figura 11.1, si assuma una dimensione della matrice di $N = 32$. Si assuma che tutte le istruzioni richiedano una unità di tempo per l'esecuzione in un processore. Determinare il guadagno di prestazione della vettorizzazione per lunghezze di vettore di 4, 8, 16 e 32.

11.6 M Per la vettorizzazione, l'efficienza è definita come il rapporto tra il guadagno di prestazione e la lunghezza del vettore. Determinare l'efficienza per ciascuno dei risultati nel Problema 11.5. Commentare sulla variazione dell'efficienza con la lunghezza del vettore.

11.7 M Nel programma parallelo in Figura 11.8 per calcolare il prodotto scalare, la somma dei prodotti parziali è calcolata da un processore dopo che tutti i thread hanno raggiunto la barriera. Modificare il programma in modo che ogni thread aggiunga la sua somma parziale incrementalmente a una somma globale per il prodotto scalare. Per evitare che due o più thread tentino di aggiornare la somma globale allo stesso tempo, è necessaria la sincronizzazione. Un metodo è quello di utilizzare una variabile contatore condivisa il cui valore è l'identificatore del thread a cui è concesso l'accesso esclusivo alla variabile di somma globale. Dopo che il thread con accesso esclusivo abbia aggiornato la somma globale, può semplicemente incrementare questo contatore per concedere l'accesso esclusivo a un altro thread.

11.8 F Si supponga che un multiprocessore abbia otto processori. Sulla base di un programma esistente che gira su un processore, un programma parallelo è scritto per girare su questo multiprocessore. Si supponga che il carico di lavoro della porzione parallela del programma possa essere distribuito uniformemente sugli otto processori. Usare la Legge di Amdahl per determinare il valore di f_e che consentirebbe un guadagno di prestazione pari a 5.

A Elementi di logica della commutazione

Problemi proposti

Soluzioni sul sito www.ateneonline.it/hamacher3e

A.1 F Esprimere in forma di somma di prodotti (SOP) la funzione booleana binaria EQ, definita come complemento di XOR.

A.2 M Quali regole dell'algebra di Boole giustificano la dimostrazione dell'universalità delle porte logiche NAND e NOR?

A.3 M Dimostrare la validità delle seguenti identità booleane, usando le regole dell'algebra di Boole e/o le tabelle di verità:

a. $\overline{x \oplus y} \oplus z = \bar{x}\,\bar{y}\,\bar{z} + x\,y\,\bar{z} + \bar{x}\,y\,z + x\,\bar{y}\,z$

b. $x + y\,\bar{x} = x + y$

c. $x_1\,\overline{x_2} + \overline{x_2}\,x_3 + x_3\,\overline{x_1} = x_1\,\overline{x_2} + x_3\,\overline{x_1}$

A.4 M Si consideri la funzione booleana definita dalla seguente espressione:

$$f(x_1, x_2, x_3, x_4) = (x_1 \oplus x_3) + (x_1\,x_3 + \overline{x_1}\,\overline{x_3})\,x_4 + x_1\,\overline{x_2}$$

a. Determinare una forma SOP di costo minimo per f usando una mappa di Karnaugh.

b. Trovare, con lo stesso metodo, una forma SOP di costo minimo per la sua complementare, cioè la funzione $\bar{f}$ che vale 1 se, e solo se, f vale 0. Prendere quindi il complemento di tale espressione, usando la regola di De Morgan, per determinare una nuova espressione per f. Quest'ultima sarà in forma POS (prodotto di somme). Confrontarne il costo (con il criterio dei letterali) con quello della sua forma SOP determinata nella parte **a.**. Si possono trarre conclusioni generali da questo risultato?

A.5 F Sintetizzare una rete combinatoria di costo minimo, secondo il criterio di costo dei letterali, della funzione booleana $f(x_1, x_2, x_3, x_4)$, dove $f = 1$ se almeno una e non più di due delle variabili d'ingresso hanno valore logico 1, altrimenti $f = 0$.

A.6 F Determinare forme SOP di costo minimo (secondo il criterio dei letterali) per ciascuna delle quattro funzioni booleane in tre variabili specificate dalle tabelle di verità sottostanti. Si noti che due delle dette funzioni sono parziali, come indicato dalle condizioni di indifferenza caratterizzate da "x" nella tabella. Qualcuna delle quattro funzioni ha più di una forma SOP di costo minimo? Se sì, determinarle tutte.

x_1	x_2	x_3	f_1	f_2	f_3	f_4
0	0	0	1	1	x	0
0	0	1	1	1	1	1
0	1	0	0	1	0	1
0	1	1	0	1	1	x
1	0	0	1	0	x	x
1	0	1	0	0	0	x
1	1	0	1	0	1	1
1	1	1	1	1	1	0

A.7 F Determinare una forma SOP di costo minimo (secondo il criterio dei letterali) per la funzione booleana parziale f in quattro variabili che coincide con quella specificata dal termine:

$$x_1(x_2\,\overline{x_3} + x_2\,x_3 + \overline{x_2}\,\overline{x_3}\,x_4) + x_2\,\overline{x_4}(\overline{x_3} + x_1)$$

eccetto che f è indefinita quando è vera la condizione di indifferenza x specificata dal termine:

$$x = x_1\,\overline{x_2}(x_3\,x_4 + \overline{x_3}\,\overline{x_4}) + \overline{x_1}\,\overline{x_3}\,x_4$$

A.8 M La codifica BCD a 4 bit delle cifre decimali è definita in Figura A.6. Progettare un circuito a quattro ingressi $b_3, \ldots b_0$ e un'uscita f tale che $f = 1$ se la configurazione dei quattro bit in ingresso è una codifica BCD valida di una cifra, altrimenti $f = 0$. Esibire una realizzazione di costo minimo del circuito.

A.9 M Si debbano confrontare due numeri a 2 bit $A = a_1a_0$ e $B = b_1b_0$ mediante una funzione booleana di quattro variabili $f(a_1, a_0, b_1, b_0)$. f deve assumere il valore 1 se, e solo se, $v(A) \leq v(B)$, dove $v(X) = x_1 \cdot 2^1 + x_0 \cdot 2^0$ per qualsiasi numero X a 2 bit. Assumere che le variabili A e B soddisfino $|v(A) - v(B)| \leq 2$. Sintetizzare f usando il minor numero possibile di porte logiche.

A.10 M Risolvere la variante del Problema A.9 per il requisito che $f = 1$ se, e solo se, $v(A) > v(B)$, assumendo invece che gli ingressi soddisfino $v(A) + v(B) \leq 4$.

A.11 F Dimostrare che l'operatore booleano NAND non è associativo.

A.12 M Progettare una rete combinatoria costituita da non più di sei porte NAND, ciascuna a tre ingressi, che realizzi la seguente funzione booleana, assumendo che gli ingressi siano disponibili sia con i loro valori sia con quelli complementati:

$$f = x_1\,x_2 + x_1\,x_2\,x_3 + \overline{x_1}\,\overline{x_2}\,\overline{x_3}\,x_4 + \overline{x_1}\,\overline{x_2}\,x_3\,\overline{x_4}$$

A.13 M Progettare una rete combinatoria di porte NAND a due ingressi che realizzi la seguente funzione booleana, impiegando non più di 6 porte NAND: $f = x_1\,x_2 + \overline{x_3} + \overline{x_1}\,x_4$. Non sono disponibili variabili di ingresso complementate.

A.14 F Progettare una rete combinatoria di porte NAND a due ingressi che realizzi la seguente funzione booleana nel modo più economico possibile: $f = (x_1 + x_3)\,(\overline{x_2} + \overline{x_4})$. Si assuma che non siano disponibili variabili di ingresso complementate.

A.15 M Progettare una rete combinatoria che realizzi la funzione booleana XOR usando solo 4 porte NAND a due ingressi.

A.16 F Quali sono i principali vantaggi della tecnologia CMOS rispetto a quella NMOS?

A.17 F Progettare un circuito CMOS che realizzi la funzione booleana di una porta logica OR a tre ingressi.

A.18 M Progettare un circuito CMOS che realizzi la funzione booleana di una porta logica NAND a tre ingressi.

A.19 M Progettare un circuito CMOS che realizzi la funzione booleana di una porta logica XOR.

A.20 M Progettare un circuito CMOS che realizzi la funzione booleana $f(x_1, x_2, x_3, x_4) = \overline{x_1}\,\overline{x_2} + \overline{x_3}\,\overline{x_4}$ usando il minimo numero di transistori.

A.21 M Stilare la tabella di transizione del bistabile asincrono di struttura simile a quella del latch SR, ma con porte NAND invece che NOR. Indicare come si può usare questo circuito per costruire un bistabile sincrono SR.

A.22 M Stilare la tabella di transizione del bistabile sincrono di struttura simile a quella del latch sincrono di tipo D, ma con porte NOR invece che NAND. Si può usare questo circuito per costruire un bistabile sincrono di tipo D? Se sì, come? Se no, perché?

A.23 M Calcolare sia il tempo di impostazione sia il tempo di mantenimento per il flip-flop di tipo D edge-triggered negativo mostrato in Figura A.29, espressi in termini dei ritardi delle porte NOR.

A.24 D Si consideri un circuito sequenziale sincrono in cui il segnale di clock sia simultaneamente inviato a uno degli ingressi di una porta AND e a una porta NOT, la cui uscita vada all'altro ingresso della porta AND. Da un punto di vista meramente logico, l'uscita della porta AND dovrebbe avere valore logico costante 0. Se però si tiene conto delle caratteristiche fisiche del circuito, in particolare del ritardo di propagazione attraverso le porte logiche, risulta che il segnale in uscita dalla porta AND ha ben altra forma d'onda. Quale? Potrebbe essere utile combinare questo circuito con quello di un flip-flop di tipo master-slave? Se sì, a quale fine? Se no, perché?

A.25 M La Figura A.33 mostra un registro a scorrimento di tipo semplice, dove i dati scorrono a destra di una posizione per volta sotto il controllo di un segnale di clock. Modificare questo registro per ottenere lo scorrimento a destra di una o due posizioni per volta, controllato dal clock e da un ulteriore segnale di controllo UNO/DUE.

A.26 D Si vuole realizzare un registro a scorrimento da 4 bit che abbia due ingressi di controllo: INI e DX/SX. Quando INI è impostato a 1, si deve caricare il numero binario 1000 nel registro, indipendentemente dall'input di clock. Quando INI = 0, gli impulsi all'input di clock devono causare la rotazione del contenuto del registro. La rotazione ha luogo verso destra o sinistra quando DX/SX rispettivamente vale 1 o 0. Produrre un progetto adatto per questo registro usando flip-flop di tipo D con preset e clear, come mostrato in Figura A.32.

A.27 D Il contatore binario a 3 bit mostrato in Figura A.35 è detto "verso l'alto" perché il conteggio procede per incrementi, mentre uno in cui il conteggio procede per decrementi è detto "verso il basso". Se il conteggio può procedere in entrambe le direzioni, sotto il controllo di un segnale A/B, il contatore è detto "bidirezionale". Tracciare lo schema logico di un contatore bidirezionale che sia anche inizializzabile a uno stato qualsiasi mediante scrittura parallela dei suoi flip-flop da una fonte esterna, aggiungendovi un ulteriore segnale di controllo LOAD/COUNT per determinare se il modulo viene inizializzato o funziona da contatore.

A.28 D Il contatore binario a 3 bit mostrato in Figura A.35 è "asincrono" poiché i suoi flip-flop non sono pilotati simultaneamente da un unico segnale di clock. A causa del ritardo di propagazione nei flip-flop si hanno quindi piccoli scostamenti temporali delle loro transizioni di stato rispetto ai multipli del periodo, come indicato in figura. Un contatore binario "sincrono" non presenta questo fenomeno, poiché i suoi flip-flop sono tutti pilotati dallo stesso clock primario. Tracciare lo schema logico di un contatore binario sincrono a 4 bit, mantenendo l'uso di flip-flop di tipo T, i cui ingressi di clock sono però tutti connessi direttamente al clock primario.

A.29 M Tracciare la rete combinatoria di un decodificatore a tre ingressi e otto uscite, facendo uso di porte logiche a non più di due ingressi.

A.30 D Un registro seriale-parallelo come quello mostrato in Figura A.34 può essere impiegato per la memorizzazione di dati provenienti da una linea di comunicazione seriale. Il problema converso, ovvero quello della trasmissione di dati, memorizzati su registri a scrittura parallela, su una linea di uscita seriale può essere risolto mediante un multiplatore opportunamente combinato con un contatore binario. Tracciare lo schema logico di un modulo che realizzi tale funzione.

A.31 M Si consideri la funzione logica specificata dall'espressione:

$$f(x_1, x_2, x_3, x_4) = x_1 x_3 \overline{x_4} + \overline{x_1}\,\overline{x_3} x_4 + \overline{x_2}\,\overline{x_3}\,\overline{x_4}$$

a. Mostrare una realizzazione di f ottenuta usando un multiplatore a tre ingressi di selezione, sintetizzandone la tabella di verità (come esemplificato in Figura A.39).

b. Si può realizzare la stessa funzione mediante un multiplatore a due ingressi di selezione? Se sì, mostrare come.

A.32 M Risolvere la variante del Problema A.31 per la funzione logica specificata dall'espressione $f(x_1, x_2, x_3, x_4) = x_1 \overline{x_2} x_3 + x_2 x_3 x_4 + \overline{x_1}\,\overline{x_4}$.