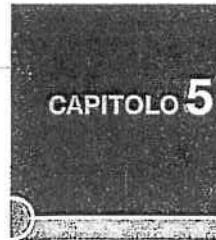


22. Dopo aver esaminato, nell'esercizio precedente, versioni più economiche della IFU, consideriamo ora quelle più costose. Ci potrebbe essere una qualche ragione per avere registri di scorrimento più grandi, per esempio di 12 byte? Si motivi la risposta.
23. Nel microprogramma Mic-2, il codice dell'istruzione `if_icmpeq6` porta a T quando Z è impostato a 1. Il codice in T è tuttavia identico a quello in `goto1`. Sarebbe stato possibile andare direttamente in `goto1`? Così facendo la macchina sarebbe risultata più veloce?
24. In Mic-4 l'unità di decodifica fa corrispondere al codice operativo IJVM un indice della ROM dove sono memorizzate le micro-operazioni corrispondenti. Potrebbe sembrare più semplice omettere semplicemente la fase di decodifica e inserire direttamente il codice operativo IJVM nella coda. Si potrebbe usare il codice operativo di IJVM come un indice all'interno della ROM, allo stesso modo di Mic-1. Che cosa c'è di errato in questa strategia?
25. Perché i computer sono equipaggiati con più livelli di cache? Non sarebbe meglio avere semplicemente un unico livello di cache più grande?
26. Un calcolatore ha due livelli di cache. Supponiamo che il 60% dei riferimenti alla memoria sia soddisfatto dalla cache di primo livello, il 35% da quella di secondo livello e il 5% fallisca con entrambe le cache. I tempi di accesso sono rispettivamente di 5 ns, 15 ns e 60 ns. I tempi relativi alla cache di secondo livello e alla memoria iniziano a partire dal momento in cui si sa che è necessario accedere a queste memorie (per esempio, un accesso alla cache di secondo livello non comincia prima di un fallimento della cache di primo livello). Qual è il tempo medio di accesso?
27. Alla fine del Paragrafo 4.5.1 abbiamo detto che la tecnica della *write allocation* è vantaggiosa solo se è probabile che in una riga ci siano più scritture nella stessa linea. Che cosa si può dire riguardo il caso in cui una scrittura sia seguita da più letture?
28. Nella prima stesura di questo libro la Figura 4.39 mostrava una cache associativa a tre vie invece di una a quattro vie. Uno dei revisori del testo ha avuto un improvviso scatto d'ira, affermando che quella figura avrebbe confuso orribilmente gli studenti, dato che tre non è una potenza di due e i calcolatori eseguono ogni operazione in binario. Poiché il cliente ha sempre ragione la figura è stata modificata in una cache associativa a quattro vie. Questa persona aveva effettivamente ragione? Si argomenti la risposta.
29. Molti architetti dei computer passano molto tempo cercando di rendere le pipeline più profonde. Perché?
30. Un calcolatore con una pipeline a cinque stadi tratta i salti condizionali ponendo la pipeline in stallo per tre cicli dopo aver rilevato una di queste diramazioni. Di quanto degrada le prestazioni questa strategia di mettere in stallo l'esecuzione, nel caso in cui il 20% di tutte le istruzioni siano salti condizionali? Si ignorino tutti gli altri motivi che possono mettere in stallo l'esecuzione, eccezion fatta per i salti condizionali.
31. Supponiamo che un calcolatore sia in grado di effettuare il prefetch di un massimo di 20 istruzioni. Tuttavia, in media, quattro di queste istruzioni sono salti condizionali, ciascuno dei quali può essere predetto correttamente nel 90% dei casi. Qual è la probabilità che il prefetching sia sulla strada giusta?
32. Supponiamo di voler cambiare la progettazione della macchina usata nella Figura 4.43, in modo da avere 16 registri al posto di 8. Modifichiamo quindi l'istruzione 6 in modo che usi R8 come propria destinazione. Che cosa succede nei cicli a partire dal ciclo 6?
33. Generalmente le dipendenze provocano dei problemi alle CPU con pipeline. Ci sono delle ottimizzazioni che possono essere fatte per gestire le dipendenze di tipo VWAR e che possono migliorare realmente la situazione? Quali?
34. Si riscriva l'interprete di Mic-1, facendo però in modo che LV punti alla prima variabile locale invece che al puntatore di collegamento.
35. Si scriva un simulatore per una cache a corrispondenza diretta a 1 via. Il numero degli elementi e la dimensione della linea devono essere parametri della simulazione. Si facciano degli esperimenti con il simulatore e si commentino i dati ricavati.



CAPITOLO 5

Livello di architettura dell'insieme d'istruzioni

Questo capitolo tratta in dettaglio il livello di architettura dell'insieme d'istruzioni (ISA, *Instruction Set Architecture*). Tale livello si trova tra quello della microarchitettura e il sistema operativo, come raffigurato nella Figura 1.2. Storicamente, il livello ISA fu il primo a essere sviluppato, e costituiva infatti l'unico livello presente. Ancor oggi si usa riferirsi a questo livello come all'"architettura" di una macchina o altre volte (in modo improprio) come al "linguaggio assemblativo".

Il livello ISA ha una rilevanza particolare che lo rende importante per i progettisti di sistemi: costituisce l'interfaccia tra il software e l'hardware. Anche se, in linea di principio, sarebbe possibile disporre di un hardware che esegue direttamente programmi scritti in C, C++, Java o in altri linguaggi d'alto livello, non si tratta di una buona idea. Così facendo si perderebbe l'incremento di prestazioni garantito dalla compilazione rispetto all'interpretazione. Inoltre, per ragioni di praticità, è auspicabile che i computer siano capaci di eseguire codice scritto in più linguaggi, invece che in uno solo.

L'approccio prediletto dalla quasi totalità dei progettisti di sistemi è di partire da vari linguaggi d'alto livello per poi tradurli in una forma intermedia comune, il livello ISA, e quindi costruire l'hardware in grado di eseguire direttamente i programmi di livello ISA. Tale livello definisce l'interfaccia tra i compilatori e l'hardware ed è il linguaggio che entrambi possono comprendere. Le relazioni intercorrenti tra compilatori, livello ISA e hardware sono mostrate nella Figura 5.1.

In linea di principio, la fase di progettazione di una nuova macchina richiede la consultazione sia dei progettisti del compilatore, sia dei progettisti dell'hardware, al fine di individuare le caratteristiche desiderate per il livello ISA.

Se gli autori del compilatore richiedono alcune caratteristiche che gli ingegneri non possono implementare con costi contenuti (per esempio un'istruzione *branch-and-do-payroll*, "salta e contabilizza"), allora queste vengono escluse dal progetto. Allo stesso modo se i responsabili dell'hardware ideano una qualche nuova caratteristica ingegnosa che pretendono di implementare (per esempio una memoria in cui è velocissimo acce-

dere alle locazioni il cui indirizzo è un numero primo), ma gli addetti al software non riescono a capire come scrivere del codice che la possa usare, la proposta rimarrà sulla carta. Dopo molte trattative e simulazioni emergerà e verrà infine implementato un livello ISA ottimizzato alla perfezione per il linguaggio di programmazione richiesto.

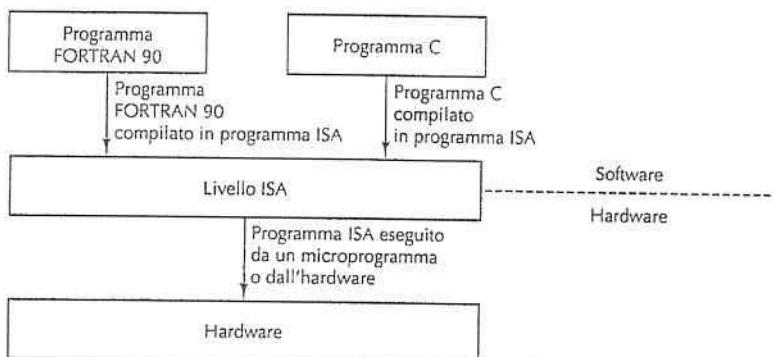


Figura 5.1 Il livello ISA è l'interfaccia tra compilatori e hardware.

Almeno in teoria. Nella bieca realtà, quando si tratta di sviluppare una nuova macchina la prima domanda posta dai potenziali utenti è: "il progetto è compatibile con il suo predecessore?". La seconda è: "posso farci girare il mio vecchio sistema operativo?". La terza è: "riuscirà a eseguire tutte le mie applicazioni senza bisogno di modificarle?". Qualora la risposta a una qualsiasi di queste domande fosse "no", i progettisti avrebbero dei seri problemi a giustificare il proprio lavoro. I clienti sono poco inclini a gettar via il loro vecchio software e ricominciare a scriverlo da capo.

Questo atteggiamento costringe gli architetti di computer a mantenere l'ISA costante di modello in modello, o a renderlo almeno retrocompatibile. Con ciò intendiamo che la nuova macchina debba essere in grado di eseguire i vecchi programmi senza modifiche. D'altra parte è del tutto comprensibile che una nuova macchina metta a disposizione alcune istruzioni e caratteristiche innovative che possono essere sfruttate solo dal software nuovo. Con riferimento alla Figura 5.1, i progettisti sono liberi di fare ciò che vogliono a livello hardware a patto che garantiscono un ISA retrocompatibile con i modelli precedenti; il livello hardware non interessa quasi a nessuno (e quasi nessuno sa come funziona). Così è possibile optare per un progetto di microprogrammazione o per l'esecuzione diretta, o aggiungere pipeline, funzionalità superscalari o qualunque altra cosa si desideri, ammesso che sia mantenuta la retrocompatibilità con l'ISA precedente. L'obiettivo è assicurare che i vecchi programmi girino sul nuovo processore e la sfida diventa la progettazione di macchine migliori soggette ai vincoli di retrocompatibilità.

Questo non vuol dire che la progettazione ISA conti poco; un buon ISA presenta vantaggi significativi rispetto a un cattivo progetto, specie in termini di potenza grezza di calcolo a parità di costi. Progetti ISA altrimenti equivalenti possono differire anche

del 25% in termini di prestazioni. Intendiamo affermare semplicemente che il mercato rende difficile (se non impossibile) l'abbandono di un ISA datato per introdurne uno nuovo. Ciononostante, di tanto in tanto emerge un nuovo ISA per uso generale, il che capita molto più spesso nei settori di mercato specializzati (per esempio nella progettazione di sistemi integrati o di processori multimediali). Per queste ragioni è importante comprendere la progettazione ISA.

Che cosa rende un ISA un "buon ISA"? Ci sono due fattori principali. Innanzitutto un buon ISA dovrebbe definire un insieme d'istruzioni che può essere implementato efficientemente da tecnologie presenti e future, il che risulta in progetti duraturi e dunque economicamente vantaggiosi. Una cattiva progettazione è più difficile da implementare e potrebbe richiedere molte più porte logiche per realizzare il processore, nonché più memoria per eseguirne i programmi. Potrebbe inoltre rallentare l'esecuzione perché offuscherebbe la possibilità di sovrapporre l'esecuzione di alcune istruzioni, richiedendo espedienti molto sofisticati per raggiungere prestazioni equivalenti. Un progetto che trae vantaggio dalle peculiarità di una particolare tecnologia può rivelarsi un fuoco di paglia e fornire un'esperienza d'implementazioni economicamente vantaggiose, per poi essere sorpassato da ISA più lungimiranti.

In secondo luogo, un buon ISA dovrebbe favorire una compilazione del codice "pulita". La regolarità e la completezza del ventaglio di scelte disponibili per il compilatore costituiscono un tratto importante ché non sempre gli ISA rispettano. Si tratta di proprietà importanti per il compilatore, che altrimenti potrebbe trovarsi in difficoltà nell'operare la scelta migliore tra alternative limitate, in special modo quando alcune scelte apparentemente ovvie non sono permesse dall'ISA. In breve, dato che l'ISA è l'interfaccia tra hardware e software, dovrebbe soddisfare sia i progettisti hardware (essere facile da implementare efficientemente), sia i progettisti software (essere favorevole alla produzione di codice di qualità).

5.1 Panoramica del livello ISA

Per cominciare lo studio del livello ISA chiediamoci che cos'è. Potrebbe sembrare una domanda semplice a cui rispondere, ma solleva più complicazioni di quanto si potrebbe immaginare a prima vista. Qui di seguito trattiamo alcune questioni controverse, dopodiché daremo uno sguardo a modelli di memoria, registri e istruzioni.

5.1.1 Proprietà del livello ISA

In linea di principio, il livello ISA si può definire come l'aspetto che la macchina assume agli occhi di un programmatore in linguaggio macchina. Siccome oramai nessun programmatore (sensato) programma più in linguaggio macchina, ridefiniamo il concetto dicendo che il codice di livello ISA è l'output di un compilatore (senza tener conto, per il momento, delle chiamate di sistema operativo e del linguaggio assemblativo simbolico). Al fine di produrre codice di livello ISA, il progettista del compilatore deve conoscere il modello di memoria, quali registri ci sono, quali sono i tipi di dati e d'istruzioni disponibili, e così via. L'insieme di tutte queste informazioni definisce il livello ISA.

Secondo questa definizione, il fatto che la microarchitettura sia o meno microprogrammata (o che disponga di pipeline, o che sia superscalare, e così via) non fa parte del livello ISA, perché non è visibile al progettista del compilatore. In realtà questa osservazione non è del tutto corretta, poiché alcune di queste proprietà influenzano le prestazioni, il che è visibile a chi scrive il compilatore. Si consideri, per esempio, un progetto superscalare che prevede l'emissione di due istruzioni in successione immediata all'interno dello stesso ciclo se un'istruzione è di tipo intero e l'altra in virgola mobile. Se il compilatore alternasse istruzioni intere a istruzioni in virgola mobile otterebbe prestazioni visibilmente migliori che non in caso contrario. Così i dettagli dell'operazione superscalare *solo* sono visibili a livello ISA, e quindi la separazione tra i livelli non è così chiara come potrebbe apparire inizialmente.

Per alcune architetture, ma non sempre, il livello ISA è specificato attraverso un documento formale di definizione, spesso redatto da un consorzio di aziende. Per esempio ARM v7 (la versione 7 di ARM) ha una definizione ufficiale pubblicata da ARM Ltd. Lo scopo di un documento di definizione è di mettere i diversi produttori in grado di costruire macchine capaci di eseguire lo stesso codice, la cui esecuzione dia esattamente gli stessi risultati.

Nel caso di ARM, l'idea è di consentire ai diversi fornitori di chip di produrre chip ARM che funzionino tutti in modo identico, differendo solo per costi e prestazioni. Perché l'idea funzioni, è necessario che i produttori di chip conoscano il funzionamento di un chip ARM (a livello ISA). Di conseguenza il documento di definizione specifica il modello di memoria, i registri presenti, il comportamento delle istruzioni e così via, ma non la microarchitettura.

Tali documenti di definizione contengono sezioni **normative**, che impongono alcuni requisiti, e sezioni **informative**, concepite per aiutare il lettore nella comprensione, ma che non fanno parte della definizione formale. Le sezioni normative usano frequentemente locuzioni quali *deve*, *non deve* e *dovrebbe* rispettivamente per richiedere, proibire e suggerire aspetti dell'architettura. Per esempio, una frase tipo

L'esecuzione di un codice operativo riservato deve causare una trap

asserisce che se il programma esegue un codice operativo che non è definito, deve provocare il sollevamento di una trap e non limitarsi a ignorare l'avvenimento. Un approccio alternativo potrebbe lasciare la scelta libera, nel qual caso la frase diventerebbe

L'effetto dell'esecuzione di un codice operativo riservato è definito dall'implementazione.

Ciò significa che chi scrive il compilatore non può contare su nessun comportamento prestabilito; si lascia a ciascun produttore la libertà di fare la propria scelta. La maggior parte delle specifiche architettoniche è corredata da risultati di test sperimentali che verificano la reale conformità di un'implementazione alla specifica corrispondente.

È evidente che la ragione per cui l'ARM v7 ha un documento che ne definisce il livello ISA è fare in modo che tutti i chip ARM possano eseguire lo stesso codice. Per diversi anni non è esistito nessun documento formale di definizione per l'ISA IA-32 (chiamato anche ISA x86), perché Intel non aveva interesse ad agevolare altri produttori a costruire chip compatibili ai suoi. Anzi, Intel è ricorsa alla giustizia nel tentativo di

impedire agli altri produttori di clonare i suoi chip, ma ha perso la causa. Tuttavia, verso la fine degli anni '90, Intel rilasciò le specifiche complete del set di istruzioni IA-32, forse perché sentì di aver sbagliato strada e volle aiutare progettisti e programmati, oppure perché gli Stati Uniti, il Giappone e l'Europa stavano accusandola di possibili violazioni delle leggi antitrust. Il riferimento per l'ISA, ben scritto, viene continuamente aggiornato ed è disponibile sul sito web per gli sviluppatori Intel (<http://developer.intel.com>). La versione rilasciata con il Core i7 conta 4161 pagine, a ricordarci ancora una volta che il Core i7 è un computer con un insieme delle istruzioni *complesso*.

Un'altra proprietà importante del livello ISA è che la maggior parte dei processori è dotata di almeno due modalità d'esecuzione. La modalità kernel serve a eseguire il sistema operativo e permette l'esecuzione di tutte le istruzioni. La modalità utente ha lo scopo di eseguire i programmi applicativi e non consente l'esecuzione di certe istruzioni "delicate" (come quelle che manipolano direttamente la cache). All'interno di questo capitolo focalizziamo l'attenzione in prevalenza sulle istruzioni in modalità utente e sulle loro proprietà.

5.1.2 Modelli di memoria

Tutti i computer suddividono la memoria in celle indirizzate in modo consecutivo. Al momento la dimensione più comune delle celle è di 8 bit, ma in passato sono state usate celle di dimensioni diverse, da 1 a 60 bit (Figura 2.10). Una cella di 8 bit si chiama byte (o otetto). La ragione per prediligere i byte è che i caratteri nella tabella ASCII occupano 7 bit, così che un carattere ASCII più un bit di parità (raramente usato) riempiono esattamente un byte. Altre codifiche, come Unicode e UTF-8, usano multipli di 8 bit per rappresentare i caratteri.

In genere i byte vengono raggruppati in parole di 4 byte (32 bit) o di 8 byte (64 bit) ed esistono istruzioni apposite per manipolare intere parole. Molte architetture esigono che le parole siano allineate lungo le loro estremità e così, per esempio, una parola di 4 byte può cominciare agli indirizzi 0, 4, 8, e così via, ma non agli indirizzi 1 o 2. Allo stesso modo una parola di 8 byte può cominciare all'indirizzo 0, 8 o 16, ma non all'indirizzo 4 o 6. L'allineamento delle parole di 8 byte è illustrato nella Figura 5.2.

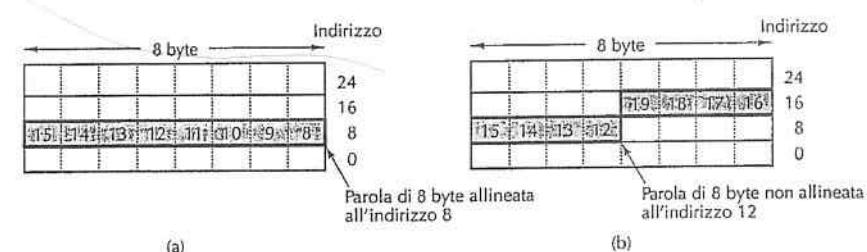


Figura 5.2 Una parola di 8 byte in una memoria little-endian (a) allineata e (b) non allineata. Alcune macchine richiedono che le parole siano allineate.

Spesso l'allineamento è richiesto perché in tal modo la memoria riesce a funzionare in modo più efficiente. Per esempio il Core i7 preleva dalla memoria 8 byte alla volta per mezzo di un'interfaccia DDR3 che supporta solamente accessi allineati ai 64 bit. Il Core i7 non potrebbe quindi, neanche volendo, referenziare indirizzi di memoria non allineati, perché l'interfaccia di memoria richiede indirizzi multipli di 8.

D'altra parte il requisito dell'allineamento può anche causare difficoltà. Nel Core i7 i programmi possono referenziare parole che cominciano a qualsiasi indirizzo, una proprietà ereditata dall'8088 che aveva un bus dati largo 1 byte (e perciò nessun bisogno di allineare i riferimenti alla memoria lungo multipli di 8 byte). Se un programma del Core i7 legge una parola di 4 byte dall'indirizzo 7, l'hardware deve effettuare un accesso a memoria per recuperare i byte da 0 a 7, più un secondo accesso per recuperare i byte da 8 a 15. Infine la CPU deve estrarre i 4 byte richiesti dai 16 byte letti dalla memoria e assestarli nel giusto ordine affinché formino una parola di 4 byte. Eseguire regolarmente queste operazioni non porta a velocità fulminanti.

La capacità di leggere parole che cominciano a indirizzi arbitrari richiede nel chip funzionalità logiche supplementari, il che lo rende più grande e più costoso. Gli ingegneri di progetto farebbero volentieri a meno di ciò, e imporrebbbero che ogni programma effettuasse riferimenti alla memoria allineati. Il problema è che, ogniqualvolta gli ingegneri chiedono "a chi interessa poter eseguire codice 8088 antiquato che effettua riferimenti sbagliati in memoria?", gli addetti al marketing rispondono lapidari: "ai nostri clienti".

Gran parte dei processori a livello ISA dispone di un solo spazio lineare degli indirizzi, che si estende dall'indirizzo 0 fino a un certo massimo, generalmente 2^{32} o 2^{64} byte. Esistono tuttavia macchine che dispongono di spazi degli indirizzi separati per le istruzioni e per i dati, di modo che il fetch di un'istruzione all'indirizzo 8 proviene da un diverso spazio degli indirizzi rispetto al fetch di un dato all'indirizzo 8. Questo schema è sì più complesso di quello con spazio degli indirizzi unitario, ma presenta due vantaggi. Per prima cosa è possibile referenziare 2^{32} byte di programma e 2^{32} byte di dati usando indirizzi di soli 32 bit. In secondo luogo, poiché le scritture avvengono sempre nello spazio dei dati diviene impossibile sovrascrivere il programma accidentalmente, il che elimina una possibile sorgente di bachi di programma. La separazione dello spazio delle istruzioni da quello dei dati rende inoltre più difficili gli attacchi dei malware, perché il software "maligno" non può accedere al programma (non può nemmeno indirizzarlo).

Si noti che disporre di spazi degli indirizzi separati per dati e istruzioni non è lo stesso che disporre di una cache separata di primo livello. Nel primo caso il numero totale d'indirizzi viene raddoppiato e gli accessi agli indirizzi portano a risultati differenti, a seconda che avvengano nello spazio dei dati o delle istruzioni. Nel caso della cache separata c'è un solo spazio degli indirizzi, soltanto che cache differenti ne contengono porzioni differenti.

Un ulteriore aspetto del modello di memoria a livello ISA è la semantica della memoria. È ragionevole aspettarsi che un'istruzione LOAD, eseguita dopo un'istruzione STORE e sullo stesso indirizzo, restituisca il valore appena memorizzato. Invece sappiamo dal Capitolo 4 che in molte architetture le microistruzioni sono riordinate, e questo

genera il pericolo concreto che la memoria esibisca comportamenti inattesi. La faccenda si complica ulteriormente nel caso di un multiprocessore, dove ogni CPU invia un flusso di richieste di accessi in lettura e scrittura (eventualmente riordinate) alla stessa memoria condivisa.

I progettisti di sistema possono scegliere tra molti approcci risolutivi del problema. A un estremo c'è la possibilità di serializzare tutte le richieste d'accesso a memoria, così che ciascuna viene completata prima che venga emessa la successiva. Questa strategia degrada le prestazioni, ma dà luogo alla semantica di memoria più semplice in assoluto (tutte le operazioni sono eseguite esattamente nell'ordine specificato dal programma).

All'altro estremo c'è il caso in cui non si dà nessun tipo di garanzia. Per forzare un ordine sulla memoria il programma deve eseguire un'istruzione SYNC, che blocca l'emissione di nuove operazioni sulla memoria finché le precedenti non risultino completate. Questa scelta genera un grosso carico di lavoro supplementare per il compilatore, che deve conoscere il funzionamento della microarchitettura in dettaglio, ma assicura ai progettisti hardware la massima libertà nell'ottimizzazione dell'utilizzo della memoria.

Si danno anche modelli di memoria intermedi, in cui l'hardware blocca automaticamente l'emissione di certi accessi a memoria (per esempio quelli che coinvolgono dipendenze RAW o WAR), ma non di altri. Nonostante l'esposizione della microarchitettura a livello ISA generi tutte queste anomalie abbastanza fastidiose (quanto meno per i programmati che scrivono il compilatore e il linguaggio assemblativo), l'abitudine è molto diffusa. Questa tendenza è dovuta alle implementazioni sottostanti, quali il ricondizionamento delle microistruzioni, le pipeline profonde, i livelli multipli di cache, e così via. Nel corso di questo capitolo incontreremo altri esempi di effetti così poco naturali.

5.1.3 Registri

Tutti i computer dispongono di qualche registro visibile a livello ISA. Il loro compito è il controllo dell'esecuzione del programma, il contenimento dei risultati temporanei o altro. In genere i registri visibili a livello microarchitetturale, quali il TOS e il MAR nella Figura 4.1, non sono visibili a livello ISA. Tuttavia alcuni di loro, come il program counter e il puntatore allo stack, sono visibili a entrambi i livelli. D'altro canto i registri visibili a livello ISA sono sempre visibili a livello della microarchitettura, perché è lì che sono implementati.

I registri del livello ISA possono essere divisi grosso modo in due categorie: registri specializzati e registri d'uso generale. I primi comprendono il program counter, il puntatore allo stack e altri registri dedicati a funzioni specifiche. I registri d'uso generale sono destinati invece a contenere le variabili locali più importanti e i risultati parziali del calcolo. La loro funzione principale è di consentire un accesso rapido a dati usati ricorrentemente (in pratica per evitare accessi in memoria). Le macchine RISC, dotate di CPU veloci e memorie relativamente lente, hanno in genere almeno 32 registri d'uso generale, ma nei nuovi progetti la tendenza è di incrementarne il numero.

I registri d'uso generale di alcune macchine sono del tutto simmetrici e intercambiabili. Ognuno può fare esattamente le cose che gli altri possono fare. Se i registri sono tutti equivalenti il compilatore può scegliere indifferentemente se usare R1 o R25 per mantenere un risultato temporaneo: la scelta del registro non ha importanza.

I registri d'uso generale di altre macchine possono invece essere in qualche modo specializzati. Per esempio, il Core i7 ha un registro chiamato EDX utilizzabile come registro d'uso generale, ma che è anche destinato a ricevere la metà del prodotto in una moltiplicazione e la metà del dividendo in una divisione.

Anche quando i registri d'uso generale sono completamente intercambiabili, è usuale che i sistemi operativi o i compilatori adottino convenzioni nel modo di utilizzarli. Per esempio alcuni registri possono contenere i parametri di chiamata a una procedura e altri essere usati come registri di lavoro. Se un compilatore memorizza una variabile locale importante in R1 e poi richiama una procedura di libreria che considera R1 un registro di lavoro, al termine dell'esecuzione della procedura R1 potrebbe contenere della spazzatura. Laddove esistono convenzioni su come vadano usati i registri di un sistema, è consigliabile che i compilatori e i programmatori del linguaggio assemblativo le rispettino... per evitare problemi.

Oltre ai registri del livello ISA visibili ai programmi dell'utente, esiste un numero sostanziale di registri specializzati visibile solo in modalità kernel e che controlla cache, memoria, dispositivi di I/O e altre funzionalità hardware della macchina. Possono essere impiegati solo dal sistema operativo, perciò i compilatori e gli utenti non hanno bisogno di riconoscerli.

Il registro di flag, detto anche PSW (*Program Status Word*), è una specie di ibrido tra la modalità kernel e quella utente. Questo registro di controllo contiene vari bit di natura eterogenea che sono necessari alla CPU, tra cui i più importanti sono i codici di condizione, che vengono impostati a ogni ciclo dell'ALU e riflettono lo stato del risultato dell'operazione più recente. Alcuni bit tipici che rappresentano codici di condizione sono:

- N – posto a 1 dopo risultato negativo;
- Z – posto a 1 dopo risultato uguale a zero;
- V – posto a 1 se il risultato ha causato un overflow;
- C – posto a 1 se il risultato ha causato un riporto oltre l'ultimo bit più significativo;
- A – posto a 1 se si è verificato un riporto oltre il terzo bit (riporto ausiliario, si veda di seguito);
- P – posto a 1 se il risultato è pari (parità nulla).

I codici di condizione sono importanti perché sono utilizzati dalle istruzioni di confronto e di salto condizionato. Per esempio, l'istruzione CMP sottrae due operandi e imposta il codice di condizione in base alla differenza. Se gli operandi sono uguali, la differenza è zero e il bit di codice di condizione Z viene posto a 1. Una successiva istruzione BEQ (*branch on equal*, “salta se uguali”) controlla il bit Z ed effettua il salto se questo ha valore 1.

Il PSW non contiene soltanto codici di condizione, ma il resto del contenuto varia da macchina a macchina. Alcuni campi addizionali molto comuni sono la modalità di macchina (cioè, utente o kernel), i bit di traccia (usati nel debugging), il livello di priorità della CPU e lo stato di attivazione degli interrupt. Spesso il PSW è leggibile in modalità utente, ma alcuni dei suoi campi possono essere scritti solo in modalità kernel (per esempio il bit di modalità kernel/utente).

5.1.4 Istruzioni

La caratteristica principale del livello ISA è l'insieme d'istruzioni macchina che definisce, che specifica ciò che la macchina è in grado di fare. Comprende sempre le istruzioni STORE e LOAD (in forme diverse) finalizzate al trasferimento di dati dai registri alla memoria e viceversa, nonché l'istruzione MOVE per la copia di dati tra registri. Sono sempre presenti le istruzioni aritmetiche, le istruzioni booleane e quelle di confronto dei dati con eventuale salto condizionato dal risultato del confronto. Abbiamo già incontrato alcune istruzioni ISA comuni (vedi la Figura 4.11) e ne incontreremo molte di più in questo capitolo.

5.1.5 Panoramica del livello ISA del Core i7

In questo capitolo analizziamo estesamente tre ISA molto differenti tra loro: IA-32 di Intel, incorporato nel Core i7, la versione 7 dell'architettura ARM, implementata nel SoC OMAP4430, e l'architettura AVR a 8 bit, utilizzata dal microcontrollore ATmega168. L'intento non è quello di fornire una descrizione esaustiva di questi ISA, ma piuttosto di mostrare gli aspetti importanti dell'ISA in generale e la loro variabilità da un particolare ISA all'altro. Cominciamo con il Core i7.

Il Core i7 è frutto dell'evoluzione di molte generazioni, e risalendo il suo albero genealogico possiamo raggiungere alcuni tra i primi processori mai costruiti, come già trattato nel Capitolo 1. Il suo ISA, oltre a mantenere completo supporto per l'esecuzione di programmi scritti per l'8086 e per l'8088 (che avevano lo stesso ISA), contiene addirittura rimembranze dell'8080, un processore a 8 bit diffusissimo negli anni '70. L'8080 fu a sua volta influenzato da vincoli di compatibilità dell'ancora precedente 8008, basato sul 4004, un chip a 4 bit in uso ai tempi in cui i dinosauri vagavano sulla Terra.

Dal punto di vista del software, sia l'8086 sia l'8088 erano macchine a 16 bit effettivi (anche se l'8088 aveva un bus dati a 8 bit). Anche il loro successore, l'80286, era una macchina a 16 bit, e presentava come principale miglioria un maggiore spazio degli indirizzi, sebbene pochi programmi se ne siano mai avvantaggiati, visto che era composto da 16.384 segmenti di 64 KB, invece che da una memoria lineare di 2^{30} byte.

L'80386 fu la prima macchina a 32 bit della famiglia Intel. Tutti i processori successivi (80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, Celeron, Xeon, Pentium M, Centrino, Core 2 duo, Core i7 e così via) hanno essenzialmente la stessa architettura dell'80386, nota come IA-32, perciò focalizziamo la nostra attenzione su questa architettura. Gli unici cambiamenti architettonici significativi introdotti dopo l'80386 sono stati le istruzioni MMX, SSE e SSE2 nelle versioni più recenti. Si tratta d'istruzioni altamente specializzate e progettate per incrementare le prestazioni di applicazioni multimediali. Un'altra importante estensione è stata la x-86 a 64 bit (nota come x86-64), che porta la dimensione dei calcoli interi e degli indirizzi virtuali a 64 bit. Mentre molte estensioni sono state introdotte da Intel e poi implementate dalla concorrenza, in questo caso il primato spetta ad AMD.

Il Core i7 è dotato di tre modalità operative, due delle quali lo fanno funzionare come un 8088. Nella modalità reale tutte le caratteristiche introdotte dopo l'8088 sono disattivate e il Core i7 si comporta come un semplice 8088. Se un programma fa qualcosa di

sbagliato, la macchina si blocca. Se l'Intel avesse progettato gli esseri umani, li avrebbe fatti con un bit per ritornare alla modalità scimpanzé (con gran parte del cervello disattivata, privi di parola, con dimora sugli alberi, predilezione per le banane, e così via).

A un gradino più in alto c'è la modalità virtuale 8086, che consente di eseguire vecchi programmi dell'8088 in modo protetto. In questa modalità la macchina è controllata da un vero e proprio sistema operativo. Per eseguire un programma 8088, il sistema operativo crea un ambiente isolato speciale che si comporta come un 8088, con la sola differenza che, se il programma si blocca, la macchina non si arresta, ma passa il controllo al sistema operativo. Quando un utente di Windows apre una finestra MS-DOS, il programma che la esegue gira in modalità virtuale 8086 per proteggere lo stesso Windows da eventuali comportamenti errati dei programmi MS-DOS.

Infine abbiamo la modalità protetta, in cui il Core i7 si comporta come un Pentium 4 e non come un 8088 molto costoso. Sono previsti quattro livelli di privilegi, controllati da bit di PSW. Il livello 0, usato dal sistema operativo, corrisponde alla modalità kernel degli altri computer e ha accesso completo alla macchina. Il livello 3, usato per i programmi utente, blocca l'accesso a certe istruzioni critiche e controlla i registri per impedire a un programma utente malintenzionato di mandare in panne l'intera macchina. I livelli 1 e 2 sono usati raramente.

Il Core i7 dispone di uno spazio degli indirizzi enorme: la memoria è divisa in 16.384 segmenti, ciascuno dei quali va dall'indirizzo 0 all'indirizzo $2^{32} - 1$. Tuttavia la maggior parte dei sistemi operativi (compreso UNIX e tutte le versioni di Windows) supporta un solo segmento, così che la maggior parte delle applicazioni vede a tutti gli effetti solo uno spazio degli indirizzi lineare di 2^{32} byte, e talvolta parte di questo spazio è occupata dal sistema operativo. Ogni byte dello spazio degli indirizzi ha un proprio indirizzo, e le parole sono lunghe 32 bit. Le parole sono memorizzate in formato little-endian (il byte meno significativo ha indirizzo più basso).

I registri del Core i7 sono mostrati nella Figura 5.3. I primi quattro registri, EAX, EBX, ECX ed EDX, sono registri a 32 bit di uso più o meno generale, anche se ciascuno ha le sue peculiarità. EAX è il registro aritmetico principale; EBX è usato per contenere puntatori (a indirizzi di memoria); ECX è usato nei cicli; EDX è necessario per le moltiplicazioni e per le divisioni durante le quali, insieme a EAX, contiene prodotti e dividendi di 64 bit. Tutti questi registri sono utilizzabili come registri di 16 bit (nei 16 bit meno significativi) o di 8 bit (negli 8 bit meno significativi) e quindi agevolano la manipolazione di quantità di 16 e 8 bit rispettivamente. I registri a 32 bit furono introdotti con l'80386, come anche il prefisso E, che sta per Esteso.

Anche i tre registri successivi sono in un certo senso generali, ma con maggiori peculiarità. I registri ESI ed EDI servono a contenere puntatori alla memoria, in special modo per le istruzioni hardware di manipolazione di stringhe, dove ESI punta alla stringa sorgente ed EDI alla stringa destinazione. Anche il registro EBP è un puntatore ed è usato generalmente per referenziare l'indirizzo base del record d'attivazione corrente, analogamente a LV nella IJVM. Quando un registro (come EBP) è usato per puntare all'indirizzo base del record d'attivazione locale viene detto per l'appunto **puntatore al record d'attivazione** (*frame pointer*). Infine ESP è il puntatore allo stack.

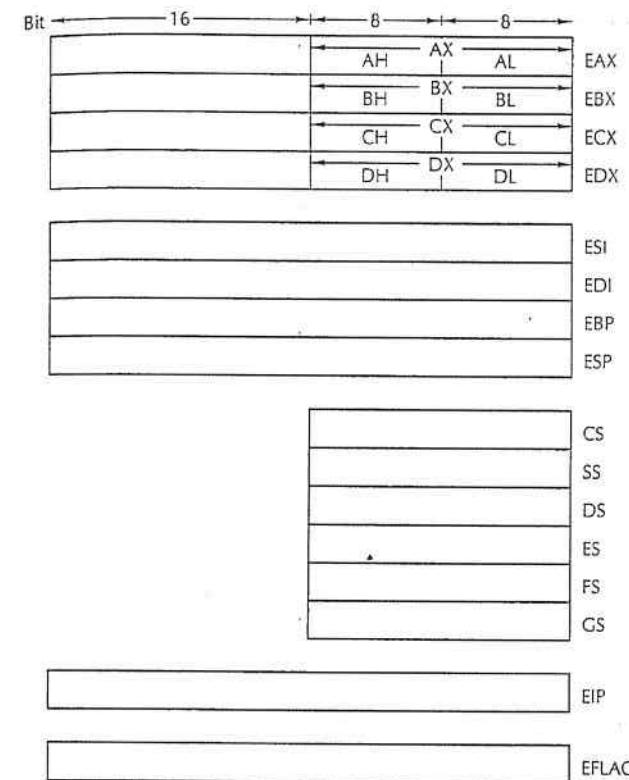


Figura 5.3 Registri principali di Core i7.

Il gruppo successivo di registri, che vanno da CS a GS, è costituito da registri di segmento. Si può dire che sono trilobiti elettronici, antichi fossili giunti fino a noi dall'epoca in cui l'8088 cercava d'indirizzare 2^{20} byte di memoria usando indirizzi da 16 bit. Basti sapere che possono essere tranquillamente ignorati quando il Core i7 è impostato per usare un solo spazio degli indirizzi lineare a 32 bit. Poi c'è l'EIP (Extended Instruction Pointer), che è il program counter. Infine troviamo EFLAGS, che è analogo a PSW.

5.1.6 Panoramica del livello ISA dell'OMAP4430 ARM

L'architettura ARM venne introdotta per la prima volta da Acorn Computer nel 1985 ed era ispirata dall'attività di ricerca svolta a Berkeley negli anni '80 (Patterson, 1985; Patterson e Séquin, 1982). L'architettura ARM originale (chiamata ARM2) era a 32 bit e supportava uno spazio degli indirizzi a 26 bit. L'OMAP4430 utilizza la microarchitettura ARM Cortex A9 che implementa la versione 7 dell'architettura ARM, oggetto di

studio in questo capitolo. Per coerenza con il resto del libro parleremo dell'OMAP4430, anche se a livello ISA tutti i progetti basati su ARM Cortex A9 si corrispondono.

La struttura della memoria dell'OMAP4430 è chiara e semplice: la memoria indirizzabile è un vettore di 2^{32} byte. I processori ARM sono bi-endian, in modo da poter accedere alla memoria nei due ordini big-endian e little-endian. L'ordine è specificato in un blocco di memoria di sistema che viene letto immediatamente dopo il reset del processore. Per assicurare una lettura corretta questo blocco deve essere in formato little-endian anche se la macchina va configurata per operare in big-endian.

È importante che l'ISA preveda una limitazione dello spazio degli indirizzi più grande delle necessità implementative, perché in futuro quasi certamente sarà necessario incrementare la dimensione della memoria accessibile dal processore. Lo spazio di indirizzamento a 32 bit dell'ISA ARM sta dando diverse preoccupazioni ai progettisti, perché molti sistemi basati su ARM, come gli smartphone, hanno già più di 2^{32} byte di memoria. Fino a oggi i progettisti hanno aggirato l'ostacolo utilizzando unità di memoria flash a cui si accede tramite un'interfaccia disco che supporta uno spazio di indirizzamento di dimensioni maggiori orientato ai blocchi. Per dare una soluzione a questo problema che potrebbe ridurre le vendite, la società ARM ha recentemente pubblicato la definizione dell'ISA ARM versione 8, con il supporto di uno spazio di indirizzamento a 64 bit.

Uno dei problemi più seri che hanno dovuto affrontare le architetture di successo è stata la limitazione alla quantità di memoria imposta dal livello ISA. Nell'informatica l'unico errore che non si può aggirare è la mancanza di bit. Un giorno i vostri nipoti vi chiederanno come potevano funzionare i computer dei vecchi tempi con indirizzi di soli 32 bit e con soli 4 GB di memoria effettiva, quando un qualsiasi videogioco avrà bisogno di almeno 1 TB solo per partire.

L'ISA ARM è elegante, anche se l'organizzazione dei registri è complicata dal tentativo di semplificare la codifica di alcune istruzioni. L'architettura mappa il *program counter* nel banco dei registri come registro R15, per permettere la creazione di salti con operazioni dell'ALU aventi R15 come registro destinazione. L'esperienza dimostra che l'organizzazione dei registri causa più problemi di quanti ne risolva, ma la vecchia regola della retrocompatibilità la rende imprescindibile.

L'ISA ARM ha due gruppi di registri: 16 d'uso generale da 32 bit e 32 in virgola mobile da 32 bit (se è supportato il coprocesso VFP). I registri d'uso generale hanno nomi che vanno da R0 fino a R15, anche se possono assumere nomi diversi a seconda del contesto. La Figura 5.4 mostra i nomi alternativi e la funzione dei vari registri.

Tutti i registri d'uso generale sono di 32 bit e possono essere letti e scritti da una varietà d'istruzioni di caricamento e memorizzazione. Gli utilizzi indicati nella Figura 5.4 sono in parte dovuti a convenzioni, ma anche giustificati dal modo in cui l'hardware tratta i registri. In generale è poco saggio deviare dalle modalità d'utilizzo elencate nella figura, a meno che non siate "cintura nera" di ARM e sappiate veramente ciò che state facendo. È responsabilità del compilatore o del programmatore assicurarsi che un programma acceda ai registri in maniera corretta e che effettui l'aritmetica appropriata al caso. Per esempio, è molto facile caricare numeri in virgola mobile nei registri d'uso

generale e poi effettuare su di loro l'addizione intera, il che produce un risultato che non ha alcun senso, ma che la CPU calcolerà senza batter ciglio se così istruita.

Registri	Nomi alternativi	Funzione
R0 – R3	A1 – A4	Contengono i parametri della procedura che viene invocata
R4 – R11	V1 – V8	Contengono le variabili locali della procedura corrente
R12	IP	Registro chiamata intraprocedura (per chiamate a 32 bit)
R13	SP	Puntatore allo stack
R14	LR	Contiene l'indirizzo di ritorno della funzione corrente
R15	PC	Program counter

Figura 5.4 Registri d'uso generale di ARM v7.

I registri Vx sono usati per memorizzare costanti, variabili e puntatori necessari alle procedure e devono essere memorizzati e ricaricati all'ingresso e all'uscita di ogni procedura, se necessario. I registri Ax sono usati per il passaggio di parametri a procedura per evitare accessi in memoria. Più avanti analizzeremo questo meccanismo in dettaglio.

Quattro registri dedicati sono usati per scopi specifici. Il registro IP serve per aggirare le limitazioni dell'istruzione ARM per le chiamate di funzioni (BL) che non può indirizzare tutti i 2^{32} byte dello spazio degli indirizzi. Se la destinazione di una chiamata è troppo lontana per poter essere espressa, l'istruzione chiamerà un frammento di codice che utilizza l'indirizzo contenuto nel registro IP come destinazione della chiamata alla funzione. Il registro SP indica la posizione corrente della cima dello stack; il suo valore viene aggiornato ogni volta che si effettuano operazioni di push e pop. Il terzo registro a uso speciale è LP, utilizzato dalle chiamate a procedura per mantenere l'indirizzo di ritorno. Come menzionato in precedenza il quarto registro a uso speciale è il PC. La scrittura di un valore in questo registro redirige il prelievo delle istruzioni al nuovo indirizzo depositato nel program counter. Un altro importante registro dell'architettura ARM è il registro di stato del programma, PSR (*Program Status Register*), che mantiene lo stato delle precedenti operazioni dell'ALU, tra cui i bit Zero, Negative e Overflow.

L'ISA ARM (nella configurazione che include il coprocesso VFP), dispone anche di 32 registri in virgola mobile di 32 bit. A questi registri si può accedere direttamente, trattandoli come 32 valori in virgola mobile a precisione singola, oppure li si può trattare come 16 valori in virgola mobile da 64 bit, a precisione doppia. La dimensione del registro in virgola mobile è determinata dall'istruzione; in generale, di tutte le istruzioni in virgola mobile sono disponibili le due varianti per lavorare in precisione singola o doppia.

L'architettura ARM è un'architettura load/store, ossia le sole operazioni che accedono direttamente alla memoria sono quelle di load e store, utili al trasferimento di dati tra i registri e la memoria. Tutti gli operandi delle istruzioni logiche e aritmetiche devono essere contenuti nei registri o nell'istruzione stessa (non in memoria), e così tutti i risultati devono essere salvati in un registro (non in memoria).

5.1.7 Panoramica del livello ISA dell'ATmega168 AVR

Il nostro terzo esempio è l'ATmega168. A differenza del Core i7 (usato prevalentemente nei desktop e nelle *server farm*) e dell'OMAP4430 (impiegato soprattutto nei telefoni, tablet e altri dispositivi mobili), l'ATmega168 è usato nei sistemi integrati, quali i semafori e le radiosveglie, per il loro controllo, la gestione dei pulsanti, delle luci e delle altre parti che costituiscono l'interfaccia utente. In questo paragrafo forniamo una breve introduzione tecnica dell'ISA AVR dell'ATmega168.

L'ATmega168 ha una sola modalità e nessuna protezione hardware, poiché non si dà mai il caso che esegua programmi di utenti potenzialmente ostili. Anche il modello della memoria è estremamente semplice: c'è uno spazio di memoria di 16 KB per i programmi e uno, distinto, di 1 KB per i dati. Ogni indirizzo fa quindi riferimento a una diversa memoria a seconda se si sta accedendo al programma o ai dati. Gli spazi di programma e dati sono separati al fine di implementare lo spazio dei programmi in una memoria flash e lo spazio dei dati in una SRAM.

L'ATmega168 fa uso di un'organizzazione di memoria a due livelli per offrire una maggior sicurezza. La memoria flash del programma è divisa nella sezione del boot loader e nella sezione delle applicazioni; le dimensioni delle sezioni sono determinate da bit che vengono programmati una volta sola al primo avvio del microcontrollore. Per ragioni di sicurezza solo il codice della sezione del boot loader può aggiornare la memoria flash. Grazie a questa funzionalità un qualsiasi codice (comprese le applicazioni scaricate di terze parti) può essere posizionato nella sezione delle applicazioni con la certezza che non possa intaccare altro codice presente nel sistema (perché il codice sarà eseguito dallo spazio delle applicazioni dal quale non è possibile scrivere sulla memoria flash). Per vincolare maggiamente il sistema, un distributore può firmare digitalmente il codice. Quando il codice è firmato, il boot loader lo caricherà nella memoria flash solo se la firma digitale proviene da un distributore di software approvato. In tal modo, il sistema eseguirà solamente il codice che è stato "benedetto" da un distributore fidato. Questo approccio è piuttosto flessibile e permette di sostituire anche il boot loader, a patto che il nuovo codice sia correttamente firmato. La stessa tecnica viene utilizzata da Apple e TiVo per assicurare che il codice in esecuzione sui propri dispositivi sia al riparo da possibili danneggiamenti.

L'ATmega168 contiene 32 registri a uso generale da 8 bit, chiamati R0 – R31, a cui le istruzioni accedono tramite un campo di 5 bit che specifica il numero del registro. Una peculiarità dei registri dell'ATmega168 è che essi sono anche presenti nello spazio di memoria. Il byte 0 dello spazio dati è equivalente al registro R0 e quando un'istruzione modifica il registro R0 e poi legge il byte 0 di memoria, trova in questa posizione il nuovo valore scritto in R0. Analogamente, il byte 1 nella memoria è R1, e così via fino al byte 31. Questa organizzazione è rappresentata nella figura 5.5.

Agli indirizzi di memoria da 32 a 95, immediatamente sopra ai 32 registri a uso generale, ci sono 64 byte riservati per l'accesso ai registri dei dispositivi di I/O, inclusi i dispositivi interni al SoC.

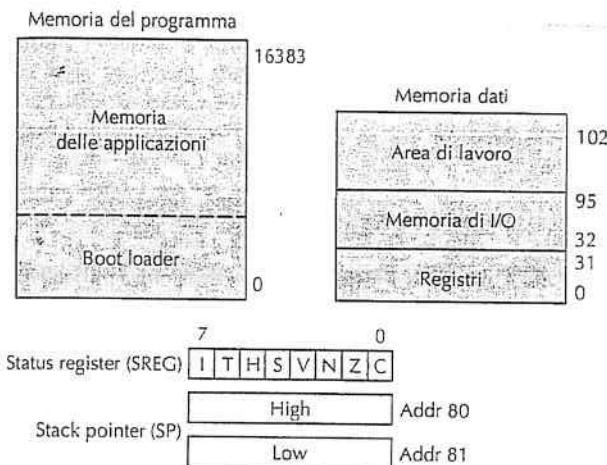


Figura 5.5 Organizzazione dei registri e della memoria sul chip dell'ATmega168.

Oltre ai quattro insiemi di otto registri ciascuno, l'ATmega168 dispone di un piccolo numero di registri specializzati, tra cui quelli più importanti sono indicati nella Figura 5.5. A partire da sinistra, il *registro di stato* contiene nell'ordine: bit di abilitazione degli interrupt, bit ausiliario di riporto, bit di segno, bit di overflow, flag negativo, flag zero, bit di riporto. Tutti questi bit di stato, a eccezione del bit di abilitazione degli interrupt, sono impostati a seguito di un'istruzione aritmetica.

Il bit I del registro di stato consente di attivare e disattivare gli interrupt globalmente. Se il bit I è posto a 0 tutti gli interrupt sono disabilitati, perciò con una sola istruzione di azzeramento di questo bit è possibile disabilitare tutti gli interrupt. Afferendo il bit si permettono eventuali interrupt pendenti e interrupt futuri. A ogni dispositivo è associato un bit di abilitazione degli interrupt. Se il dispositivo è abilitato e il bit di abilitazione globale degli interrupt è asserito, il dispositivo può interrompere il processore.

Lo stack pointer, SP, mantiene l'indirizzo corrente dello spazio dei dati a cui le istruzioni di PUSH e POP accederanno, in maniera simile alle analoghe istruzioni della JVM Java del Capitolo 4. Lo stack pointer si trova all'indirizzo 80 della memoria di I/O. Visto che un singolo byte non è sufficiente per indirizzare i 1024 byte della memoria dati, lo stack pointer è composto da due locazioni di memoria consecutive che formano un indirizzo di 16 bit.

5.2 Tipi di dati

Tutti i computer hanno bisogno di dati. In effetti lo scopo di molti sistemi di calcolo è proprio quello di elaborare dati finanziari, commerciali, scientifici, ingegneristici o di altro tipo. All'interno del computer, i dati devono essere rappresentati in una forma

specifica. A livello ISA sono disponibili una varietà di tipi di dati diversi che esamineremo in seguito.

Una delle questioni chiave è la presenza o meno di supporto hardware per un particolare tipo di dati. Supporto hardware vuol dire che una o più istruzioni si aspettano i dati in un formato particolare e l'utente non è libero di scegliere un formato differente. Per esempio i contabili hanno la singolare abitudine di scrivere i numeri negativi con il segno meno alla destra del numero invece che alla sua sinistra, dove lo mettono gli informatici. Immaginate che il responsabile del centro di calcolo di uno studio di contabilità decida di fare una buona impressione sul suo direttore modificando i numeri di tutti i computer di modo che usino il bit meno significativo (invece del più significativo) come bit di segno. Ciò produrrebbe sicuramente un grande effetto sul direttore, infatti tutto il software inizierebbe a fare degli errori assurdi. L'hardware si aspetta un certo formato d'interi e non funziona correttamente se gli si passa altro.

Considerate ora un altro studio di contabilità, questa volta incaricato dallo Stato di valutare il debito pubblico. Non basterebbe certo l'aritmetica a 32 bit perché i numeri coinvolti sono molto più grandi di 2^{32} (pari a 4 milioni circa). Una possibile soluzione potrebbe essere l'utilizzo di due interi di 32 bit per rappresentare ciascun numero, il che fa 64 bit in tutto. Se la macchina non supporta questo tipo di numeri in precisione doppia, si rende necessaria la gestione software di tutta la loro aritmetica, e in tal caso l'ordine delle due parti costituenti il numero non conta, visto che l'hardware non ne viene coinvolto. È questo un esempio di tipo di dati non supportato dall'hardware, per cui quindi non è richiesta una rappresentazione hardware. Nei prossimi paragrafi ci concentreremo sui tipi di dati supportati dall'hardware, per i quali sono quindi richiesti formati specifici.

5.2.1 Tipi di dati numerici

I tipi di dati possono essere divisi in due categorie: numerici e non. Il primo tipo di dati numerici è costituito dagli interi. Ci sono interi di lunghezze diverse, in genere di 8, 16, 32 e 64 bit. Gli interi servono a contare oggetti (per esempio il numero di cacciaviti nel magazzino di un ferramenta), per identificare oggetti (per esempio il numero di conto corrente bancario) e per molto altro. La maggior parte dei computer moderni memorizza gli interi nella notazione binaria in complemento a due, pur se in passato sono stati usati anche altri sistemi. I numeri binari sono analizzati in dettaglio nell'Appendice A.

Alcuni computer supportano sia gli interi senza segno sia quelli con segno. Nel primo caso non c'è alcun bit di segno e tutti i bit rappresentano dati. Questo tipo di dati presenta il vantaggio di disporre di un bit in più e così una parola di 32 bit può contenere un intero senza segno di valore compreso tra 0 e $2^{32} - 1$, estremi inclusi. D'altra parte un intero con segno di 32 bit rappresentato in complemento a due può gestire numeri minori o uguali a $2^{31} - 1$ ma, naturalmente, può gestire anche numeri negativi.

Per rappresentare i numeri che non possono essere espressi con gli interi (per esempio 3,5) si usano i numeri in virgola mobile, trattati nell'Appendice B. Questi sono lunghi 32, 64 o a volte 128 bit. Quasi tutti i computer dispongono d'istruzioni per svolgere aritmetica in virgola mobile, e la maggioranza di questi ha registri separati per contenere operandi interi e operandi in virgola mobile.

Alcuni linguaggi di programmazione, in particolare il COBOL, mettono a disposizione un tipo di dati per i numeri decimali. Le macchine che vogliono favorire l'uso del COBOL spesso supportano i numeri decimali nell'hardware, codificando ogni cifra decimale con 4 bit e quindi impacchettando due cifre decimali in un byte (*Binary Coded Decimal*, "decimali in codifica binaria" o BCD). Purtroppo l'aritmetica dei numeri decimali impacchettati non funziona molto bene, perciò si rendono necessarie delle istruzioni di correzione-decimale-aritmetica. Queste istruzioni hanno bisogno di conoscere il riporto oltre il terzo bit, ed è per questo che i codici di condizione spesso contengono anche un bit di riporto ausiliario. A margine di ciò, l'infame baco del millennio, detto anche Y2K (da *Year 2000*), è stato causato dalla decisione dei programmati COBOL di usare due sole cifre decimali (rappresentate con 8 bit) per rappresentare l'anno, perché più economico rispetto all'uso di quattro cifre decimali. Questione di ottimizzazione.

5.2.2 Tipi di dati non numerici

Sebbene i primi computer si siano guadagnati da vivere macinando numeri, i computer moderni sono spesso impiegati per svolgere applicazioni non numeriche, quali spedizione di email, navigazione su Internet, fotografia digitale, creazione di contenuti multimediali e loro riproduzione. Queste applicazioni richiedono altri tipi di dati spesso supportati dalle istruzioni del livello ISA. Ovviamente i caratteri sono un tipo di dati importanti in questo contesto, ma non tutti i computer ne forniscono il supporto hardware. I codici carattere più comuni sono ASCII e UNICODE, che definiscono rispettivamente caratteri di 7 e di 16 bit. Per una loro definizione dettagliata si veda il Capitolo 2.

Non è raro che il livello ISA comprenda istruzioni speciali per la gestione di stringhe di caratteri, ovvero sequenze di caratteri. Le stringhe sono spesso delimitate da un carattere speciale di fine stringa, oppure dotate di un campo lunghezza che può essere usato per ricavare la terminazione della stringa.

Anche i valori booleani sono importanti. Un valore booleano può assumere solo uno dei seguenti valori: vero o falso. In teoria un bit solo basta a rappresentare un dato booleano, associando 0 a falso e 1 a vero (o viceversa). In pratica vengono usati comunque interi byte per rappresentare questi dati, dal momento che i bit individuali non sono indirizzabili direttamente e sono perciò difficili da accedere. Comunemente si adotta la convenzione secondo cui 0 vuol dire falso e tutto il resto rappresenta il valore vero.

L'unica situazione in cui un valore booleano è rappresentato da un solo bit è all'interno di array di booleani, laddove una parola di 32 bit può contenere 32 valori binari. Una struttura di dati siffatta è detta *bit map* ("mappa di bit") e la si ritrova in molti contesti. Per esempio si può usare una bit map per tener traccia dei blocchi liberi di un disco. Se un disco ha n blocchi, la bit map è lunga n bit.

L'ultimo tipo di dati che consideriamo è il tipo puntatore, nient'altro che un indirizzo di macchina. Abbiamo già incontrato i puntatori più volte. Nelle macchine Mic-x i registri SP, PC, LV e CPP sono tutti esempi di puntatori. Un'operazione molto comune su tutte le macchine consiste nell'usare i puntatori per accedere a variabili che si trovano a una distanza prefissata da loro, che è precisamente il modo in cui funziona ILOAD. Anche se utili, i puntatori sono causa di un gran numero di errori di programmazione che spesso portano a gravi conseguenze. Per questa ragione devono essere usati con estrema cura.

5.2.3 Tipi di dati del Core i7

Il Core i7 supporta gli interi con segno in complemento a due, gli interi senza segno, i numeri decimali in codifica binaria e i numeri in virgola mobile nel formato IEEE 754 (come indicato nella Figura 5.6). A causa delle sue umili origini di macchina a 8/16 bit, il Core i7 gestisce, oltre agli interi a 32 bit, anche gli interi di quelle lunghezze, e mette a disposizione numerose istruzioni per gestire la loro aritmetica, le operazioni booleane e i confronti su di loro. Il processore può optionalmente essere utilizzato nella modalità a 64 bit che supporta registri e operazioni a 64 bit. Gli operandi non devono essere necessariamente allineati in memoria, ma se lo sono, cioè se gli indirizzi delle parole sono multipli di 4 byte, le prestazioni ne risultano incrementate.

Tipo	8 bit	16 bit	32 bit	64 bit
Interi con segno	x	x	x	x (64 bit)
Interi senza segno	x	x	x	x (64 bit)
Interi decimali in codifica binaria	x			
Numeri in virgola mobile			x	x

Figura 5.6 Tipi di dati numerici del Core i7. I tipi supportati sono indicati dalla crocetta x. I tipi contrassegnati con "64 bit" sono supportati soltanto nella modalità a 64 bit.

Il Core i7 è ben fornito anche nella manipolazione di caratteri ASCII di 8 bit: ci sono istruzioni speciali per la copia e la ricerca di stringhe di caratteri. Queste istruzioni si applicano sia alle stringhe di lunghezza nota, sia alle stringhe con carattere di fine stringa e sono usate molto di frequente dalle librerie che manipolano le stringhe.

5.2.4 Tipi di dati dell'OMAP4430 ARM

La CPU OMAP4430 supporta un ampio spettro di formati di dati, come evidenziato nella Figura 5.7. Per quanto riguarda i soli interi, può gestire operandi da 8, 16 e 32 bit, con o senza segno. La gestione dei tipi di dato piccoli dell'OMAP4430 è un po' più intelligente di quella del Core i7. Internamente, l'OMAP4430 è una macchina a 32 bit con percorso dati e istruzioni da 32 bit. Un programma può specificare dimensione e segno di un valore da caricare (per esempio, per caricare un byte con segno si usa LDRSB) e il valore viene convertito dalle istruzioni di caricamento nel corrispondente valore a 32 bit. Analogamente, anche le istruzioni di memorizzazione specificano dimensione e segno del valore da scrivere in memoria e fanno accesso solamente alla porzione specificata del registro di input.

Gli interi con segno usano il formato in complemento a due. Sono disponibili operandi in virgola mobile di 32 e 64 che rispettano lo standard IEEE 754. Tutti gli operandi devono essere allineati in memoria.

Non ci sono istruzioni hardware speciali per il supporto di tipi di dati carattere e stringa. La loro manipolazione avviene totalmente via software.

Tipo	8 bit	16 bit	32 bit	64 bit
Interi con segno	x	x	x	
Interi senza segno	x	x	x	
Interi decimali in codifica binaria				
Numeri in virgola mobile			x	x

Figura 5.7 Tipi di dati numerici della CPU ARM OMAP4430. I tipi supportati sono indicati dalla crocetta x.

5.2.5 Tipi di dati dell'ATmega168

L'ATmega168 dispone di una varietà di tipi di dati molto limitata. Tutti i registri sono lunghi 8 bit, con una sola eccezione, e così anche gli interi sono di 8 bit, così come i caratteri. In pratica il solo tipo di dati le cui operazioni aritmetiche sono realmente supportate dall'hardware è il byte di 8 bit, come si evince dalla Figura 5.8.

Per facilitare l'accesso alla memoria, l'ATmega168 include anche un supporto limitato a puntatori senza segno di 16 bit. I puntatori da 16 bit X, Y e Z sono formati dalla concatenazione delle coppie di registri di 8 bit R26/R27, R28/R29 e R30/R31 rispettivamente. Quando un'istruzione di caricamento utilizza X, Y o Z come operando per l'indirizzo, il processore può incrementare o decrementare il valore a seconda della necessità.

Tipo	8 bit	16 bit	32 bit	64 bit
Interi con segno	x			
Interi senza segno	x	x		
Interi decimali in codifica binaria				
Numeri in virgola mobile				

Figura 5.8 Tipi di dati numerici dell'ATmega168. I tipi supportati sono indicati dalla crocetta x.

5.3 Formati d'istruzione

Un'istruzione consiste in un opcode (codice operativo), di solito corredata da altre informazioni quali la provenienza degli operandi e la destinazione dei risultati. L'argomento generale che tratta della provenienza degli operandi (cioè il loro indirizzo) è detto indirizzamento e verrà illustrato in dettaglio nel prosieguo di questo paragrafo.

La Figura 5.9 mostra i diversi formati delle istruzioni di livello 2. Un'istruzione è dotata di un opcode che ne specifica il comportamento, e può contenere nessuno, uno, due o tre indirizzi.

Alcune macchine hanno tutte le istruzioni della stessa lunghezza, altre dispongono d'istruzioni di lunghezza diversa. Le istruzioni possono essere più corte, altrettanto lunghe o più lunghe della dimensione di parola. La scelta di avere tutte le istruzioni della stessa lunghezza semplifica la loro decodifica, ma spesso implica uno spreco di spazio,

visto che tutte le istruzioni devono essere lunghe quanto la più lunga. Sono anche possibili altri compromessi. La Figura 5.10 illustra alcune delle relazioni che possono intercorrere tra la lunghezza delle istruzioni e la dimensione di parola.

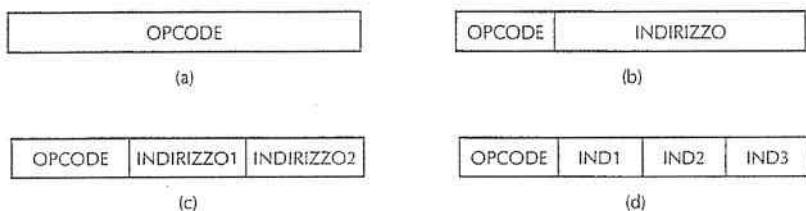


Figura 5.9 Quattro formati d'istruzioni: (a) Istruzione senza indirizzi. (b) Istruzione a un indirizzo. (c) Istruzione a due indirizzi. (d) Istruzione a tre indirizzi.

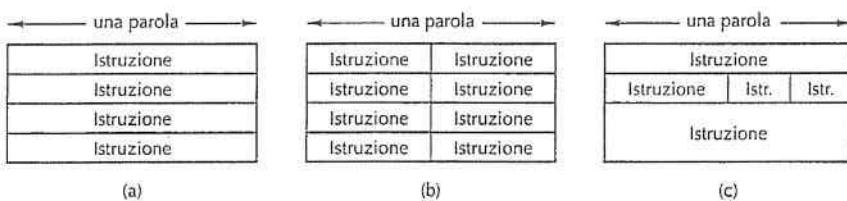


Figura 5.10 Possibili relazioni tra lunghezza delle istruzioni e dimensione di parola.

5.3.1 Criteri progettuali per i formati d'istruzioni

La scelta dei formati d'istruzione prevede che i progettisti di computer tengano conto di molti fattori. Si tratta di una decisione difficile, da non sottovalutare, e che va presa molto presto nel processo di progettazione. Se poi il computer ha successo commerciale è possibile che il suo insieme d'istruzioni sopravviva per altri quarant'anni o più. Una capacità molto preziosa è quella di saper aggiungere nuove istruzioni all'insieme per sfruttare le opportunità che si possono presentare nel corso degli anni, sempre che l'architettura, e l'azienda che l'ha progettata, sopravvivano abbastanza a lungo perché il progetto si affermi.

L'efficienza di un determinato ISA dipende fortemente dalla tecnologia impiegata nell'implementazione del computer. La tecnologia cambia molto velocemente con il passare degli anni e alcune scelte fatte per l'ISA possono rivelarsi (con il senno di vent'anni dopo) infelici. Per esempio un progetto basato sullo stack (come quello dell'IJVM) è molto valido se gli accessi in memoria sono veloci, altrimenti la strada da percorrere è quella di avere molti registri (come fa l'OMAP4430). Se pensate che sia questa una scelta facile da prendere vi invito a procurarvi un foglio e a scriverci le vostre previsioni riguardo (1) la velocità di una CPU tipica e (2) il tempo di accesso a una

comune memoria RAM reperibile nei computer tra vent'anni. Piegate il foglio, conservatelo e rileggetelo al momento opportuno. Altrimenti pubblicate oggi stesso le vostre previsioni direttamente su Internet.

Naturalmente anche i progettisti più lungimiranti a volte sbagliano. E anche se non sbagliassero, devono spesso tener in considerazione il breve termine: se il loro progetto elegante di ISA costasse anche solo poco più dello scadente progetto concorrente, l'azienda potrebbe non sopravvivere abbastanza perché il mondo riesca ad apprezzare l'eleganza del loro progetto.

A parità di progetto, le istruzioni corte sono preferibili a quelle lunghe. Un programma fatto di n istruzioni di 16 bit occupa metà spazio di memoria rispetto a un programma di n istruzioni di 32 bit. Questo fattore potrebbe risultare sempre meno importante in futuro, a giudicare dalla continua diminuzione del prezzo della memoria, se non fosse che le dimensioni del software metastizzano con una velocità superiore rispetto alla riduzione del prezzo della stessa.

Minimizzare la dimensione delle istruzioni potrebbe renderle per giunta più difficili da decodificare o da sovrapporre. In conseguenza di ciò, il criterio di ottenere istruzioni di dimensione minima deve essere valutato in base al tempo richiesto per la decodifica e l'esecuzione delle istruzioni.

Un'altra ragione importante per minimizzare la dimensione delle istruzioni, e che assumerà maggior rilievo al crescere delle prestazioni dei processori, è legata all'ampiezza (o larghezza) di banda della memoria (il numero di bit che può trasferire in un secondo). Nell'ultimo decennio abbiamo assistito a una crescita impressionante della velocità dei processori, che non è stata affiancata da un incremento altrettanto significativo dell'ampiezza di banda della memoria. E così le defezioni dei sistemi di memoria, che non sono in grado di trasferire istruzioni e operandi con la stessa celerità con cui vengono processati dalla CPU, diventano sempre più un freno per i processori. La larghezza di banda delle memorie dipende dal loro contenuto tecnologico e dalla qualità della ingegnerizzazione del progetto.

Non solo la memoria principale costituisce un collo di bottiglia, ma anche le cache: se la larghezza di banda di un'istruzione in cache è di t bps e se la lunghezza media di un'istruzione è di r bit, la cache può distribuire al massimo t/r istruzioni al secondo. Si noti che questa quantità è un *limite superiore* al tasso di prestazione del processore, anche se si stanno facendo molti sforzi di ricerca per oltrepassare questa barriera apparentemente invalicabile. Ovviamente il ritmo con cui vengono eseguite le istruzioni (cioè la velocità del processore) è legato alla lunghezza delle istruzioni: istruzioni più corte significano un processore più veloce. Dal momento che molti processori moderni sono in grado di eseguire più istruzioni nello stesso ciclo di clock, si impone la necessità di effettuare fetch multipli a ogni ciclo. Questo aspetto della cache delle istruzioni rende la dimensione delle istruzioni un criterio progettuale importante, le cui conseguenze hanno una forte influenza sulle prestazioni.

Il secondo criterio progettuale è prevedere nel formato delle istruzioni spazio sufficiente a esprimere tutte le operazioni volute. Progettare una macchina con 2^n operazioni, ognuna delle quali esprimibile con meno di n bit, è impossibile. Non ci sarebbe neanche lo spazio sufficiente nel codice operativo per indicare di che istruzione si tratti. La storia

è piena di esempi di progetti folli in cui non sono stati previsti abbastanza codici operativi in eccedenza per fronteggiare aggiunte successive all'insieme d'istruzioni.

Un terzo criterio concerne il numero di bit in un campo degli indirizzi. Si consideri il progetto di una macchina con caratteri di 8 bit e con una memoria principale sufficiente a contenere 2^{32} caratteri. I progettisti potrebbero scegliere d'indirizzare unità di 8, 16, 24 o 32 bit, ma anche di altre dimensioni.

Immaginate che cosa succederebbe se il gruppo di progettazione si dividesse in due fazioni agguerrite, una a sostegno del byte come unità base della memoria, l'altra favorevole ad avere parole di 32 bit. I primi proporrebbero una memoria di 2^{32} byte, numerati da 0 a 4.294.967.295, mentre i secondi proporrebbero una memoria di 2^{30} parole, numerate da 0 a 1.073.741.823.

Il primo gruppo porrebbe l'accento sul fatto che, nell'organizzazione con parole di 32 bit, per confrontare due caratteri un programma non solo dovrebbe fare il fetch delle intere parole contenenti i caratteri, ma dovrebbe poi estrarre i caratteri dalle parole al fine di confrontarli. Così facendo si richiederebbero istruzioni aggiuntive e si sprecherebbe quindi dello spazio. L'organizzazione a 8 bit, d'altro canto, fornirebbe un indirizzo per ogni carattere, rendendo il confronto molto più semplice.

I sostenitori della parola di 32 bit replicherebbero indicando il fatto che la loro proposta comporterebbe 2^{30} indirizzi diversi, raggiungibili con soli 30 bit invece che con i 32 richiesti dall'altra proposta per indirizzare la stessa memoria. Indirizzi più corti consentono istruzioni più corte, che occuperebbero perciò meno spazio e richiederebbero meno tempo in fase di fetch. In alternativa si potrebbe mantenere in uso gli indirizzi di 32 bit e referenziare così una memoria di 16 GB invece di miseri 4.

Questo esempio dimostra che una maggiore risoluzione nell'accesso alla memoria si paga al prezzo d'indirizzi più lunghi e conseguentemente d'istruzioni più lunghe. Il massimo di risoluzione si ha quando l'organizzazione di memoria è tale per cui ogni bit è indirizzabile direttamente (come, per esempio, nel Burroughs B1700). All'altro estremo ci sono le memorie costituite da parole molto lunghe (per esempio, la serie Cyber della CDC aveva parole di 60 bit).

I computer moderni hanno raggiunto un compromesso che, in un certo senso, racchiude il peggio di entrambe le scelte. Da un lato si usano tanti bit quanti sono necessari per indirizzare individualmente tutti i byte, dall'altra spesso si accede alla memoria per leggere una, due, o addirittura quattro parole alla volta. Tanto per fare un esempio, la lettura di un byte di memoria comporta nel Core i7 la lettura di almeno 8 byte e probabilmente di un'intera linea di cache di 64 byte.

5.3.2 Codice operativo espandibile

Nel paragrafo precedente abbiamo visto come una lunghezza degli indirizzi contenuta e una buona risoluzione di memoria si ottengano l'una a discapito dell'altra. In questo paragrafo esaminiamo altri compromessi che coinvolgono i codici operativi e gli indirizzi. Si consideri un'istruzione lunga $(n + k)$ bit, composta da un opcode di k bit e da un solo indirizzo di n bit. Un'istruzione siffatta consente 2^k operazioni diverse e permette d'indirizzare 2^n celle di memoria. In alternativa gli stessi $(n + k)$ bit potrebbero essere

spezzati in $(k - 1)$ bit di opcode e $(n + 1)$ bit d'indirizzo, dimezzando il numero d'istruzioni, ma al contempo raddoppiando la dimensione della memoria raggiungibile, oppure raggiungendo la stessa quantità di memoria ma con risoluzione doppia. Viceversa la scelta di avere opcode di $(k + 1)$ bit e indirizzi di $(n - 1)$ bit rende di più in termini di operazioni, ma al prezzo di un minor numero di celle indirizzabili, o di una peggiore risoluzione nell'accesso alla stessa quantità di memoria. Sulla falsariga degli esempi appena esposti, si possono costruire compromessi molto più sofisticati tra lunghezza degli opcode e lunghezza degli indirizzi.

Introduciamo ora il concetto di **codice operativo espandibile**, che descriviamo mediante un semplice esempio. Si consideri una macchina in cui le istruzioni sono lunghe 16 bit e gli indirizzi 4 bit, come nella Figura 5.11. Questa situazione potrebbe essere ragionevole per una macchina che ha 16 registri (dunque indirizzabili con 4 bit), sufficienti per lo svolgimento di tutte le operazioni aritmetiche. Un'architettura possibile sarebbe quella di comporre ogni istruzione con 4 bit di opcode e tre indirizzi, per un totale di 16 bit.

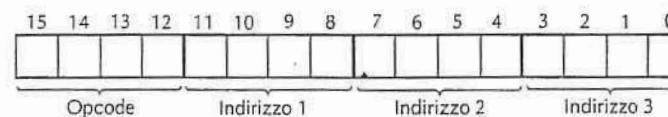


Figura 5.11 Un'istruzione composta da un opcode di 4 bit e da tre campi d'indirizzo di 4 bit.

Tuttavia, se i progettisti avessero bisogno di 15 istruzioni con tre indirizzi, 14 con due indirizzi, 31 con un indirizzo e 16 senza alcun indirizzo, potrebbero usare gli opcode da 0 a 14 per le istruzioni da tre indirizzi e interpretare l'opcode 15 diversamente (Figura 5.12).

L'opcode 15 indica che il codice operativo è contenuto nei bit da 8 a 15 invece che nei bit da 12 a 15. I bit da 0 a 3 e quelli da 4 a 7 formano ancora due indirizzi, come di consueto. Le 14 istruzioni con due indirizzi hanno tutte i 4 bit più significativi posti a 1111, mentre i bit di posto da 8 a 11 progrediscono da 0000 a 1101. Le istruzioni con bit più significativi pari a 1111 e bit da 8 a 11 pari a 1110 oppure a 1111 vengono tratte diversamente, come se il loro opcode fosse contenuto nei bit da 4 a 15. Si dispone così di 32 nuovi opcode. Poiché ne sono richiesti solo 31, l'opcode 111111111111 è interpretato come se il vero opcode fosse costituito dai bit da 0 a 15, il che fornisce altre 16 istruzioni senza indirizzi.

Nel corso della spiegazione abbiamo visto crescere la dimensione dell'opcode sempre più: da 4 bit per le istruzioni con tre indirizzi, a 8 bit per le istruzioni con due indirizzi, a 12 bit per le istruzioni con un indirizzo, per finire a 16 bit per le istruzioni senza indirizzi.

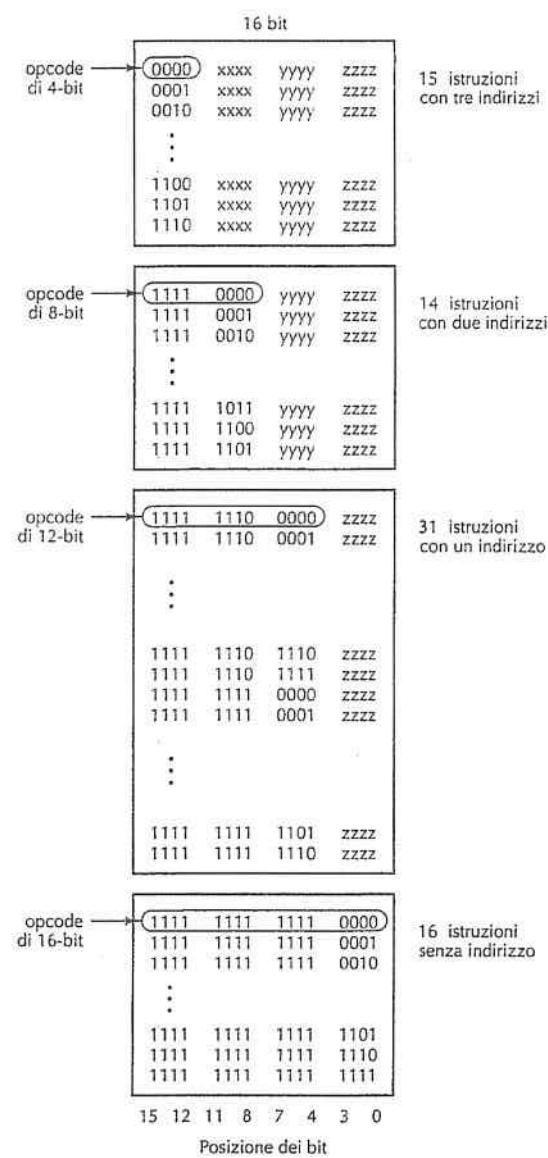


Figura 5.12 Un opcode espandibile che consente di avere 15 istruzioni con tre indirizzi, 14 con due indirizzi, 31 con un indirizzo e 16 senza indirizzo. I campi contrassegnati con xxxx, yyyy e zzzz sono di 4 bit.

L'idea di codice operativo espandibile evidenzia il compromesso che sussiste tra lo spazio riservato all'opcode e lo spazio per le altre informazioni. Nella pratica i codici operativi espandibili non sono così regolari come nel nostro esempio. La capacità di usare dimensioni di opcode variabili può venire sfruttata in due modi diversi: in primo luogo si può mantenere costante la dimensione delle istruzioni, assegnando gli opcode più corti alle istruzioni che hanno bisogno di più bit per specificare informazioni di altra natura; in secondo luogo, è possibile minimizzare la lunghezza media delle istruzioni, scegliendo gli opcode più corti possibile per le istruzioni comuni e lasciando i più lunghi per le istruzioni d'uso più raro.

Se si spinge l'idea di opcode di lunghezza variabile fino alle sue estreme conseguenze, è possibile minimizzare la lunghezza media delle istruzioni codificandone ciascuna con il minimo numero di bit necessari. Sfortunatamente ciò comporterebbe istruzioni di lunghezze diverse e non allineate nemmeno al byte. Sebbene siano esistiti ISA con questa proprietà (per esempio lo sventurato Intel 432), l'importanza dell'allineamento è così rilevante per la decodifica rapida delle istruzioni che un tale grado di ottimizzazione è quasi certamente controproducente. Nondimeno viene spesso impiegato a livello dei byte.

5.3.3 Formati delle istruzioni del Core i7

I formati delle istruzioni del Core i7 sono molto complessi e irregolari, con fino a sei campi di lunghezza variabile, cinque dei quali sono opzionali. Il loro schema generale è illustrato nella Figura 5.13. Lo stato delle cose è dovuto al fatto che l'architettura si è evoluta per molte generazioni e fin dagli inizi ha incluso alcune scelte inadeguate che non sono più state messe in discussione, nel nome della retrocompatibilità. Come regola generale per le istruzioni con due operandi, se un operando si trova in memoria, l'altro non vi si può trovare, perciò esistono istruzioni per fare la somma di due registri, di un registro e una locazione di memoria, ma non di due locazioni di memoria.

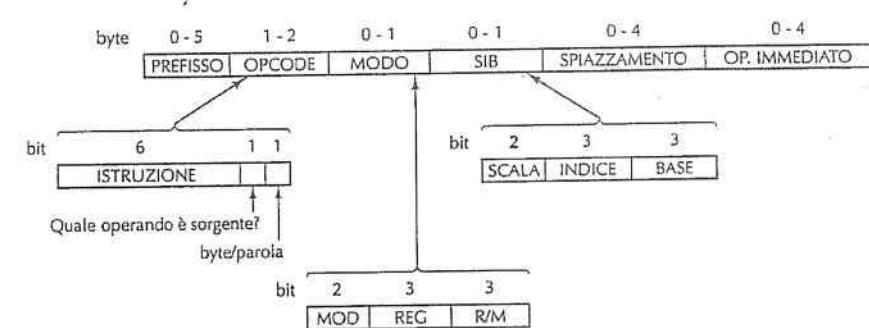


Figura 5.13 Format d'istruzione di Core i7.

Nelle prime architetture Intel tutti i codici operativi erano lunghi un byte, sebbene venisse usato largamente il concetto di byte prefisso per modificare alcune istruzioni. Un byte prefisso è un codice operativo supplementare preposto all'inizio di un'istruzione per cambiarne il comportamento. L'istruzione WIDE dell'IJVM è un esempio di byte prefisso. A un certo stadio dell'evoluzione dei suoi processori, Intel esaurì malauguratamente gli opcode disponibili, così l'opcode 0xFF fu scelto per designare un codice di escape, a indicare la presenza di un secondo byte per la specifica dell'istruzione.

I singoli bit degli opcode del Core i7 danno ben poche informazioni riguardo le istruzioni. L'unica struttura riconoscibile nel campo opcode di alcune istruzioni è l'uso del bit meno significativo per distinguere tra istruzioni di un byte o di una parola, e l'uso del bit adiacente per stabilire se l'eventuale indirizzo di memoria è da considerarsi come sorgente o come destinazione dell'operazione. Perciò l'opcode va interamente decodificato per determinare quale classe di operazioni eseguire e, di conseguenza, quanto è lunga l'istruzione corrispondente. Questo si traduce nella difficoltà di ottenere implementazioni con prestazioni elevate, dal momento che si richiede una decodifica impegnativa prima di poter stabilire dove comincia l'istruzione successiva.

In gran parte delle istruzioni che prevedono un operando in memoria, dopo il byte di opcode si trova un byte (di modo) contenente tutte le informazioni relative all'operando; è composto da un campo MOD da 2 bit e da due campi di 3 bit, REG e R/M. Talvolta i primi 3 bit di questo byte sono usati come estensione dell'opcode, dando vita a un opcode di 11 bit in totale. A ogni modo, il campo MOD di due bit consente solo quattro modalità d'indirizzamento degli operandi e almeno un operando è vincolato a trovarsi nei registri; secondo logica ciascuno dei registri EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP dovrebbe poter fungere da operando, ma le regole di codifica proibiscono alcune combinazioni e le riservano a casi particolari. Alcune modalità richiedono un bit supplementare, detto SIB (da *Scala, Indice e Base*), che fornisce un'ulteriore specificazione. Nel complesso questo schema non è certo ideale, bensì un compromesso tra esigenze contrastanti, da un lato la retrocompatibilità e dall'altro la voglia di aggiungere nuove funzionalità originariamente non previste.

Inoltre, alcune istruzioni sono dotate di 1, 2 o 4 byte addizionali che specificano l'indirizzo di memoria (lo spiazzamento) e addirittura di altri 1, 2 o 4 byte atti a contenere una costante (operando immediato).

5.3.4 Formati delle istruzioni dell'OMAP4430 ARM

L'ISA dell'OMAP4430 comprende istruzioni da 16 e 32 bit allineate in memoria. In genere le istruzioni sono molto semplici e ciascuna specifica una singola azione. Una tipica istruzione aritmetica specifica due indirizzi che forniscono gli operandi sorgente e un solo registro come destinazione. Le istruzioni a 16 bit sono versioni più snelle di quelle a 32 bit: fanno le stesse operazioni, ma accettano come operandi solo due registri (il registro di destinazione deve essere uno dei due registri in input) e possono utilizzare soltanto i primi otto registri. I progettisti hanno chiamato "Thumb ISA" questa versione ridotta dell'ISA ARM.

Alcune varianti aggiuntive delle istruzioni permettono di utilizzare, al posto di uno dei registri, delle costanti senza segno di 3, 8, 12, 16 o 24 bit. Nelle istruzioni di carica-

mento vengono sommati due registri (o un registro e una costante con segno di 8 bit) per specificare l'indirizzo di memoria da leggere. Il dato viene poi scritto in un altro registro specificato.

Il formato delle istruzioni ARM a 32 bit è mostrato nella figura 5.14. Il lettore attento noterà che alcuni formati hanno gli stessi campi (per esempio, LONG MULTIPLY e SWAP). Nel caso dell'istruzione SWAP, il decodificatore riconosce l'istruzione solo quando si accorge che la combinazione dei valori dei campi non è ammessa dall'istruzione MUL. Alcuni formati sono stati aggiunti per le estensioni delle istruzioni e per il Thumb ISA. Quando questo libro è stato scritto il numero di formati delle istruzioni era 21, ma questo numero continua a crescere (fra non molto qualche azienda la pubblicherà come "la macchina RISC più complessa del mondo"!). Tuttavia la maggior parte delle istruzioni utilizza ancora i formati mostrati nella figura.

31	2827	1615	87	0	Tipo di istruzione		
Cond	0 0 1	Opcode \$	Rn	Rd	Operand2		
Cond	0 0 0 0 0	A S	Rd	Rn	RS	1 0 0 1	Rm
Cond	0 0 0 0 1	U A S	RdHi	RdLo	RS	1 0 0 1	Rm
Cond	0 0 0 1 0	B 0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm
Cond	0 1 P U B W L	Rn	Rd		Offset		
Cond	1 0 0 P U S W L	Rn			Register List		
Cond	0 0 0 P U 1 W L	Rn	Rd	Offset1	1 S H 1	Offset2	
Cond	0 0 0 P U 0 W L	Rn	Rd	0 0 0 0	1 S H 1	Rm	
Cond	1 0 1 L			Offset			
Cond	0 0 0 1	0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn
Cond	1 1 0 P U N W L	Rn	CRd	CPNum		Offset	
Cond	1 1 1 0	Op1	CRn	CRd	CPNum	Op2 0	CRm
Cond	1 1 1 0	Op1 L	CRn	Rd	CPNum	Op2 1	CRm
Cond	1 1 1 1				SWI Number		
					Interrupt software		

Figura 5.14 Formati delle istruzioni a 32 bit di ARM.

I bit 26 e 27 di ogni istruzione sono quelli inizialmente utilizzati per determinare il formato dell'istruzione e dicono all'hardware dove trovare il resto del codice operativo, se c'è. Per esempio, se i bit 26 e 27 sono entrambi 0, se il bit 25 è 0 (l'operando non è un valore immediato) e se lo shift degli operandi in ingresso non è illegale (a indicare che l'istruzione è una moltiplicazione o una branch exchange), allora entrambe le sorgenti sono registri. Se invece il bit 25 è 1, allora una sorgente è un registro e l'altra una costante compresa tra 0 e 4095. In entrambi i casi la destinazione è un registro. È disponibile uno spazio di codifica sufficiente per un massimo di 16 istruzioni, tutte correntemente utilizzate.

Nelle istruzioni a 32 bit non è possibile includere una costante di 32 bit. L'istruzione MOVT imposta i 16 bit più significativi di un registro a 32 bit, lasciando che un'altra istruzione imposta i 16 bit rimanenti. La MOVT è l'unica istruzione nel suo formato.

Ogni istruzione da 32 bit ha lo stesso campo di 4 bit nei bit più significativi (da 28 a 31). Questi bit costituiscono il campo condizione e rendono ogni istruzione una *istruzione*.

ne predicated (predicated instruction). Le istruzioni predicated vengono normalmente eseguite nel processore, ma prima di scrivere il risultato in un registro (o in memoria), si controlla la condizione dell'istruzione. Per le istruzioni ARM la condizione è basata sul registro di stato del processore (PSR). Questo registro mantiene le proprietà aritmetiche dell'ultima operazione aritmetica (per esempio, zero, negativo, overflow, ...). Se la condizione non è verificata il risultato dell'istruzione condizionale viene eliminato.

Il formato delle istruzioni di salto incorpora il valore immediato più grande, utilizzato per calcolare l'indirizzo destinazione per i salti o le chiamate a procedura. Si tratta di un formato speciale, perché è l'unico in cui sono necessari 24 bit di dati per specificare un indirizzo. Per questa istruzione vi è un unico opcode di 3 bit. L'indirizzo corrisponde all'indirizzo destinazione diviso per 4: ciò permette un'estensione dei salti di circa 2^{25} , relativamente all'istruzione corrente.

I progettisti dell'ISA ARM volevano utilizzare completamente ogni combinazione di bit, incluse combinazioni illegali di operandi, per specificare le istruzioni. Questo approccio rende la logica di decodifica delle istruzioni estremamente complicata, ma allo stesso tempo consente al numero massimo di operazioni di essere codificate in istruzioni della lunghezza fissa di 16 o 32 bit.

5.3.5 Formati delle istruzioni dell'ATmega16 AVR

L'ATmega16 AVR dispone di sei semplici formati d'istruzioni, come illustrato nella Figura 5.15. Le istruzioni sono lunghe 2 o 4 byte. Il primo formato consiste in un opcode e due registri operandi, di cui uno è utilizzato solo in input e l'altro sia in input che in output. L'istruzione ADD per i registri, per esempio, utilizza questo formato.

Anche il secondo formato è di 16 bit e comprende un opcode aggiuntivo e un numero di registro di 5 bit. Questo formato permette di aumentare il numero di operazioni codificate dall'ISA, al prezzo di ridurre il numero di operandi a uno solo. Le istruzioni che utilizzano questo formato eseguono un'operazione unaria, ricevendo in ingresso un solo registro e scrivendo l'output sullo stesso registro. Tra gli esempi di queste operazioni vi sono la negazione e l'incremento.

Il terzo formato ha un operando immediato senza segno di 8 bit. Per far spazio a un valore immediato così grande per soli 16 bit, le istruzioni che utilizzano questa codifica possono avere un solo registro operando (utilizzato sia come input che come output) e il registro può appartenere soltanto all'insieme dei registri da R16 a R31 (in modo da poter limitare la codifica del numero di registro a 4 bit). Anche il numero dei bit per l'opcode è dimezzato, permettendo così a solo 4 istruzioni (SUBCI, SUBI, ORI, ANDI) di utilizzare questo formato.

Il quarto formato codifica le istruzioni di load e store, che utilizzano un operando immediato senza segno di 6 bit. Il registro base è un registro fissato che non viene specificato nella codifica dell'istruzione, perché è implicito nell'opcode.

Il quinto e il sesto formato sono utilizzati per i salti e le chiamate di procedura. Il primo utilizza un valore immediato con segno di 12 bit che viene aggiunto al valore del registro PC per calcolare la destinazione del salto. L'ultimo formato espande lo spiazzamento a 22 bit, portando la dimensione dell'istruzione AVR a 32 bit.

Formato	15	0	
1	00cc	ccrd	dddd rrrr
2	1001	010d	dddd cccc
3	01cc	KKKK	dddd KKKK
4	10Q0	QQcd	dddd cQQQ
5	11cc	KKKK	KKKK KKKK
31	1001	010K	KKKK 11cK KKKK KKKK KKKK
6	1001	010K	KKKK KKKK KKKK KKKK
			Call/jmp: call/jmp(c) #K

Figura 5.15 Il formato delle istruzioni dell'ATmega16 AVR.

5.4 Indirizzamento

Molte istruzioni contengono operandi e si pone il problema di come specificarne la posizione. L'indirizzamento è l'argomento che tratta di queste problematiche che ora affrontiamo.

5.4.1 Modalità d'indirizzamento

Finora ci siamo curati molto poco di come vengono interpretati i bit dei campi d'indirizzo per trovare gli operandi. È giunto ora il momento di approfondire l'argomento della modalità d'indirizzamento. Come vedremo, ci sono diversi modi per implementarla.

5.4.2 Indirizzamento immediato

Il modo più semplice con cui un'istruzione può specificare un operando è di contenere, nel campo riservato al suo indirizzo, l'operando stesso invece che un indirizzo o qualunque altra informazione che ne descriva la posizione. Un operando così specificato si dice **immediato**, poiché viene recuperato automaticamente dalla memoria nello stesso momento in cui viene effettuato il fetch dell'istruzione; dunque è immediatamente disponibile all'uso. La Figura 5.16 mostra una possibile istruzione immediata per il caricamento della costante 4 nel registro R1.

MOV	R1	4
-----	----	---

Figura 5.16 Istruzione per il caricamento della costante 4 nel registro R1.

L'indirizzamento immediato ha la virtù di non richiedere un riferimento supplementare in memoria per effettuare il fetch dell'operando. Presenta però lo svantaggio di poter fornire un solo operando per volta; inoltre, l'entità del valore è limitata dalla dimensione del campo. Ciononostante è una tecnica in uso presso molte architetture per la specifica di piccole costanti intere.

5.4.3 Indirizzamento diretto

Un metodo per specificare un operando in memoria è darne l'indirizzo completo. Questa modalità si chiama **indirizzamento diretto**. Al pari di quello immediato, l'indirizzamento diretto presenta alcune limitazioni: l'istruzione accederà sempre alla stessa locazione di memoria. Se da una parte il valore contenuto può cambiare, la locazione non può. Per questa ragione l'indirizzamento diretto serve solo ad accedere a variabili globali il cui indirizzo è noto in fase di compilazione. Nonostante ciò, tale modalità è molto usata, perché molti programmi definiscono variabili globali. Nel seguito considereremo il modo in cui il computer stabilisce quali indirizzi sono immediati e quali diretti.

5.4.4 Indirizzamento a registro

L'indirizzamento a registro è concettualmente analogo all'indirizzamento diretto, ma specifica un registro invece di una locazione di memoria. Si tratta della modalità d'indirizzamento di gran lunga più utilizzata nella quasi totalità dei computer, dato che i registri sono veloci in accesso e hanno indirizzi brevi. Molti compilatori si sforzano di prevedere quali variabili saranno richiamate più spesso (per esempio gli indici di ciclo) e le destinano ai registri.

Questa modalità d'indirizzamento è nota semplicemente come **modalità a registro**. Nelle architetture load/store quali l'OMAP4430 ARM, quasi tutte le istruzioni usano esclusivamente questa modalità d'indirizzamento. L'unico caso in cui non viene usata è quando un operando è trasferito dalla memoria in un registro (istruzione LOAD) o da un registro in memoria (istruzione STORE). Ma anche in questi casi uno degli operandi è un registro contenente l'indirizzo della parola di memoria in lettura o scrittura.

5.4.5 Indirizzamento a registro indiretto

In questa modalità l'operando in esame proviene o è destinato alla memoria, ma il suo indirizzo non è incorporato nell'istruzione, come nel caso dell'indirizzamento diretto: l'indirizzo è contenuto in un registro. Un indirizzo usato in questa maniera prende il nome di **puntatore**. Un grande vantaggio dell'indirizzamento a registro indiretto è che può referenziare la memoria senza dover necessariamente incorporare un intero indirizzo di memoria all'interno dell'istruzione. Per di più è anche possibile usare diverse parole di memoria in occasione di esecuzioni diverse della stessa istruzione.

Per capire perché può essere utile indirizzare una parola diversa a ogni esecuzione, immaginate un ciclo che passi in rassegna i 1024 elementi di un vettore d'interi e calcoli la loro somma nel registro R1. Altri due registri, poniamo R2 e R3, sono usati fuori dal ciclo per puntare rispettivamente al primo elemento dell'array e all'indirizzo immediatamente successivo all'ultimo elemento dell'array. Se l'array comincia all'indirizzo A ed è composto da 1024 interi di 4 byte ciascuno, il primo indirizzo dopo l'array sarà $A + 4096$. Un codice assemblativo tipico per un calcolo del genere su una macchina a due indirizzi è riportato nella Figura 5.17.

MOV R1,#0	; aggiorna la somma in R1, posto inizialmente a zero
MOV R2,#A	; R2 = indirizzo dell'array A
MOV R3,#A+4096	; R3 = indirizzo della prima parola dopo A
CICLO: ADD R1,(R2)	; recupera l'operando attraverso R2, registro indiretto
ADD R2,#4	; incrementa R2 di una parola (4 byte)
CMP R2,R3	; abbiamo già finito?
BLT CICLO	; se R2 < R3 non abbiamo finito, quindi si continua

Figura 5.17 Generico programma assemblativo per il calcolo della somma degli elementi di un array.

In questo semplice programma ci avvaliamo di parecchie modalità d'indirizzamento. Le prime tre istruzioni usano la modalità a registro per il primo operando (la destinazione) e la modalità immediata per il secondo operando (una costante denotata dal simbolo #). La seconda istruzione copia l'*indirizzo* di A in R2, non il suo contenuto, il che è specificato all'assembler tramite il simbolo #. allo stesso modo la terza istruzione copia in R2 l'indirizzo della prima parola oltre l'array.

È interessante notare che il ciclo vero e proprio non contiene nessun indirizzo di memoria. La quarta istruzione usa le modalità a registro e quella a registro indiretto; la quinta usa la modalità a registro e quella immediata; la sesta usa due volte la modalità a registro. L'istruzione BLT potrebbe usare un indirizzo di memoria, ma è più probabile che specifichi l'indirizzo del salto con uno spiazzamento di 8 bit relativo a se stessa. Il rifiuto totale di utilizzare indirizzi di memoria ha prodotto un ciclo conciso e veloce. Detto per inciso, questo è un vero programma per il Core i7, laddove abbiamo rinominato le istruzioni e i registri e modificato la notazione per renderlo di più facile lettura. Infatti la sintassi standard del linguaggio assemblativo del Core i7 (MASM) rasenta l'assurdo a causa del retaggio lasciatogli dall'8088, suo antenato diretto.

Facciamo notare che, in teoria, esiste un'altra soluzione per svolgere questa computazione, che non usa l'indirizzamento a registro indiretto: il ciclo avrebbe dovuto contenere un'istruzione per sommare A a R1, quale

```
ADD R1, A
```

dopodiché, a ogni iterazione del ciclo, l'istruzione stessa potrebbe venire incrementata di 4, di modo che dopo una sola iterazione diventi

```
ADD R1, A+4
```

e così via fino alla fine del ciclo.

Un programma che modifica se stesso si dice un programma auto-modificante. L'idea non fu di altri se non di John Von Neumann e poteva essere sensata sui primi computer, che non disponevano d'indirizzamento a registro indiretto. Oggigiorno i programmi automodificanti sono considerati di pessimo stile e difficili da capire. Inoltre non sono condivisibili da processi diversi e non funzionerebbero nemmeno su macchine con una cache di primo livello separata, laddove la I-cache (cache delle istruzioni) non disponesse di circuiti per i write-back (proprio perché i progettisti dell'hardware avrebbero supposto che i programmi non si automodifichino). Infine, i programmi auto-modificanti non possono funzionare su macchine che hanno spazi separati di memoria per dati e indirizzi. L'idea del codice auto-modificante è comunque, fortunatamente, scomparsa.

5.4.6 Indirizzamento indicizzato

Spesso è auspicabile poter referenziare una parola di memoria che si trova a un dato spiazzamento dal contenuto di un registro. Abbiamo visto qualche esempio con l'IJVM in cui le variabili locali sono referenziate specificando il loro spiazzamento rispetto a LV. L'indirizzamento alla memoria che si ottiene specificando un registro (in via esplicita o implicita), più uno spiazzamento costante, si dice **indirizzamento indicizzato**.

Gli accessi alle variabili locali dell'IJVM usano un puntatore alla memoria (LV) contenuto in un registro, più un piccolo spiazzamento contenuto nell'istruzione stessa, come mostrato dalla Figura 4.19(a). Tuttavia è possibile anche la soluzione opposta: tenere il puntatore alla memoria nell'istruzione e il piccolo offset in un registro. Per mostrare il funzionamento di questa modalità consideriamo un esempio. Supponiamo di avere due vettori, A e B , di 1024 parole ciascuno e di voler calcolare $A_i \text{ AND } B_i$ per tutti i componenti e poi fare l'OR di questi 1024 prodotti logici per verificare se c'è almeno una coppia di componenti che non sia nulla. Sarebbe possibile salvare gli indirizzi di A e B in due registri e poi visitare gli array in stretta successione, un elemento per volta, in modo analogo a quanto esposto nella Figura 5.17. Benché sia una soluzione corretta, possiamo fare di meglio e usare uno schema più generale, come illustrato nella Figura 5.18.

Il funzionamento di questo programma è semplice. C'è bisogno di quattro registri:

1. $R1$ – contiene l'OR cumulativo dei prodotti logici;
2. $R2$ – l'indice i usato per la visita degli array;
3. $R3$ – la costante 4096, che è il primo valore di i da non considerare;
4. $R4$ – un registro di lavoro per mantenere il calcolo di ogni prodotto.

CICLO: MOV R4,A(R2) AND R4,B(R2) OR R1,R4 ADD R2,#4 CMP R2,R3 BLT CICLO	; aggiorna gli OR in R1, posto inizialmente a zero ; R2 = indice i del prodotto corrente: $A[i]$ AND $B[i]$; R3 = indice del primo valore da non considerare ; R4 = $A[i]$; $R4 = A[i]$ AND $B[i]$; OR di tutti i prodotti booleani in R1 ; $i = i + 4$ (passi di 1 parola = 4 byte alla volta) ; abbiamo già finito? ; se $R2 < R3$ non abbiamo finito, quindi si continua
--	--

Figura 5.18 Generico programma assemblativo per il calcolo dell'OR di A , AND B (A e B sono array di 1024 elementi).

Dopo l'inizializzazione dei registri comincia il ciclo di sei istruzioni. La prima, etichettata come CICLO, effettua il fetch di A_i in $R4$ con modalità indicizzata: il registro $R2$ viene sommato all'indirizzo A e la somma è usata come riferimento in memoria, sebbene non venga memorizzata in nessun registro visibile all'utente. La notazione

```
MOV R4, A(R2)
```

indica che la destinazione, $R4$, è usata in modalità registro, mentre la sorgente usa la modalità indicizzata con offset A e registro $R2$. Se A vale, poniamo, 124300, l'aspetto reale di questa istruzione macchina è qualcosa di simile a quanto mostrato nella Figura 5.19.

MOV	R4	R2	124300
-----	----	----	--------

Figura 5.19 Rappresentazione di MOV R4, A(R2).

Alla prima iterazione del ciclo, $R2$ vale 0 (è stato così inizializzato), perciò la parola di memoria indicizzata è A_0 , all'indirizzo 124300, e questa viene salvata in $R4$. All'iterazione successiva $R2$ vale 4, perciò la parola di memoria indicizzata è A_4 , all'indirizzo 124304, e così via.

Come avevamo anticipato, in questo esempio l'offset che si trova nell'istruzione è in realtà il puntatore alla memoria e il valore contenuto nel registro è un piccolo intero che viene incrementato durante il calcolo. Questa forma ovviamente richiede che nell'istruzione ci sia un campo offset abbastanza grande da contenere un indirizzo, perciò è meno efficiente dell'altra alternativa. Nondimeno si rivela spesso la scelta migliore.

5.4.7 Indirizzamento indicizzato esteso

Alcune macchine dispongono della cosiddetta modalità d'indirizzamento indicizzato esteso, in cui l'indirizzo di memoria è calcolato sommando tra loro il contenuto di due

registri più un offset (opzionale). Un registro funge da base e l'altro da indice. Questa modalità ci sarebbe stata utile nell'esempio precedente, perché avremmo potuto iniziare R5 con A e R6 con B fuori dal ciclo. Così facendo avremmo rimpiazzato l'istruzione all'etichetta CICLO e quella successiva con

```
CICLO:    MOV R4, (R2+R5)
          AND R4, (R2+R6)
```

Poter disporre di una modalità d'indirizzamento indiretto tramite la somma di due registri e senza offset sarebbe l'ideale. Alternativamente, anche un'istruzione con un offset di 8 bit costituirebbe un miglioramento rispetto al codice originario, perché potremmo sempre porre entrambi gli offset a 0. D'altra parte, se gli offset fossero di 32 bit, non avremmo guadagnato nulla usando questa modalità. Nella pratica, le macchine che dispongono di questa modalità sono corredate della forma con offset di 8 o 16 bit.

5.4.8 Indirizzamento a stack

Abbiamo già sottolineato che è molto consigliabile rendere le istruzioni macchina quanto più corte possibile. Il limite alla riduzione della lunghezza degli indirizzi equivale a non averne per nulla. Come già visto nel Capitolo 4 le istruzioni senza indirizzi, come l'istruzione IADD, sono possibili in associazione con uno stack. In questo paragrafo analizziamo più da vicino l'indirizzamento a stack.

Notazione polacca inversa

È tradizione in matematica indicare l'operatore in mezzo agli operandi, come in $x + y$, invece che dopo gli operandi, come in $x y +$. La forma con l'operatore "in" mezzo è detta notazione infissa, mentre la forma con l'operatore dopo gli operandi si chiama postfissa o anche notazione polacca inversa, dal logico polacco J. Lukasiewicz (1958) che ne studiò le proprietà.

La notazione polacca inversa presenta un certo numero di vantaggi rispetto all'infissa in merito alla scrittura di formule algebriche. Per prima cosa qualsiasi formula può essere espressa correttamente senza parentesi. Inoltre la valutazione delle formule in tale notazione si addice particolarmente ai compilatori con stack. Infine, per gli operatori infissi si definisce una precedenza, che è arbitraria e poco gradita. Per esempio sappiamo che $a \times b + c$ vuol dire $(a \times b) + c$ e non $a \times (b + c)$ perché alla moltiplicazione è stata assegnata, in modo arbitrario, una precedenza maggiore rispetto alla somma. Ma lo scorrimento verso sinistra ha precedenza sull'AND booleano? Chi può dirlo? La notazione polacca inversa elimina questa seccatura.

Esistono molteplici algoritmi per convertire formule infisse in notazione polacca inversa. Quella fornita qui è un riadattamento di un'idea di E. W. Dijkstra. Supponete che una formula sia composta dai seguenti simboli: le variabili, gli operatori binari (a due operandi) $+ - \times /$ e le parentesi destra e sinistra. Il simbolo \perp delimita la formula: è anteposto al suo primo simbolo e segue l'ultimo.

La Figura 5.20 mostra un tracciato ferroviario che collega New York alla California, con al centro una diramazione che si diparte in direzione del Texas. Si pensi alla linea da New York alla California come a una linea principale con una ramificazione in basso,

verso il Texas, intesa come una soluzione per lo stoccaggio temporaneo. I nomi e le direzioni non sono importanti: ciò che importa è la distinzione tra la linea principale e il sito alternativo utilizzato come magazzino. A ogni simbolo della formula corrisponde un vagone ferroviario. Il treno procede verso ovest (verso sinistra) e ogni volta che un vagone raggiunge lo scambio deve fermarsi e stabilire se procedere direttamente verso la California o passare prima per il Texas. I vagoni contenenti le variabili procedono sempre diretti per la California, mentre quelli contenenti altri simboli, prima di entrare nello scambio, devono ispezionare il contenuto del vagone più vicino che si trova lungo la diramazione per il Texas.

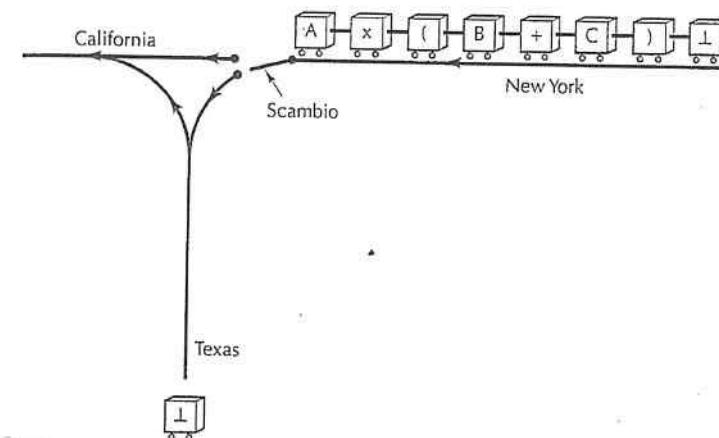


Figura 5.20 Ogni vagone ferroviario rappresenta un simbolo della formula infissa da convertire in notazione polacca inversa.

La tabella nella Figura 5.21 elenca le situazioni che si possono verificare, a seconda del vagone fermo allo scambio e del vagone a lui più vicino lungo il tracciato che porta in Texas. Il primo \perp va sempre in Texas. I numeri della figura rimandano ai seguenti eventi:

1. il vagone allo scambio procede verso il Texas;
2. il vagone che per ultimo ha intrapreso la linea per il Texas inverte il percorso e prosegue per la California;
3. il vagone allo scambio e quello che è entrato più di recente nella diramazione per il Texas vengono entrambi dirottati altrove e scompaiono (cioè si elidono);
4. stop: i simboli presenti al momento in California rappresentano la notazione polacca inversa della formula se letti da sinistra verso destra;
5. stop: è stato rilevato un errore sintattico nella formula originaria.

Vagone fermo allo scambio							
	+	-	x	/	()	
+	4	1	1	1	1	1	5
-	2	2	2	1	1	1	2
x	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(5	1	1	1	1	1	3

Figura 5.21 Tavola di decisione utilizzata dall'algoritmo per la conversione da notazione infissa a polacca inversa.

Al termine di ogni azione intrapresa viene effettuato un nuovo confronto tra il vagone attualmente fermo allo scambio, che potrebbe essere lo stesso del confronto precedente o il successivo, e il vagone che per ultimo è proseguito per il Texas. Il processo continua finché non si raggiunge l'evento 4. Notate che la linea per il Texas è usata come uno stack, dove l'instradamento di un vagone verso il Texas corrisponde a un'operazione di push, mentre l'inversione di marcia di un vagone che si trova già in Texas e che prosegue per la California corrisponde a un'operazione di pop.

L'ordine delle variabili è lo stesso nelle due notazioni, invece l'ordine degli operatori può cambiare. Nella notazione polacca inversa gli operatori compaiono nell'ordine in cui verranno effettivamente eseguiti durante la valutazione dell'espressione. La Figura 5.22 fornisce diversi esempi di formule infisse e delle stesse in notazione polacca inversa.

Notazione infissa	Notazione polacca inversa
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C - D)	A B + C D - /
A × B / C	A B × C /
((A + B) × C + D) / (E + F + G)	A B + C D × E F + G + /

Figura 5.22 Esempi di espressioni in notazione infissa e le stesse in notazione polacca inversa.

Valutazione delle formule in notazione polacca inversa

La notazione polacca inversa è la notazione ideale per la valutazione di una formula da parte di un computer dotato di stack. Una formula è costituita da n simboli, ciascuno dei quali è un operando o un operatore. L'algoritmo che si avvale di uno stack per la valu-

tazione di una formula in notazione polacca inversa è molto semplice: scorre la stringa da sinistra verso destra e, quando incontra un operando, lo impila sullo stack. Invece quando incontra un operatore ne esegue l'istruzione corrispondente.

La Figura 5.23 illustra la valutazione della formula

$$(8 + 2 \times 5) / (1 + 3 \times 2 - 4)$$

nell'IJVM. La stessa formula espressa in notazione polacca inversa è

$$825 \times +132 \times +4 - /$$

Passo	Stringa rimanente	Istruzione	Stack
1	8 2 5 × + 1 3 2 × + 4 - /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 - /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 - /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 - /	IMUL	8, 10
5	+ 1 3 2 × + 4 - /	IADD	18
6	1 3 2 × + 4 - /	BIPUSH 1	18, 1
7	3 2 × + 4 - /	BIPUSH 3	18, 1, 3
8	2 × + 4 - /	BIPUSH 2	18, 1, 3, 2
9	× + 4 - /	IMUL	18, 1, 6
10	+ 4 - /	IADD	18, 7
11	4 - /	BIPUSH 4	18, 7, 4
12	- /	ISUB	18, 3
13	/	IDIV	6

Figura 5.23 Uso dello stack per la valutazione di una formula in notazione polacca inversa.

Nella figura abbiamo introdotto le istruzioni IMUL e IDIV, rispettivamente di moltiplicazione e di divisione. Il numero sulla cima dello stack è l'operando destro, non quello sinistro; questa precisazione è importante perché nella sottrazione l'ordine degli operandi conta (a differenza dell'addizione e della moltiplicazione). In altre parole, IDIV è stata definita appositamente in modo tale che, dopo aver fatto il push del numeratore e poi quello del denominatore, la sua esecuzione produca come risultato la divisione corretta. Si noti la semplicità di generazione del codice per l'IJVM: si scorre la notazione polacca inversa della formula e si restituisce un'istruzione per ciascun simbolo. Se il simbolo è una costante o una variabile, si restituisce un'istruzione di push sullo stack; se il simbolo è un operatore, si restituisce l'istruzione che esegue l'operazione.

5.4.9 Modalità d'indirizzamento per istruzioni di salto

Fin qui abbiamo analizzato le istruzioni che operano sui dati. Anche le istruzioni di salto (e le chiamate di procedura) necessitano di modalità d'indirizzamento per specificare

l'indirizzo di destinazione. Le modalità riscontrate finora funzionano grosso modo anche per i salti. L'indirizzamento diretto è di certo un'opzione possibile, secondo cui l'indirizzo di destinazione viene semplicemente riportato per intero all'interno dell'istruzione.

Esistono anche altre modalità d'indirizzamento sensate. L'indirizzamento a registro indiretto consente al programma di calcolare l'indirizzo di destinazione, scriverlo in un registro e quindi effettuare il salto. È la modalità più flessibile perché l'indirizzo di destinazione può essere calcolato durante l'esecuzione, ma è anche la più incline alla generazione di bachi quasi impossibili da scoprire.

Un'altra modalità ragionevole è la modalità indicizzata, che specifica un certo offset rispetto all'indirizzo contenuto in un registro. Manifesta le stesse proprietà della modalità a registro indiretto.

Un'ulteriore opzione è l'indirizzamento relativo al PC: un offset (con segno) contenuto nell'istruzione stessa viene sommato al program counter per ottenere l'indirizzo di destinazione. In realtà non è altro che una modalità indicizzata che usa il PC come registro base.

5.4.10 Ortogonalità dei codici operativi e delle modalità d'indirizzamento

Dal punto di vista software, le istruzioni e l'indirizzamento dovrebbero manifestare una struttura regolare, con un numero minimo di formati d'istruzioni. Tale struttura agevolerebbe molto il compilatore a produrre codice di qualità. Gli opcode dovrebbero consentire tutte le modalità d'indirizzamento, laddove sensato, e ogni registro (anche FP, SP e PC) dovrebbe essere utilizzabile da tutte le modalità a registro.

Un esempio di progetto elegante per una macchina a tre indirizzi prevede i formati d'istruzioni di 32 bit della Figura 5.24 e supporta fino a 256 codici operativi. Il formato 1 prevede due indirizzi sorgente e un indirizzo destinazione ed è usato da tutte le istruzioni logico-aritmetiche.

Bit	8	1	5	5	5	8
1	OPCODE	0	DEST	SRC1	SRC2	
2	OPCODE	1	DEST	SRC1		OFFSET
3	OPCODE				OFFSET	

Figura 5.24 Semplice progetto dei formati d'istruzione di una macchina a tre indirizzi.

Il campo finale di 8 bit è inutilizzato e può essere destinato per differenziare ulteriormente le istruzioni. Per esempio, un solo opcode potrebbe specificare l'insieme di operazioni in virgola mobile, e il campo supplementare potrebbe distinguere l'una dall'al-

tra. Proseguendo, se il bit 23 è asserito, l'istruzione usa il formato 2, in cui il secondo operando non è più un registro, bensì una costante immediata con segno di 13 bit. È questo un formato confacente alle istruzioni LOAD e STORE che referenziano la memoria in modo indicizzato.

Occorrono poche altre istruzioni, come i salti condizionati, facilmente adattabili al formato 3. Per esempio tutte le istruzioni di salto (condizionato), di chiamata di procedura, e così via, potrebbero avere un proprio opcode, il che lascia 24 bit a disposizione per l'offset relativo al PC. Se l'offset è interpretato come numero di parole, allora abbraccia un intervallo di ± 32 MB. Si potrebbe anche riservare un certo numero di opcode per istruzioni LOAD e STORE che necessitano dell'offset lungo del formato 3. Si tratterebbe d'istruzioni non del tutto generali (che per esempio potrebbero agire per definizione solo sul registro R0), ma il loro uso sarebbe alquanto limitato.

Si consideri ora il progetto di una macchina a due indirizzi, mostrata nella Figura 5.25, che può specificare parole di memoria per entrambi gli operandi. Una tale macchina è in grado di sommare una parola di memoria a un registro, oppure sommare tra loro due parole di memoria. Allo stato attuale gli accessi in memoria sono relativamente gravosi, perciò questo progetto non è quasi mai preso in considerazione. Se i progressi tecnologici renderanno in futuro gli accessi in memoria e alla cache più convenienti, risulterà un progetto particolarmente semplice ed efficiente. Le macchine PDP-11 e VAX ottennero un grande successo e dominarono la scena informatica per vent'anni proprio grazie a un progetto di questo tipo.

Bit	8	3	5	4	3	5	4
	OPCODE	MODO	REG	OFFSET	MODO	REG	OFFSET
(Offset o indirizzo diretto di 32 bit opzionale)							
(Offset o indirizzo diretto di 32 bit opzionale)							

Figura 5.25 Semplice progetto dei formati d'istruzione di una macchina a due indirizzi.

Anche in questo schema gli opcode sono di 8 bit, ma questa volta disponiamo di 12 bit per specificare la sorgente e di altri 12 per la destinazione. Ciascun operando è corredato di 3 bit per la modalità, 5 bit per il registro e 4 per l'offset. Con 3 bit potremmo gestire tutte le modalità, immediata, diretta, a registro, a registro indiretto, indicizzata, a stack, e avremmo ancora spazio per altre due modalità. È questo un progetto semplice ed elegante, assicura una facile compilazione ed è abbastanza flessibile, soprattutto se il program counter, il puntatore allo stack e il puntatore alle variabili locali sono tutti accessibili come registri d'uso generale.

L'unico problema che si presenta è che per l'indirizzamento diretto abbiamo bisogno di un numero di bit maggiore per gli indirizzi. La soluzione del PDP-11 e del VAX era di aggiungere all'istruzione una parola supplementare per ogni indirizzo di operando indirizzato direttamente. Potremmo anche usare una delle due modalità d'indirizzamento libere per una modalità indicizzata con un offset di 32 bit posposto all'istruzione. Così facendo, una somma di due operandi in memoria, entrambi indirizzati direttamente o

con la lunga forma indicizzata, prenderebbe nel caso peggiore 96 bit e userebbe tre cicli di bus (uno per l'istruzione, due per gli indirizzi). Sarebbero inoltre necessari tre cicli aggiuntivi per prelevare i due operandi e per scrivere il risultato. D'altra parte quasi tutte le architetture RISC richiederebbero almeno 96 bit, se non di più, per sommare due parole di memoria qualsiasi, e userebbero almeno quattro cicli di bus, a seconda della modalità di indirizzamento degli operandi.

Le alternative alla Figura 5.26 sono molte. Questo progetto rende possibile l'esecuzione dell'istruzione

```
i = j;
```

con una sola istruzione di 32 bit, ammesso che *i* e *j* si trovino entrambe tra le prime 16 variabili locali. Per variabili oltre la sedicesima bisogna optare però per gli offset di 32 bit. Un'alternativa sarebbe quella di avere un altro formato con un solo offset di 8 bit invece di due offset di 4 bit, più una regola per stabilire se si riferisce alla sorgente o alla destinazione, ma non a entrambe. Le possibilità sono illimitate così come i compromessi, e i progettisti di una macchina devono giocare su molti fattori per ottenere un buon risultato.

5.4.11 Modalità d'indirizzamento del Core i7

Le modalità d'indirizzamento del Core i7 sono assai irregolari e cambiano a seconda che una certa istruzione sia in modalità di 16, 32 o 64 bit. Non affronteremo le modalità di 16 e 64 bit, dato che quella di 32 è già abbastanza intricata. Il Pentium prevede le modalità immediata, diretta, a registro, a registro indiretto, più una speciale per l'indirizzamento di elementi di un array. Il problema è che non tutte le modalità si applicano a tutte le istruzioni, e non tutti i registri sono utilizzabili da tutte le modalità, il che accresce la difficoltà delle mansioni del compilatore e porta a un codice più scadente.

Il byte MODE della Figura 5.13 controlla le modalità d'indirizzamento. Uno degli operandi è specificato in combinazione dai campi MOD e R/M, mentre l'altro è sempre un registro, specificato dal valore del campo REG. La Figura 5.26 elenca le 32 combinazioni che possono essere specificate congiuntamente dai 2 bit di MOD e dai 3 bit di R/M. Per esempio quando entrambi i campi valgono zero l'operando viene letto all'indirizzo di memoria contenuto nel registro EAX.

Le colonne 01 e 10 hanno a che fare con modalità in cui il registro è sommato a un offset di 8 o 32 bit posposto all'istruzione. Se l'offset è di 8 bit, viene portato a 32 (estensione del segno) prima della somma. Per fare un esempio, un'istruzione ADD, con R/M = 011, MOD = 01 e offset pari a 6, calcola la somma di EBX più 6 e usa il risultato come indirizzo della parola di memoria contenente un operando. EBX non ne risulta modificato.

La colonna MOD = 11 consente di specificare due registri alla volta: il primo è usato per le istruzioni di una parola, il secondo per le istruzioni di un byte. Osservate la parziale irregolarità della tabella: per esempio non c'è modo di usare EBP in modalità indiretta, né di usare ESP come base di un offset.

R/M	MOD			
	00	01	10	11
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX o AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX o CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX o DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX o BL
100	SIB	SIB con OFFSET8	SIB con OFFSET32	ESP o AH
101	Diretta	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP o CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI o DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI o BH

Figura 5.26 Modalità d'indirizzamento a 32 bit del Core i7. M[x] è la parola di memoria all'indirizzo x.

In alcune modalità c'è un byte aggiuntivo, detto SIB, che segue il byte MODE (Figura 5.13). Il byte SIB specifica due registri e un fattore di scala. Quando il byte SIB è presente, l'indirizzo dell'operando si calcola moltiplicando l'indirizzo indice per 1, 2, 4 o 8 (a seconda della scala), sommando il risultato al registro base; infine, c'è la possibilità che il tutto vada sommato a uno spiazzamento di 8 o 32 bit, secondo quanto specificato da MOD. Pressoché tutti i registri possono essere usati come indice o come base.

Le modalità SIB sono utili per accedere a elementi di un array; considerate per esempio l'istruzione Java

```
for (i = 0; i < n; i++) a[i] = 0;
```

dove a è un array d'interi di 4 byte locale alla procedura corrente. In genere EBP è usato per puntare alla base del record d'attivazione dello stack contenente le variabili e gli array locali, come mostrato nella Figura 5.27. Il compilatore potrebbe mantenere *i* in EAX e, per accedere ad *a[i]*, userebbe una modalità SIB in cui l'indirizzo dell'operando si otterrebbe comando $4 \times \text{EAX} + \text{EBP} + 8$. L'istruzione Java potrebbe così assegnare un valore ad *a[i]* con una sola istruzione di STORE.

Il gioco vale la candela? Difficile a dirsi. Sicuramente questa istruzione, se usata propriamente, fa risparmiare qualche ciclo. La frequenza del suo utilizzo dipende dal compilatore e dall'applicazione. Il problema è che questa istruzione occupa dello spazio sul chip che si sarebbe potuto impiegare in modo diverso se non fosse esistita. Per esempio la cache di primo livello avrebbe potuto essere più grande, o il chip più piccolo, consentendo forse una velocità di clock leggermente superiore.

Sono questi i compromessi sui quali i progettisti sono spesso chiamati a pronunciarsi. Di norma vengono eseguite numerose simulazioni prima dell'implementazione sul silicio, ma le simulazioni sono indicative solo se si ha già una buona idea di quel che sarà il carico di lavoro. Si può scommettere sul sicuro che i progettisti dell'8088 non abbiano incluso i browser di Internet nel loro insieme di prova. Ciononostante, alcuni discendenti di quel prodotto sono oggi usati principalmente per la navigazione del Web

e così le scelte fatte vent'anni fa potrebbero rivelarsi completamente inadeguate per le applicazioni correnti. Ancora una volta in nome della retrocompatibilità, quando una caratteristica entra in un progetto è impossibile estrometterla.

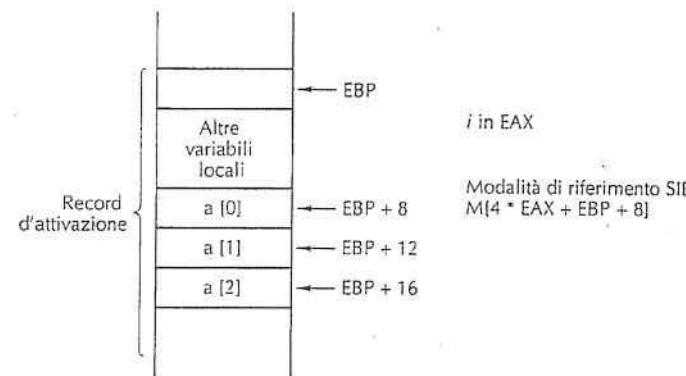


Figura 5.27 Accesso ad $a[i]$.

5.4.12 Modalità d'indirizzamento dell'OMAP4430

Nell'ISA dell'OMAP4430 tutte le istruzioni si avvalgono dell'indirizzamento immediato o a registro, eccezion fatta per quelle che indirizzano la memoria. Nella modalità a registro sono i 5 bit a specificare quale indirizzo usare. Nella modalità immediata, i dati sono contenuti in una costante (con segno) di 12 bit. Non ci sono altre modalità disponibili per le istruzioni logico-aritmetiche o simili.

Ci sono due tipi d'istruzioni che indirizzano la memoria: le istruzioni di load (LDR) e quelle di store (STR). Le istruzioni LDR e STR possono indirizzare la memoria in tre modalità diverse: nella prima si usa la somma di due registri come valore d'indirizzamento indiretto; la seconda calcola l'indirizzo come somma di un registro base e di uno spiazzamento con segno di 13 bit; la terza calcola l'indirizzo come somma del program counter e di uno spiazzamento con segno di 13 bit. Quest'ultima modalità, chiamata indirizzamento relativo al program counter, è utile per caricare costanti memorizzate con il codice del programma.

5.4.13 Modalità d'indirizzamento dell'ATmega168 AVR

L'ATmega168 presenta una struttura d'indirizzamento abbastanza regolare. Ci sono quattro modalità basilari. La prima è la modalità di indirizzamento a registro, in cui l'operando è un registro. I registri sono utilizzati sia come sorgente che come destinazione. La seconda è la modalità di indirizzamento immediato, in cui un valore immediato senza segno di 8 bit viene codificato nell'istruzione. Le restanti modalità possono essere utilizzate soltanto dalle istruzioni di load e store. La terza modalità è l'indirizza-

mento diretto. In questa modalità l'operando è in memoria a un indirizzo contenuto nell'istruzione stessa. Nelle istruzioni a 16 bit l'indirizzamento diretto è limitato a 7 bit (possono quindi essere specificati solo gli indirizzi da 0 a 127). L'architettura AVR definisce anche un'istruzione a 32 bit, in modo da poter ospitare un indirizzo di 16 bit, garantendo così il supporto per 64 KB di memoria. La quarta modalità è l'indirizzamento indiretto basato su un registro. In questo caso il registro contiene un puntatore all'operando. Visto che i registri normali hanno una dimensione di 8 bit, le istruzioni di load e store utilizzano coppie di registri per specificare indirizzi di memoria a 16 bit, così da poter indirizzare fino a 64 KB di memoria. L'architettura supporta l'utilizzo di tre copie di registri, chiamate X, Y e Z e formate rispettivamente dai registri R26/R27, R28/R29 e R30/R31. Per caricare un indirizzo nel registro X, per esempio, il programma dovrà caricare un valore di 8 bit nei registri R26 e R27, utilizzando due istruzioni.

5.4.14 Analisi delle modalità d'indirizzamento

Abbiamo dunque introdotto un certo numero di modalità d'indirizzamento. Quelle utilizzate da Core i7, OMAP4430 e ATmega168 sono riassunte nella Figura 5.28. Si tenga conto però, come già sottolineato, che non tutte le istruzioni possono avvalersi della gamma completa delle modalità d'indirizzamento.

Modalità d'indirizzamento	Core i7	OMAP4430 ARM	ATmega168 AVR
Immediata	×	×	×
Diretta	×		×
A registro	×	×	×
A registro indiretta	×	×	×
Indicizzata	×	×	
Indicizzata estesa		×	

Figura 5.28 Modalità d'indirizzamento a confronto.

Nella pratica un ISA, per essere efficace, non ha bisogno di molte modalità d'indirizzamento. Oggigiorno la maggior parte del codice scritto a questo livello viene generato dai compilatori (a parte forse nel caso dell'ATmega168), perciò gli aspetti più importanti delle modalità d'indirizzamento di un'architettura sono la possibilità di scegliere tra poche alternative chiare, il cui costo (in termini di tempo di calcolo e dimensioni del codice) sia valutabile prontamente. Ciò si traduce in una presa di posizione drastica: una macchina dovrebbe offrire ogni scelta possibile, oppure una sola. Ogni via di mezzo costringe il compilatore a fare delle scelte per cui non è abbastanza preparato o sofisticato.

Le architetture più eleganti dispongono in genere di un numero estremamente limitato di modalità d'indirizzamento e pongono vincoli molto stringenti al loro uso. Nella pratica, alla maggior parte delle applicazioni basta poter disporre delle modalità immediata, diretta, a registro e indicizzata. Inoltre, ogni registro dovrebbe essere utilizzabile

al pari di ogni altro, compresi il puntatore al record d'attivazione, il puntatore allo stack e il program counter. Modalità d'indirizzamento più complicate possono sì ridurre il numero d'istruzioni, ma al costo dell'introduzione di sequenze di operazioni non facilmente eseguibili in modo parallelo ad altre operazioni sequenziali.

Abbiamo così completato lo studio dei compromessi possibili tra opcode e indirizzi da una parte, e forme d'indirizzamento dall'altra. Ogniqualsiasi volta vi avvicinate a un nuovo computer dovrete esaminarne le istruzioni e le modalità d'indirizzamento, non solo per scoprire quali sono disponibili, ma anche per capire le scelte fatte e le conseguenze che avrebbero avuto le scelte alternative.

5.5 Tipi d'istruzioni

Anche se possono differire nei dettagli, le istruzioni del livello ISA possono essere suddivise approssimativamente in una mezza dozzina di gruppi facilmente rintracciabili in tutte le macchine. Oltre a questi, ciascun computer dispone di una manciata d'istruzioni insolite, aggiunte per motivi di compatibilità con modelli precedenti o a seguito di un'idea brillante di un progettista, o perché un'agenzia governativa ha pagato il produttore perché la includesse nel progetto. Proveremo a dar conto brevemente di tutte le categorie più comuni, senza pretendere di essere esaustivi.

5.5.1 Istruzioni di trasferimento dati

Poter copiare dati da una locazione all'altra è fondamentale per tutte le operazioni. Per copia intendiamo la creazione di un nuovo oggetto costituito da una sequenza di bit identica all'originale. L'uso tecnico della parola "trasferimento" è diverso da quello del linguaggio comune. Quando diciamo che Mario Cervi si è trasferito da New York alla California, non intendiamo dire che è stata creata una copia identica di Mario Cervi in California e che l'originale si trova ancora a New York. Quando diciamo che il contenuto della locazione di memoria 2000 è stato trasferito in un qualche registro, intendiamo sempre che vi è stata creata una copia fedele e che l'originale si trova ancora indisturbato nella sua locazione 2000. Le istruzioni di trasferimento dati dovrebbero chiamarsi piuttosto istruzioni di "duplicazione dei dati", ma oramai la locuzione "trasferimento di dati" è entrata nell'uso.

Ci sono due ragioni per copiare i dati da una locazione a un'altra, una delle quali è fondamentale: l'assegnamento di valori a variabili. L'assegnamento

A = B

si implementa con la copia del valore all'indirizzo di memoria *B* nella locazione di memoria *A*, come specificato dal programmatore. La seconda motivazione per la copia dei dati è prepararli a un accesso e a un uso efficienti. Si è già visto come molte istruzioni abbiano accesso solo alle variabili contenute nei registri. Poiché i dati possono provenire da due sorgenti (la memoria o i registri) ed essere destinati a due locazioni (in memoria o nei registri), ci sono quattro tipi diversi di trasferimenti possibili. Alcuni computer dispongono di quattro istruzioni diverse, una per ogni situazione, altri hanno

una sola istruzione per tutte le situazioni. Altri ancora usano LOAD per trasferire dalla memoria verso i registri, STORE dai registri alla memoria, MOVE per i trasferimenti tra registri e non dispongono di alcuna istruzione per la copia tra locazioni di memoria.

Le istruzioni per il trasferimento dati devono specificare in qualche maniera la quantità di dati da trasferire. In alcuni ISA esistono istruzioni per trasferire quantità di dati variabili da un byte fino all'intera memoria. Sulle macchine con parole di lunghezza fissa l'unità di trasferimento consueta è proprio la parola. Il trasferimento di quantità maggiori o minori di una parola va svolto via software per mezzo di operazioni di scorrimento e fusione. Alcuni ISA prevedono funzionalità aggiuntive per la copia sia di quantità più piccole di una parola (solitamente per multipli di byte), sia di più parole per volta. La copia di più parole è tanto più delicata quanto più grande è il massimo numero di parole trasferibili, perché un'operazione di questo genere può impiegare molto tempo e c'è il rischio che venga interrotta nel bel mezzo dell'esecuzione. Alcune macchine con lunghezza di parola variabile dispongono d'istruzioni che specificano solo gli indirizzi sorgente e destinazione, senza indicare la quantità da trasferire, così la copia dei dati continua fino al raggiungimento di un marcitore di fine dati.

5.5.2 Operazioni binarie

Le operazioni binarie sono quelle che producono un risultato dalla combinazione di due operandi. Tutti gli ISA hanno istruzioni per l'addizione e la sottrazione d'interi. Anche la moltiplicazione e la divisione d'interi è pressoché standard. È abbastanza ovvio che un computer sia dotato d'istruzioni aritmetiche e non ci dilunghiamo in merito.

Un altro insieme di operazioni binarie comprende le istruzioni booleane. Benché esistano 16 funzioni booleane in due variabili, ben poche macchine (forse nessuna) dispongono d'istruzioni per tutte e 16. In genere sono disponibili AND, OR e NOT, qualche volta anche XOR (OR ESCLUSIVO), NOR e NAND.

Un uso importante di AND è l'estrazione di bit da una parola. Considerate per esempio una macchina con parole di 32 bit che possono ospitare quattro caratteri di 8 bit. Se si vuole separare il secondo carattere dagli altri tre al fine di visualizzarlo sullo schermo, è necessario creare una parola che lo contenga negli 8 bit più a destra e che contenga tutti zero nei 24 bit rimanenti, perciò detta parola giustificata a destra.

L'estrazione del carattere avviene facendo l'AND della parola con una costante, detta maschera. Il risultato di questa operazione è che tutti i bit indesiderati vengono posti a zero, vale a dire mascherati, come mostrato nello schema seguente

10110111 10111100 11011011 10001011	A
<u>00000000 11111111 00000000 00000000</u>	B (maschera)
00000000 10111100 00000000 00000000	A AND B

Quindi il risultato viene fatto scorrere di 16 bit verso destra in modo da isolare il carattere da estrarre all'estremità destra della parola.

Un uso importante di OR è quello di impacchettare bit in una parola, che è l'operazione inversa dell'estrazione. Per cambiare gli 8 bit meno significativi di una parola da 32 bit senza modificare gli altri, per prima cosa mascheriamo gli 8 bit indesiderati, dopodiché il nuovo carattere è inserito facendone l'OR, come mostrato di seguito.

```

10110111 10111100 11011011 10001011 A
11111111 11111111 11111111 00000000 B (maschera)
10110111 10111100 11011011 00000000 A AND B
00000000 00000000 00000000 01010111 C
10110111 10111100 11011011 01010111 (A AND B) OR C

```

L'operazione AND tende a rimuovere i bit 1, perché il suo risultato non contiene mai più bit 1 di quanti ce ne siano in ciascun operando. L'operazione OR tende a inserire bit 1 perché il risultato contiene sempre almeno tanti bit 1 quanti ce ne sono nell'operando con numero maggiore. L'operazione XOR, d'altra parte, ha un comportamento simmetrico, perché tende a inserire in media tanti bit 1 quanti ne rimuove. Si tratta di una proprietà utile in alcune situazioni, come nella generazione di numeri casuali.

Molti dei computer odierni supportano anche un insieme d'istruzioni in virgola mobile, più o meno corrispondenti alle operazioni aritmetiche sugli interi. Gran parte delle macchine mette a disposizione almeno due formati di numeri in virgola mobile, il più corto per esigenza di velocità, l'altro per quelle situazioni particolari in cui si richiede una precisione maggiore. Ci sono molte varianti possibili per i formati in virgola mobile, eppure c'è uno standard che è oggi quasi universalmente accettato: IEEE 754. I numeri in virgola mobile, e il formato IEEE 754, sono presentati nell'Appendice B.

5.5.3 Operazioni unarie

Le operazioni unarie prendono in ingresso un operando e restituiscono un risultato. Queste istruzioni sono in genere più corte di quelle binarie, visto che contengono un indirizzo in meno, sebbene spesso richiedano la specifica di altre informazioni.

Le istruzioni per lo scorrimento (*shift*) o la rotazione del contenuto di una parola si rivelano molto utili, perciò sono spesso presenti in più varianti. Le operazioni di scorrimento possono spostare i bit verso destra o verso sinistra, e i bit che fuoriescono dalla parola si intendono perduti. Le rotazioni sono scorrimenti in cui i bit che escono da un'estremità della parola riappaiono all'altra estremità. Illustriamo con un esempio la differenza tra le due operazioni.

```

00000000 00000000 00000000 01110011 A
00000000 00000000 00000000 00011100 A scorso verso destra di 2 bit
11000000 00000000 00000000 00011100 A ruotato verso destra di 2 bit

```

Scorrimenti e rotazioni sono utili in entrambe le direzioni. Se una parola di n bit è ruotata verso sinistra di k bit, si ottiene lo stesso risultato che ruotandola verso destra di $n - k$ bit.

Gli scorrimenti verso destra sono spesso usati in associazione con l'estensione del segno, ovvero le posizioni all'estremità di sinistra lasciate vacanti dallo scorrimento vengono riempite con il bit di segno originario, 0 o 1. È come se il bit di segno venisse trascinato verso destra dallo scorrimento. Tra le altre cose, ciò implica che un numero negativo resta negativo. È quanto succede nel seguente esempio di scorrimento di 2 bit verso destra.

```

11111111 11111111 11111111 11110000 A
00111111 11111111 11111111 11111100 A scorso senza estensione del segno
11-111111 11111111 11111111 11111100 A scorso con estensione del segno

```

Un'applicazione importante dello scorrimento è la moltiplicazione e la divisione per potenze di 2. Se un intero positivo viene fatto scorrere di k bit verso sinistra allora, a meno di overflow, il risultato rappresenta il numero iniziale moltiplicato per 2^k . Se un numero positivo viene fatto scorrere verso destra di k bit, si ottiene il numero iniziale diviso per 2^k .

Lo scorrimento può essere usato per accelerare certe operazioni aritmetiche. Si consideri, per esempio, il calcolo di $18 \times n$ con n intero positivo. Poiché $18 \times n = 16 \times n + 2 \times n$, è possibile ottenere $16 \times n$ facendo scorrere n verso sinistra di 4 bit, $2 \times n$ facendo scorrere n verso sinistra di 1 bit. La somma di questi due numeri è proprio $18 \times n$, così la moltiplicazione è stata realizzata tramite un trasferimento, due scorrimenti e un'addizione, il che è spesso più veloce di una moltiplicazione vera e propria. Ovviamente è uno stratagemma che il compilatore può usare solo se almeno uno dei fattori è una costante.

Tuttavia lo scorrimento di numeri negativi, anche se fatto con estensione del segno, dà risultati molto diversi. Si consideri per esempio il numero -1 , in complemento a uno, che, se fatto scorrere di una posizione verso sinistra, produce -3 . Un altro scorrimento di un bit verso sinistra porta a -7 :

```

11111111 11111111 11111111 11111110 -1 in complemento a uno
11111111 11111111 11111111 11111100 -1 scorso verso sinistra di 1 bit = -3
11111111 11111111 11111111 11111000 -1 scorso verso sinistra di 2 bit = -7

```

Lo scorrimento verso sinistra dei numeri negativi in complemento a uno non equivale a moltiplicare per 2. Invece il loro scorrimento a destra simula la divisione correttamente. Prendete ora la rappresentazione di -1 in complemento a due. Facendola scorrere di 6 bit verso destra si ottiene ancora -1 , il che è sbagliato visto che la parte intera di $-1/64$ è 0:

```

11111111 11111111 11111111 11111111 -1 in complemento a due
11111111 11111111 11111111 11111111 -1 scorso di 6 bit verso destra = -1

```

In generale lo scorrimento verso destra introduce errori perché tronca il risultato. Invece lo scorrimento a sinistra simula davvero la moltiplicazione per 2.

Le operazioni di rotazione sono utili per l'impacchettamento di sequenze di bit in parole e per il loro spaccettamento. Se si vuole esaminare una parola bit per bit, la si può ruotare di 1 bit alla volta (non importa in che direzione) ed esaminare a ogni passo il contenuto del bit di segno; dopo aver esaminato tutti i bit la parola risulta ripristinata nella sua forma originale. Le operazioni di rotazione sono più genuine delle operazioni di scorrimento, perché non comportano perdita d'informazione: gli effetti di un'operazione di rotazione arbitraria possono essere annullati da un'altra operazione di rotazione.

Alcune operazioni binarie coinvolgono certi operandi così di frequente che alle volte gli ISA dispongono d'istruzioni unarie per svolgerle più velocemente. La copia del

valore zero in una certa parola di memoria o in un registro è estremamente frequente nella fase di inizializzazione di una computazione. La copia di zero è naturalmente un caso speciale d'istruzione di trasferimento dati. Per ragioni di efficienza, si usa spesso l'operazione CLR che contiene un solo indirizzo, quello della locazione da azzerare (*clear*).

La somma di una parola con il valore 1 è usata comunemente per contare. Una forma unaria dell'istruzione ADD è l'operazione INC che incrementa di 1. L'operazione di negazione è un altro esempio: negare X corrisponde a calcolare $0 - X$, una sottrazione binaria, e poiché è un'operazione di uso frequente alle volte si fornisce un'apposita istruzione unaria NEG. Sottolineiamo qui la differenza tra l'operazione aritmetica NEG e l'operazione logica NOT. L'operazione NEG produce l'opposto di un numero, quello cioè che dà 0 se sommato al numero originale. L'operazione NOT si limita a invertire i bit della parola. Le due operazioni sono molto simili, e infatti sono identiche nella rappresentazione in complemento a uno (nell'aritmetica in complemento a due, l'istruzione NEG si ottiene invertendo i singoli bit e poi aggiungendo 1).

Le istruzioni a uno o due operandi sono spesso raggruppate in base al loro uso invece che secondo il numero di operandi richiesto. Un insieme contiene le operazioni aritmetiche, compresa la negazione. Un altro insieme comprende le operazioni logiche e lo scorrimento, dal momento che vengono spesso usate congiuntamente per realizzare l'estrazione di dati.

5.5.4 Confronti e salti condizionati

Quasi tutti i programmi hanno bisogno della capacità di esaminare il contenuto dei dati e alterare la sequenza di esecuzione delle istruzioni in base al risultato dell'ispezione. Un semplice esempio di ciò è la funzione radice quadrata, \sqrt{x} : se x è negativo la procedura restituisce un messaggio d'errore, altrimenti calcola la radice quadrata. La funzione sqrt (da square root) deve perciò essere in grado di esaminare x e di scegliere come proseguire nell'esecuzione a seconda che x sia o meno negativa.

Un metodo comune per farlo è fornire istruzioni di salto condizionato che verificano una certa condizione e saltano a un particolare indirizzo di memoria se la condizione è soddisfatta. Alle volte c'è un bit nell'istruzione che indica se il salto deve avvenire a condizione soddisfatta o non soddisfatta. Spesso l'indirizzo di destinazione del salto non è assoluto, ma relativo all'istruzione corrente.

La condizione che viene testata più comunemente è se un determinato bit della macchina è posto o meno a 0. Se un'istruzione esamina il bit di segno di un numero e salta all'etichetta ETICHETTA qualora il bit valga 1, allora il comando che si trova alla posizione ETICHETTA sarà eseguito solo se il numero della condizione è negativo, altrimenti (se il numero è nullo o positivo) verrà eseguito il comando che segue il salto condizionato.

Molte macchine hanno bit usati per specificare certe condizioni. Per esempio ci potrebbe essere un bit di overflow asserito ogniqualvolta un'operazione aritmetica produce un risultato errato. Esaminare questo bit vuol dire verificare se l'istruzione precedente abbia o meno causato un overflow, e in caso positivo è possibile saltare a una routine di errore in grado di intraprendere le dovute azioni correttive.

Analogamente, alcuni processori hanno un bit di riporto che viene asserito quando si verifica un riporto all'ultimo bit di sinistra, per esempio quando vengono sommati due numeri negativi. Un riporto al bit più significativo è un evento normale, da non confondere con un overflow. Il test del bit di riporto è necessario all'aritmetica in precisione multipla (per esempio quando un intero è rappresentato in due o più parole).

Verificare se un parola vale zero è importante per i cicli e per molti altri scopi. Se tutte le istruzioni di salto condizionato esaminassero un bit alla volta, ci vorrebbero tanti confronti quanti sono i bit per verificare che tutti i bit di una parola valgono zero. Per evitare questo scenario, molte macchine mettono a disposizione istruzioni di confronto per esaminare una parola intera e, se nulla, effettuare il salto. Naturalmente questa soluzione non fa che passare la "patata bollente" alla microarchitettura. Di solito l'hardware contiene un registro i cui bit sono tutti messi in OR al fine di disporre di un solo bit che stabilisce immediatamente se la parola contiene uno o più bit asseriti. Il bit Z della Figura 4.1 viene calcolato di norma invertendo il risultato dell'OR di tutti i bit in uscita dalla ALU.

I confronti tra parole o tra caratteri sono importanti per verificare se sono uguali e, se non lo sono, per stabilire qual è maggiore, il che è indispensabile per poterli ordinare. Per effettuare questo test sono necessari tre indirizzi: due per i dati e uno verso cui saltare se la condizione è vera. I computer con formati d'istruzioni a tre indirizzi non incontrano difficoltà in tal senso, gli altri devono impiegare qualche espediente per aggirare il problema.

Una soluzione comune prevede un'istruzione che effettua il confronto e imposta uno o più bit di condizione per memorizzare il risultato. Un'istruzione successiva può esaminare i bit di condizione e saltare se i due valori messi a confronto si sono rivelati uguali, o diversi, o se il primo era maggiore, e così via. Il Core i7, l'OMAP4430 ARM e l'ATmega168 AVR usano questo approccio.

Il confronto di due numeri implica la comprensione di alcune sottigliezze. Infatti il confronto non è facile quanto una sottrazione: la sottrazione tra un numero positivo molto grande e uno negativo molto grande provoca un overflow, poiché il risultato della sottrazione non può essere rappresentato. L'istruzione di confronto invece deve stabilire se la condizione in esame è soddisfatta o meno e restituire sempre il risultato corretto; un confronto non può provocare un overflow.

Un'altra questione delicata nel confronto tra numeri è stabilire se un numero è da considerarsi con o senza segno. I numeri binari di tre bit possono essere ordinati in uno dei due modi. Dal più piccolo al più grande:

senza segno	con segno
000	100 (più piccolo)
001	101
010	110
011	111
100	000
101	001
110	010
111	011 (più grande)

La colonna di sinistra mostra gli interi positivi da 0 a 7 in ordine crescente. La colonna di destra mostra gli interi con segno in complemento a due che vanno da -4 a +3. Rispondere alla domanda "se 011 sia maggiore di 100" dipende dal modo in cui sono letti i numeri, se con segno o senza segno. La maggior parte degli ISA dispone d'istruzioni per gestire entrambi gli ordinamenti.

5.5.5 Invocazione di procedura

Una procedura è un insieme d'istruzioni che svolge un certo compito e che può essere invocata (chiamata) da diversi punti di un programma. Il termine *subroutine* è usato spesso invece di procedura, soprattutto in riferimento a programmi in linguaggio assemblativo. In C le procedure si chiamano funzioni, anche se non sono propriamente funzioni in senso matematico. Il termine usato in Java è *metodo*. Quando una procedura ha terminato il proprio compito l'esecuzione deve riprendere dall'istruzione successiva alla chiamata. A tal fine l'indirizzo di ritorno deve essere passato alla procedura o salvato da qualche parte dove possa essere recuperato al momento del ritorno.

L'indirizzo di ritorno può essere salvato in tre posti diversi: in memoria, in un registro o nello stack. La soluzione di gran lunga peggiore è metterlo in una locazione di memoria fissa. In questo modo se la procedura chiamasse un'altra procedura, la seconda chiamata causerebbe la perdita dell'indirizzo di ritorno della prima.

Un lieve miglioramento consiste nel far sì che l'istruzione di chiamata di procedura salvi l'indirizzo di ritorno nella prima parola della procedura, cosicché la prima istruzione del codice eseguibile si trovi nella seconda parola. La procedura può quindi rientrare saltando indirettamente tramite la prima parola o anche direttamente, se l'hardware consente l'inserimento dell'opcode per il salto direttamente nella prima parola, insieme all'indirizzo di ritorno. La procedura potrebbe richiamare altre procedure, visto che ciascuna alloca lo spazio per il proprio indirizzo di ritorno. Questo schema fallisce però se la procedura chiama se stessa, perché il primo indirizzo di ritorno verrebbe cancellato dalla seconda chiamata. La capacità di una procedura di chiamare se stessa, detta ricorsione, è di estrema importanza tanto per gli studiosi di programmazione quanto per i programmati. Questo schema fallisce anche se la procedura *A* chiama la procedura *B*, *B* chiama la procedura *C* e *C* chiama *A* (ricorsione indiretta o "a festone", *daisy-chain*). Questo schema di memorizzazione dell'indirizzo di ritorno nella prima parola di una procedura era utilizzato sul CDC 6600, il computer più veloce al mondo per gran parte degli anni '60. Il principale linguaggio utilizzato sul 6600 era il FORTRAN, che proibiva la ricorsione e poteva dunque funzionare. Ma era, ed è tuttora, una pessima idea.

Un miglioramento sostanziale si ottiene se l'istruzione di chiamata di procedura pone l'indirizzo di ritorno in un registro, affidando alla procedura la responsabilità di salvarlo in un posto sicuro. Se la procedura è ricorsiva dovrà preoccuparsi di salvare l'indirizzo di ritorno in un posto differente ogni volta che viene invocata.

La cosa migliore che può fare l'istruzione di chiamata di procedura è porre l'indirizzo di ritorno in cima a uno stack. Quando la procedura termina fa un pop dell'indirizzo di ritorno e lo scrive nel program counter. Con questa forma di chiamata la ricorsione non causa alcun problema particolare; l'indirizzo di ritorno viene salvato automatica-

mente senza sovrascrivere indirizzi di ritorno precedenti. In queste condizioni la ricorsione funziona perfettamente. Abbiamo già visto questa modalità di salvataggio dell'indirizzo di ritorno in JVM (Figura 4.12).

5.5.6 Istruzioni di ciclo

Poiché capita spesso di dover eseguire un gruppo d'istruzioni un numero prefissato di volte, molte macchine dispongono d'istruzioni per facilitare questo compito. Tutti gli schemi prevedono un contatore che viene incrementato o decrementato di un certo valore costante a ogni iterazione del ciclo. Il contatore viene anche esaminato a ogni iterazione; il ciclo termina quando si verifica una certa condizione.

Un metodo possibile è inizializzare il contatore al di fuori del ciclo e quindi cominciare immediatamente l'esecuzione. L'ultima istruzione del ciclo aggiorna il contatore e, se la condizione di terminazione non è stata ancora soddisfatta, torna alla prima istruzione del ciclo. In caso contrario il ciclo termina e si prosegue dalla prima istruzione dopo il ciclo. Questa tipologia di ciclo è detta con valutazione in coda. Nella Figura 5.29(a) ne diamo un esempio in C (non avremmo potuto scriverlo in Java perché non fornisce istruzioni di goto).

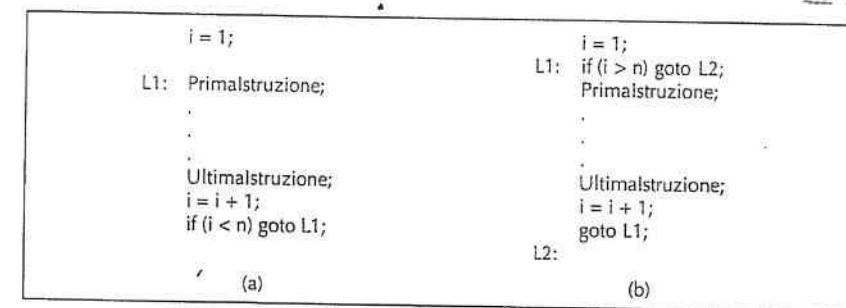


Figura 5.29 (a) Ciclo con verifica della condizione in coda. (b) Ciclo con verifica della condizione in testa.

Il ciclo con valutazione in coda ha la proprietà di essere eseguito sempre almeno una volta, anche se *n* fosse minore a uguale a 0. Considerate come esempio un programma gestore delle schede del personale di una ditta. A un certo punto della sua esecuzione il programma legge i dati di un particolare impiegato e salva in *n* il numero di figli che ha. Quindi esegue *n* volte un ciclo in cui, per ogni figlio, legge il suo nome, il sesso e la data di nascita affinché la ditta possa spedirgli un regalo di compleanno. Se l'impiegato non ha figli, *n* vale 0, eppure il ciclo verrà eseguito una volta e potrebbe spedire erroneamente un regalo di compleanno.

La Figura 5.29(b) mostra un'altra modalità di valutazione che funziona correttamente anche per valori di *n* minori o uguali a 0. Notate come il confronto cambi nelle due

modalità, perciò se l'incremento e la valutazione sono effettuati da un'unica istruzione ISA, i progettisti sono costretti a scegliere una modalità o l'altra.

Considerate il codice che dovrebbe essere generato per l'istruzione

```
for (i = 0; i < n; i++) { istruzioni }
```

Se il compilatore non dispone d'informazioni su n , allora deve usare l'approccio della Figura 5.29(b) per poter gestire correttamente anche i casi in cui $n \leq 0$. Se invece può stabilire che $n > 0$, magari osservando la locazione cui n è stata assegnata, allora può usare il codice migliore della Figura 5.29(a). Il vecchio FORTRAN stabiliva che tutti i cicli dovevano essere eseguiti almeno una volta, proprio per consentire sempre la generazione del codice più efficiente della Figura 5.29(a). Nel 1977 anche la comunità FORTRAN capì che non era una buona idea avere un'istruzione di ciclo dalla semantica stravagante, e che ogni tanto restituiva una risposta errata, solo per risparmiare un'istruzione di salto per ciclo, e corresse il tiro. C e Java non hanno mai avuto questo problema.

5.5.7 Input/Output

Nessun raggruppamento d'istruzioni manifesta la stessa variabilità tra macchine diverse come le istruzioni di I/O. Attualmente i personal computer usano tre schemi diversi di I/O:

1. I/O programmato con attesa attiva;
2. I/O *interrupt driven* (cioè innescato dagli interrupt);
3. I/O con DMA.

Vediamoli in dettaglio. Il metodo di I/O più semplice possibile è quello programmato impiegato di solito nei microprocessori di fascia bassa come, per esempio, i sistemi integrati o i sistemi che devono rispondere rapidamente agli stimoli esterni (sistemi in tempo reale). Queste CPU hanno in genere una sola istruzione di input e una sola di output, che trasferisce un carattere per volta da un prefissato registro al dispositivo di I/O prescelto. Il processore deve eseguire una sequenza prestabilita d'istruzioni per ciascun carattere letto o scritto.

Come semplice esempio di questo metodo, si consideri il terminale con quattro registri di 1 byte mostrato nella Figura 5.30. Due registri sono utilizzati per lo stato e i dati in input, mentre gli altri due sono usati per lo stato e i dati in output. Ogni registro ha un indirizzo unico. Se l'I/O è mappato in memoria, i quattro registri fanno parte dello spazio degli indirizzi della memoria del computer e possono essere letti e scritti mediante le istruzioni ordinarie. In caso contrario ci sono istruzioni speciali di I/O, che chiamiamo IN e OUT, per la lettura e la scrittura dei registri. In entrambi i casi l'I/O avviene tramite il trasferimento dati e del loro stato tra la CPU e questi registri.

Il registro dello stato della tastiera utilizza 2 soli bit degli 8 disponibili: il bit più significativo (il 7) è posto a 1 via hardware ogniqualvolta arriva un carattere; questo evento solleva un interrupt se e solo se il bit 6 era stato asserito precedentemente via software (tratteremo gli interrupt più avanti). Nel caso dell'I/O programmato la CPU aspetta dati in ingresso effettuando un ciclo serrato e ripetuto di letture sul registro di

stato della tastiera, aspettando che il bit 7 assuma il valore 1. Non appena ciò si verifica, il software legge il carattere dal registro buffer della tastiera. La lettura del registro dati della tastiera causa l'azzeramento del bit CARATTERE DISPONIBILE.

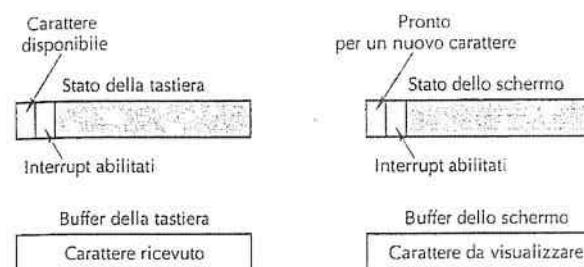


Figura 5.30 Registri dei dispositivi di un semplice terminale.

L'output funziona in modo analogo. Per visualizzare un carattere sullo schermo il software per prima cosa legge il registro di stato dello schermo per assicurarsi che il bit PRONTO valga 1 e in caso negativo si ripete finché il bit non viene asserito a indicare che il dispositivo è pronto ad accettare un carattere. Non appena il terminale è pronto, il software scrive un carattere nel registro di buffer dello schermo, il che provoca la sua trasmissione allo schermo e anche l'azzeramento del bit PRONTO (nel registro di stato dello schermo) da parte del dispositivo. Dopo la visualizzazione del carattere, il controllore imposta automaticamente a 1 il bit PRONTO non appena il dispositivo è preparato a trattare il carattere successivo.

Un esempio di I/O programmato è dato dalla procedura Java nella Figura 5.31. Questa procedura presenta due parametri: un array di caratteri per riportare il risultato e il numero di caratteri (*count*) presenti nell'array (al massimo 1024). Il corpo della procedura è costituito da un ciclo che restituisce un carattere alla volta. Per ogni carattere la CPU deve attendere innanzitutto che il dispositivo sia pronto prima di poter ottenerlo. È ragionevole pensare che le procedure *in* e *out* siano delle routine scritte in linguaggio assemblativo per la lettura e la scrittura dei registri di dispositivo specificati dai rispettivi parametri (il registro di stato per *in*, quello di buffer per *out*). L'implicita divisione per 128, ottenuta per mezzo di uno shift, permette di sbarazzarsi dei 7 bit meno significativi, riportando il bit PRONTO in posizione 0.

Lo svantaggio principale dell'I/O programmato è che la CPU passa gran parte del suo tempo in un ciclo serrato in cui attende che il dispositivo risulti pronto. Questo approccio si chiama **attesa attiva** (*busy waiting*) ed è accettabile se la CPU non ha nulla da fare (sebbene anche un semplice controllore debba spesso monitorare molteplici eventi concorrenti). Tuttavia questa strategia è uno spreco in tutti quei casi in cui ci siano da svolgere altri compiti, quali l'esecuzione di ulteriori programmi, e richiede perciò l'individuazione di un altro metodo di I/O.

```

public static void output_bufier(char buf[], int count) {
    // Spedisce un blocco di dati al dispositivo
    int status, i, ready;
    for (i = 0; i < count; i++) {
        do {
            status = in(display_status_reg);           // ottiene lo stato
            ready = (status >> 7) & 0x01;               // isola il bit PRONTO
        } while (ready != 1);
        out(display_buffer_reg, buf[i]);
    }
}

```

Figura 5.31 Esempio di I/O programmato.

Un modo per evitare l'attesa attiva è far sì che la CPU faccia partire il dispositivo di I/O e gli impedisca l'ordine di generare un interrupt quando ha finito. Possiamo capire come funziona guardando la Figura 5.30. Il software può richiedere all'hardware di segnalargli quando un'operazione di I/O è conclusa tramite l'asserzione del bit INTERRUPT ABILITATI nel registro di dispositivo. Riprenderemo l'analisi degli interrupt più avanti in questo capitolo quando tratteremo il controllo del flusso.

Val la pena far presente che in molti computer il segnale di interrupt viene generato mettendo in AND i bit INTERRUPT ABILITATI e PRONTO. Se il software abilitasse gli interrupt prima di cominciare le operazioni di I/O, verrebbe sollevato immediatamente un interrupt, poiché il bit PRONTO varrebbe 1. Dunque potrebbe essere indispensabile prima far partire il dispositivo, per poi abilitare immediatamente gli interrupt. La scrittura di un byte nel registro di stato non modifica il bit PRONTO, che è di sola lettura.

Benché l'I/O guidato dagli interrupt costituisca un grande progresso rispetto all'I/O programmato, è ben lungi dall'essere perfetto. Il problema è che ci vuole un interrupt per ogni carattere trasferito, e l'elaborazione di un interrupt è gravosa. Occorre un modo per ridurre drasticamente il numero di interrupt.

La soluzione si ottiene tornando all'I/O programmato, ma affidandolo a qualcun altro che non sia la CPU (la soluzione di molti problemi consiste nell'affidare il lavoro a qualcun altro). La Figura 5.32 mostra come fare: aggiungiamo al sistema un nuovo chip, il controllore DMA (*Direct Memory Access*), con accesso diretto al bus.

Il chip DMA ha al suo interno almeno quattro registri accessibili dal software in esecuzione nella CPU: il primo contiene l'indirizzo di memoria di partenza per la lettura o scrittura; il secondo conta il numero di byte (o parole) da trasferire; il terzo specifica il numero di dispositivo o lo spazio d'indirizzamento di I/O da usare, il che determina il dispositivo di I/O desiderato; il quarto stabilisce se i dati vanno letti dal dispositivo di I/O o se vanno scritti su di esso.

Per trasferire un blocco di 32 byte dall'indirizzo di memoria 100 al terminale (sia questo il dispositivo 4) la CPU scrive i numeri 32, 100 e 4 nei primi tre registri DMA, più il codice per la scrittura (poniamo sia 1) nel quarto registro, come illustrato nella Figura 5.32. A questo punto il DMA effettua una richiesta di bus per leggere il byte 100 dalla memoria, analogamente a come farebbe la CPU. Una volta ottenuto il con-

trollore DMA effettuerrebbe una richiesta di I/O al dispositivo 4 finalizzata alla scrittura del byte. Dopo il completamento di queste due operazioni, il controllore DMA incrementa di 1 il suo registro d'indirizzo e decrementa di 1 il suo registro contatore. Se il registro contatore è ancora positivo, si prosegue con la lettura da memoria di un altro byte e con la relativa scrittura nel dispositivo.

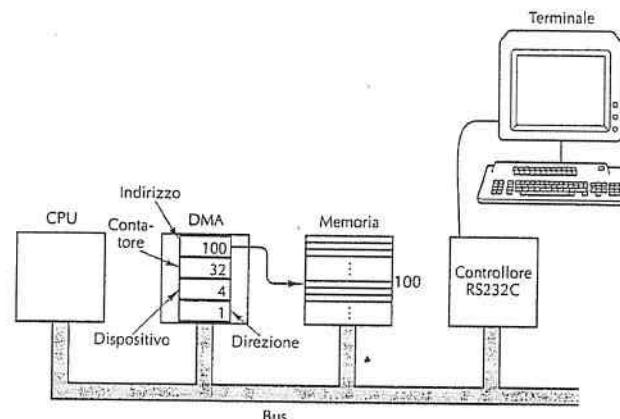


Figura 5.32 Controllore DMA.

Infine, quando il contatore si azzerà, il controllore DMA smette di trasferire dati e manda un impulso sulla linea di interrupt collegata al chip della CPU. In presenza di DMA, la CPU deve solo inizializzare pochi registri, dopo di che è libera di svolgere altri compiti fino al completamento del trasferimento, segnalato da un interrupt proveniente dal controllore DMA. Alcuni controlleri DMA dispongono di due, tre o più insiemi di registri per controllare trasferimenti simultanei.

Anche se con il DMA la CPU viene sollevata dal carico pesante dell'I/O, il procedimento non è del tutto gratuito. Se un dispositivo ad alta velocità, per esempio un disco, è in fase di trasferimento controllato dal DMA, ci vorranno molti cicli di bus per gli accessi alla memoria e al dispositivo. Durante questi cicli la CPU deve restare in attesa (il DMA ha una priorità di bus sempre maggiore di quella della CPU, perché i dispositivi di I/O difficilmente tollerano i ritardi). Il fenomeno che si verifica quando il controllore DMA sottrae cicli di bus alla CPU si dice **appropriazione di cicli** (*cycle stealing*, "furto di cicli"). Nondimeno il guadagno che si ottiene nel non dover gestire un interrupt per byte (o per parola) ripaga largamente del danno derivato dall'appropriazione di cicli.

5.5.8 Istruzioni del Core i7

In questo paragrafo e nei due successivi analizzeremo gli insiemi d'istruzioni delle nostre tre architetture esemplificative: il Core i7, l'OMAP4430, e l'ATmega168. Cia-

scuna di loro dispone di un nucleo d'istruzioni generate normalmente dai compilatori, più un insieme d'istruzioni usate raramente o destinate esclusivamente al sistema operativo. Nella nostra analisi focalizziamo l'attenzione sulle istruzioni comuni. Cominciamo dal Core i7 che è il più complicato. Dopo averlo affrontato sarà tutto in discesa.

L'insieme d'istruzioni del Core i7 è una miscellanea di vere istruzioni di 32 bit e resti dell'8088, suo antenato diretto. Nella Figura 5.33 mostriamo una piccola selezione di comuni istruzioni utilizzabili facilmente dai compilatori e dai programmati odierni. È una lista tutt'altro che completa, visto che non include le istruzioni in virgola mobile, le istruzioni di controllo, né altre di uso meno frequente (tipo l'uso del byte AL come chiave di ricerca). In ogni caso restituiscce una rappresentazione abbastanza fedele delle potenzialità del Core i7.

Molte delle istruzioni del Core i7 referenziano uno o due operandi, che possono trovarsi nei registri o in memoria. Per esempio l'istruzione ADD somma la sorgente alla destinazione, mentre la INC incrementa il suo operando di 1. Alcune istruzioni presentano varianti strettamente connesse tra loro. Per esempio l'istruzione di scorrimento può spostare i bit verso sinistra o verso destra, e può trattare il bit di segno in modo speciale o meno. Gran parte delle istruzioni presenta una varietà di codifiche, che cambia a seconda della natura degli operandi.

I campi SRC della Figura 5.33 sono sorgenti d'informazioni e non vengono modificati. Invece i campi DST sono destinazioni e di norma vengono modificati dall'istruzione. Ci sono alcune regole che stabiliscono che cosa può fungere da sorgente e che cosa da destinazione, che cambiano da istruzione a istruzione senza nessun criterio particolare, ma non ce ne curiamo in questa sede. Molte istruzioni presentano tre varianti, con operandi di 8, 16 e 32 bit. La distinzione tra i tipi avviene tramite l'opcode e/o un bit dell'istruzione. La lista della figura si sofferma soprattutto sulle istruzioni di 32 bit.

Per semplicità abbiamo suddiviso le istruzioni in vari gruppi: il primo contiene le istruzioni per il trasferimento dati all'interno della macchina, tra registri, e locazioni di memoria o di stack. Il secondo gruppo si occupa dell'aritmetica, con o senza segno. Nel caso della moltiplicazione e della divisione, il prodotto o il dividendo di 64 bit sono memorizzati in EAX (la parte meno significativa) e EDX (la parte più significativa).

Il terzo gruppo si occupa dell'aritmetica dei decimali in codifica binaria, detti anche BCD, che tratta ogni byte come due nibble (mezzi byte) di 4 bit. Ogni nibble contiene una cifra decimale (da 0 a 9); le configurazioni da 1010 a 1111 sono inutilizzate. Così un intero di 16 bit può contenere un numero decimale compreso tra 0 e 9999. Questa codifica è poco efficiente, ma non richiede la conversione dell'input da decimale in binario e quindi dell'output in decimale. Queste istruzioni sono usate per l'aritmetica dei numeri BCD, molto utilizzati dai programmi COBOL.

Le istruzioni booleane e quelle di scorrimento/rotazione manipolano in modi diversi i bit all'interno dei byte o delle parole. Ne presentiamo diverse varianti.

I due gruppi successivi trattano test, confronti, e salti basati sui loro risultati. I risultati dei test e dei confronti sono salvati in vari bit del registro EFLAGS. Jxx individua un insieme d'istruzioni di salto condizionato dal risultato del confronto precedente (cioè dai bit di EFLAGS).

Trasferimenti	
MOV DST,SRC	Trasferisce SRC in DST
PUSH SRC	Impila SRC sullo stack
POP DST	Pop di una parola dallo stack e la pone in DST
XCHG DS1, DS2	Scambia DS1 e DS2
LEA DST,SRC	Carica in DST l'indirizzo effettivo di SRC
CMOVcc DST,SRC	Trasferimento condizionato

Trasferimento del controllo	
JMP ADDR	Salta all'indirizzo ADDR
Jxx ADDR	Salti condizionati da EFLAGS
CALL ADDR	Chiama la procedura all'indirizzo ADDR
RET	Ritorno da procedura
IRET	Ritorno da interrupt
LOOPxx	Cicla finché la condizione non sia soddisfatta
INT n	Innesca un interrupt software
INTO	Innesca un interrupt se il bit di overflow = 1

Stringhe	
LODS	Carica una stringa
STOS	Memorizza una stringa
MOVS	Muove una stringa
CMPS	Copia una stringa
SCAS	Scandisce una stringa

Codici di condizione	
STC	Asserisce il bit di riposo in EFLAGS
CLC	Azzera il bit di riposo in EFLAGS
CMC	Complemento del bit di riposo in EFLAGS
STD	Asserisce il bit di direzione in EFLAGS
CLD	Azzera il bit di direzione in EFLAGS
STI	Asserisce il bit di interrupt in EFLAGS
CLI	Azzera il bit di interrupt in EFLAGS
PUSHFD	Impila EFLAGS sullo stack
POPFD	Fa la pop di EFLAGS dallo stack
LAHF	Carica AH da EFLAGS
SAHF	Memorizza AH in EFLAGS

Booleane	
AND DST, SRC	AND di SRC e DST, risultato in DST
OR DST, SRC	OR di SRC e DST, risultato in DST
XOR DST, SRC	OR esclusivo di SRC e DST, risultato in DST
NOT DST	Rimpiazza DST con il suo complemento a uno

Scorrimento/rotazione	
SHL/SAR DST,#	Scorrimento di DST verso s/d di # bit
SHU/SHR DST,#	Scorrimento logico di DST verso s/d di # bit
ROU/ROR DST,#	Rotazione di DST verso s/d di # bit
RCU/RCR DST,#	Rotazione di DST attraverso il riporto di # bit

Test/confronto	
TEST SRC1,SRC2	AND degli operandi, modifica EFLAGS
CMP SRC1,SRC2	Modifica EFLAGS secondo SRC1 - SRC2

#d = sinistra/destra # = numero di scorrimenti/rotazioni
SRC = sorgente LV = puntatore alle variabili locali
DST = destinazione

Figura 5.33 Alcune istruzioni intere del Core i7.

Il Core i7 dispone di parecchie istruzioni per caricamento, memorizzazione, trasferimento, confronto e scansione di stringhe di caratteri o di parole. Queste istruzioni possono essere precedute da un byte speciale chiamato REP che ne causa l'esecuzione ripetuta finché non venga soddisfatta la condizione che ECX, decrementato a ogni iterazione, abbia raggiunto il valore 0. Così facendo è possibile trasferire, confrontare, è così via, blocchi arbitrari di dati. Il gruppo successivo gestisce i codici di condizione.

L'ultimo gruppo è una miscellanea d'istruzioni che non rientrano in nessuna delle categorie precedenti e comprende le conversioni, la gestione dei record d'attivazione dello stack, la terminazione della CPU e l'I/O.

Il Core i7 prevede molti prefissi, tra cui il già menzionato REP. Ognuno di questi prefissi è un byte speciale che può precedere la maggior parte delle istruzioni, analogamente a WIDE nell'IJVM. REP provoca la ripetizione dell'istruzione successiva finché ECX non raggiunga 0, come già detto. REPEZ e REPNZ causano la ripetizione dell'istruzione che precedono fintanto che il codice di condizione Z non viene, rispettivamente, asserito o azzerato. LOCK riserva il bus per tutta la durata dell'esecuzione di un'istruzione, per consentire la sincronizzazione multiprocessore. Altri prefissi vengono usati per forzare un'istruzione in modalità di 16 bit o di 32 bit, il che non solo cambia la lunghezza degli operandi, ma ridefinisce completamente le modalità d'indirizzamento. Infine il Core i7 dispone di un complesso schema di segmentazione del codice, dei dati, dello stack e altro, un altro retaggio dell'8088. Sono previsti alcuni prefissi per forzare l'uso d'indirizzi di memoria in determinati segmenti, ma (fortunatamente) non sono oggetto della nostra analisi.

5.5.9 Istruzioni della CPU ARM OMAP4430

Quasi tutte le istruzioni intere ARM del modo utente che un compilatore può generare sono elencate nella figura 5.34. Non sono incluse nella lista le istruzioni in virgola mobile, le istruzioni di controllo (come quelle per la gestione della cache o il riavvio di sistema), le istruzioni che coinvolgono spazi degli indirizzi diversi da quello dell'utente e le estensioni, come per esempio Thumb. L'insieme è sorprendentemente piccolo: l'ISA ARM OMAP4430 è davvero un RISC.

Le istruzioni LDR e STR sono ovvie, con versioni per 1, 2 e 4 byte. Quando si carica un numero costituito da meno di 32 bit in un registro (di 32 bit), si può decidere se estenderlo con segno o con zero (ci sono istruzioni disponibili a entrambi gli scopi).

Il gruppo successivo riguarda le istruzioni aritmetiche, che possono opzionalmente impostare i bit dei codici di condizione del registro di stato del processore. Sulle macchine CISC sono molte le istruzioni che modificano i codici di condizione, ma su quelle RISC ciò non è auspicabile perché riduce la libertà del compilatore di spostare le istruzioni nel tentativo di riempire gli intervalli di ritardo. Se l'ordine originale delle istruzioni è A ... B ... C, dove A impone alcuni codici di condizione poi esaminati da B, il compilatore non può inserire C tra A e B nel caso C modificasse i codici di condizione. Per questa ragione molte istruzioni vengono fornite in duplice versione, di modo che il compilatore usi preferenzialmente la versione che non modifica i codici di condizione, a meno che abbia intenzione di esaminarne il contenuto in un momento successivo. Il programmatore specifica l'impostazione dei codici di condizione aggiungendo una

"S" in coda al nome dell'istruzione (per esempio ADDS). Un bit dell'istruzione indica al processore che i codici di condizione devono essere impostati. Sono anche supportate le moltiplicazioni e le moltiplicazioni con accumulo.

Il gruppo di scorrimento/rotazione contiene uno scorrimento verso sinistra e due verso destra che operano su un registro di 32 bit. L'istruzione di rotazione a destra implementa una rotazione circolare dei bit del registro, in modo tale che il valore del bit meno significativo diventa il valore del bit più significativo.

Caricamento		Scorrimento/rotazione	
LDRSB DST,ADDR	Carica byte con segno (8 bit)	LSL DST,S1,S2IMM	Scorrimento logico a sinistra
LDRB DST,ADDR	Carica byte senza segno (8 bit)	LSR DST,S1,S2IMM	Scorrimento logico a destra
LDRSH DST,ADDR	Carica mezza parola con segno (16 bit)	ASR DST,S1,S2IMM	Scorrimento aritmetico a destra
LDRH DST,ADDR	Carica mezza parola senza segno (16 bit)	ROR DSR,S1,S2IMM	Rotazione a destra
LDR DST,ADDR	Carica parola (32 bit)		
LDM S1,RECLIST	Carica parole multiple		

Memorizzazione		Booleane	
STRB DST,ADDR	Memorizza byte (8 bit)	TST DST,S1,S2IMM	Test dei bit
STRH DST,ADDR	Memorizza mezza parola (16 bit)	TEQ DST,S1,S2IMM	Test equivalenza
STR DST,ADDR	Memorizza parola (32 bit)	AND DST,S1,S2IMM	AND
STM SRC,RECLIST	Memorizza parole multiple	EOR DST,S1,S2IMM	OR esclusivo
		ORR DST,S1,S2IMM	OR
		BIC DST,S1,S2IMM	Pulizia dei bit

Aritmetica		Trasferimento di controllo	
ADD DST,S1,S2IMM	Somma	Bcc IMM	Salto a PC+IMM
ADD DST,S1,S2IMM	Somma con riporto	BLcc IMM	Salto con link a PC+IMM
SUB DST,S1,S2IMM	Sottrazione	BLcc S1	Salto con link a registro
SUB DST,S1,S2IMM	Sottrazione con riporto		
RSB DST,S1,S2IMM	Sottrazione inversa		
RSC DST,S1,S2IMM	Sottrazione inversa con riporto		
MUL DST,S1,S2	Moltiplicazione		
MLA DST,S1,S2,S3	Moltiplicazione e accumulo		
UMULL D1,D2,S1,S2	Moltiplicazione long senza segno		
SMULL D1,D2,S1,S2	Moltiplicazione long con segno		
UMLAL D1,D2,S1,S2	MLA long senza segno		
SMLAL D1,D2,S1,S2	MLA long con segno		
CMP S1,S2IMM	Confronto e imposta PSR		

Miscellanea	
MOV DST,S1	Trasferimento registro
MOVT DST,IMM	Trasferimento di imm nei bit più significativi
MVN DST,S1	NOT di un registro
MRS DST,PSR	Lettura PSR
MSR PSR,S1	Scrittura PSR
SWP DST,S1,ADDR	Scambio di parola registro/memoria
SWPB DST,S1,ADDR	Scambio di byte registro/memoria
SWI IMM	Interruzione via software

S1 = registro sorgente
 S2IMM = registro sorgente o immediato
 S3 = registro sorgente (quando sono necessari 3 registri)
 DST = registro destinazione
 D1 = registro destinazione (1 di 2)
 D2 = registro destinazione (2 di 2)

ADDR = indirizzo di memoria
 IMM = valore immediato
 RECLIST = lista dei registri
 PSR = registro dello stato del processore
 cc = condizione di salto

Figura 5.34 Le principali istruzioni intere della CPU ARM OMAP4430.

Gli scorrimenti sono usati soprattutto per la manipolazione di bit. Le rotazioni sono utili per operazioni di crittografia e di elaborazione delle immagini. Gran parte delle macchine CISC dispone di un numero ingente d'istruzioni di scorrimento e rotazione, quasi

tutte completamente inutili. Pochi scrittori di compilatori passeranno le notti a piangere la scomparsa!

Il gruppo d'istruzioni booleane è analogo al gruppo delle aritmetiche. Comprende le operazioni AND, EOR, TST, TEQ e BIC. Le ultime tre sono di dubbia utilità, però possono essere svolte in un solo ciclo e richiedono poco hardware supplementare; per questo motivo sono state incluse nel progetto. Anche i progettisti delle macchine RISC di tanto in tanto soccombono alle tentazioni.

Il gruppo successivo d'istruzioni contiene i trasferimenti di controllo. Ecco rappresentata l'insieme delle istruzioni che saltano al verificarsi di varie condizioni. L'istruzione BLcc è simile, nel senso che effettua il salto sotto opportune condizioni, ma in aggiunta deposita l'indirizzo della successiva istruzione nel registro link (R14). Questa istruzione è utile per implementare le chiamate di procedura. A differenza di tutte le altre architetture RISC non esiste una esplicità istruzione di salto a registro. Questa operazione può essere realizzata mediante un'istruzione MOV, impostando come destinazione il program counter (R15).

Sono previste due modalità di chiamata di procedura. La prima istruzione BLcc utilizza il formato di salto della figura 5.14 con uno spiazzamento di 24 bit relativo al program counter. Questo valore è sufficiente a raggiungere un'istruzione entro 32 MB dal chiamante, in entrambe le direzioni. La seconda istruzione BLcc salta all'indirizzo contenuto in un registro specificato e può essere utilizzata per implementare chiamate di procedura con binding dinamico (per esempio, le funzioni virtuali di C++) oppure chiamate oltre il limite dei 32 MB.

L'ultimo gruppo contiene alcune istruzioni di vario genere. MOVT serve per ovviare all'impossibilità di mettere un operando immediato di 32 bit in un registro. Il modo di procedere è usare MOVT per impostare i bit da 16 a 31 e affidare quindi i bit rimanenti all'istruzione successiva grazie al formato immediato. Le istruzioni MRS e MSR permettono lettura e scrittura della parola di stato del processore (PSR). Le istruzioni SWP realizzano scambi atomici tra registri e locazioni di memoria. Queste istruzioni implementano le primitive di sincronizzazione multiprocessore che impareremo nel Capitolo 8. Infine, l'istruzione SWI permette gli interrupt via software (un modo eccessivamente elegante per dire che effettua una chiamata di sistema).

5.5.10 Istruzioni dell'ATmega16 AVR

L'ATmega16 ha un repertorio d'istruzioni semplice, mostrato nella Figura 5.35. Ogni riga specifica il nome abbreviato dell'istruzione, ne dà una descrizione concisa e fornisce un segmento di pseudocodice che chiarisce il funzionamento dell'istruzione. Come ci si poteva attendere, ci sono molte istruzioni MOV per il trasferimento di dati tra i registri. Ci sono istruzioni per le operazioni di push e di pop su di uno stack indirizzato da uno stack pointer (SP) di 16 bit presente in memoria. Si può accedere alla memoria mediante indirizzamento immediato, indiretto basato su registro o indiretto basato su un registro più uno spiazzamento. Per permettere fino a 64 KB di memoria, l'istruzione di caricamento con un indirizzo immediato è un'istruzione di 32 bit. La modalità di indirizzamento indiretto utilizza le coppie di registri X, Y e Z, che combinano due registri di 8 bit per formare un singolo puntatore di 16 bit.

Istruzione	Descrizione	Semantica
ADD DST,SRC	Somma	DST \rightarrow DST + SRC
ADC DST,SRC	Somma con riporto	DST \rightarrow DST + SRC + C
ADIW DST,IMM	Somma di immediato a una parola	DST + 1:DST \rightarrow DST + 1:DST + IMM
SUB DST,SRC	Sottrazione	DST \rightarrow DST - SRC
SUBI DST,IMM	Sottrazione di immediato	DST \rightarrow DST - IMM
SBC DST,SRC	Sottrazione con riporto	DST \rightarrow DST - SRC - C
SBCI DST,IMM	Sottrazione di immediato con riporto	DST \rightarrow DST - IMM - C
SBIW DST,IMM	Sottrazione di immediato a una parola	DST + 1:DST \rightarrow DST + 1:DST - IMM
AND DST,SRC	AND logico	DST \rightarrow DST AND SRC
ANDI DST,IMM	AND logico con immediato	DST \rightarrow DST AND IMM
OR DST,SRC	OR logico	DST \rightarrow DST OR SRC
ORI DST,IMM	OR logico con immediato	DST \rightarrow DST OR IMM
EOR DST,SRC	OR esclusivo	DST \rightarrow DST XOR SRC
COM DST	Complemento a uno	DST \rightarrow 0xFF - DST
NEG DST	Complemento a due	DST \rightarrow 0x00 - DST
SBR DST,IMM	Imposta 1 bit in un registro	DST \rightarrow DST OR IMM
CBR DST,IMM	Cancella 1 bit in un registro	DST \rightarrow DST AND (0xFF - IMM)
INC DST	Incremento	DST \rightarrow DST + 1
DEC DST	Decremento	DST \rightarrow DST - 1
TST DST	Verifica per minore o uguale a zero	DST \rightarrow DST AND DST
CLR DST	Cancella il contenuto di un registro	DST \rightarrow DST XOR DST
SER DST	Imposta registro	DST \rightarrow 0xFF
MUL DST,SRC	Moltiplicazione senza segno	R1:R0 \rightarrow DST * SRC
MULS DST,SRC	Moltiplicazione con segno	R1:R0 \rightarrow DST * SRC
MULSU DST,SRC	Moltiplicazione con segno con operando senza segno	R1:R0 \rightarrow DST * SRC
RIMP IMM	Salto relativo a PC	PC \rightarrow PC + IMM + 1
IJMP	Salto indiretto a Z	PC \rightarrow Z (R30:R31)
JMP IMM	Salto	PC \rightarrow IMM
RCALL IMM	Chiamata relativa	STACK \rightarrow PC+2, PC \rightarrow PC + IMM + 1
ICALL	Chiamata indiretta (Z)	STACK \rightarrow PC+2, PC \rightarrow Z (R30:R31)
CALL	Chiamata	STACK \rightarrow PC+2, PC \rightarrow IMM
RET	Ritorno	PC \rightarrow STACK
CP DST,SRC	Confronto	DST - SRC
CPC DST,SRC	Confronto con riporto	DST - SRC - C
CPI DST,IMM	Confronto con immediato	DST - IMM
BRCC IMM	Salto condizionato	If ccttrue PC \rightarrow PC + IMM + 1
MOV DST,SRC	Copia di registro	DST \rightarrow SRC
MOVV DST,SRC	Copia di coppia di registri	DST + 1:DST \rightarrow SRC + 1:SRC
LDI DST,IMM	Caricamento immediato	DST \rightarrow IMM
LDS DST,IMM	Caricamento diretto	DST \rightarrow MEM(IMM)
LD DST,XYZ	Caricamento indiretto	DST \rightarrow MEM(XYZ)
LDD DST,XYZ+IMM	Caricamento indiretto con spiazzamento	DST \rightarrow MEM(XYZ+IMM)
STS IMM,SRC	Memorizzazione diretta	MEM(IMM) \rightarrow SRC
ST XYZ,SRC	Memorizzazione indiretta	MEM(XYZ) \rightarrow SRC
STD XYZ+IMM,SRC	Memorizzazione indiretta con spiazzamento	MEM(XYZ+IMM) \rightarrow SRC
PUSH REGIST	Push di un registro sullo stack	STACK \rightarrow REGIST
POP REGIST	Pop di un registro dallo stack	REGIST \rightarrow STACK
LSL DST	Scorrimento logico a sinistra di una posizione	DST \rightarrow DST LSL 1
LSR DST	Scorrimento logico a destra di una posizione	DST \rightarrow DST LSR 1
ROL DST	Rotazione a sinistra di una posizione	DST \rightarrow DST ROL 1
ROR DST	Rotazione a destra di una posizione	DST \rightarrow DST ROR 1
ASR DST	Scorrimento aritmetico a destra di una posizione	DST \rightarrow DST ASR 1

SRC = registro sorgente
DST = registro destinazione
IMM = IMM = valore immediato

XYZ = X, Y o Z, coppie di registri
Accesso alla memoria di tipo A

Figura 5.35 Le istruzioni dell'ATmega16 AVR.

L'ATmega168 dispone di semplici istruzioni aritmetiche per somma, sottrazione e moltiplicazione, dove le ultime due usano due registri. Sono disponibili anche istruzioni d'incremento e decremento, usate di frequente. Ci sono anche le istruzioni booleane, di scorrimento e di rotazione. Le istruzioni di salto e di chiamata possono avere come destinazione un indirizzo immediato, un indirizzo relativo al program counter o un indirizzo contenuto nella coppia di registri Z.

5.5.11 Insiemi d'istruzioni a confronto

I tre esempi di insiemi d'istruzioni che abbiamo visto sono molto diversi tra loro. Il Core i7 è una classica macchina CISC a 32 bit e a due indirizzi, con una lunga storia, modalità d'indirizzamento peculiari e molto irregolari, nonché con molte istruzioni per l'accesso in memoria. La CPU ARM OMAP4430 è una moderna macchina RISC a 32 bit e a tre indirizzi, con architettura load/store, a malapena qualche modalità d'indirizzamento e un insieme d'istruzioni efficiente e compatto. L'ATmega168 è un minuscolo processore integrato, progettato per essere realizzato su un solo chip.

Se ogni macchina è fatta a modo proprio ci sono delle buone ragioni. Il progetto del Core i7 risponde a tre esigenze principali:

1. retrocompatibilità;
 2. retrocompatibilità;
 3. retrocompatibilità.

Considerando lo stato dell'arte attuale, a nessuno verrebbe in mente di progettare una macchina così irregolare e con così pochi registri, tutti diversi tra loro. Queste caratteristiche complicano la scrittura dei compilatori. La mancanza di registri inoltre costringe i compilatori a riversare continuamente le variabili in memoria per poi ricaricarle, un lavoro gravoso anche se si hanno due o tre livelli di cache. La velocità del Core i7 è una riprova delle capacità degli ingegneri dell'Intel, a dispetto dei vincoli posti dall'ISA. Eppure abbiamo visto nel capitolo precedente quanto sia complessa la sua implementazione.

L'OMAP4430 rappresenta un progetto di ISA allo stato dell'arte. Ha un ISA di 32 bit, ha molti registri e un insieme d'istruzioni che predilige le operazioni a tre registri, più un piccolo gruppo d'istruzioni LOAD e STORE. Tutte le istruzioni sono della stessa lunghezza, anche se il numero di formati è sfuggito un po' di mano. Ciononostante l'implementazione resta semplice ed efficiente. Gran parte dei nuovi progetti tende a somigliare all'architettura ARM OMAP4430, ma con meno formati d'istruzione.

L'ATmega16 AVR offre un insieme d'istruzioni semplice e abbastanza regolare, relativamente con poche istruzioni e poche modalità d'indirizzamento. Si distingue per la presenza di 32 registri di 8 bit, per l'accesso rapido ai dati, per la capacità di rendere registri accessibili nello spazio di memoria e per le istruzioni di manipolazione dei bit sorprendentemente potenti. Il suo vanto principale è di richiedere un numero molto limitato di transistor per l'implementazione, il che consente di ospitarne un gran numero su di una sola piastra, dando luogo a un costo unitario di CPU molto contenuto.

5.6 Controllo del flusso

Il controllo del flusso riguarda la sequenza con cui le istruzioni vengono eseguite dinamicamente, ovvero durante l'esecuzione del programma. In mancanza d'istruzioni di salto o di chiamate di procedura, le istruzioni vengono eseguite di norma nello stesso ordine con cui si susseguono in memoria. Le chiamate di procedura alterano il controllo del flusso, perché arrestano la procedura correntemente in esecuzione e cominciano l'esecuzione della procedura chiamata. Le coroutine sono legate alle procedure e causano un'alterazione simile (sono utili nella simulazione di processi paralleli). Anche le trap e gli interrupt provocano un'alterazione del flusso esecutivo quando si verificano certe condizioni speciali. Sono tutti argomenti trattati nei paragrafi successivi.

5.6.1 Flusso sequenziale e diramazioni

Molte istruzioni non alterano il controllo del flusso; dopo l'esecuzione di un'istruzione, viene recuperata ed eseguita l'istruzione che la segue in memoria. Al termine di ciascuna istruzione, il program counter viene incrementato della lunghezza dell'istruzione elaborata. Se lo si osserva per un tempo abbastanza lungo, rispetto al tempo di esecuzione di una singola istruzione, il program counter è una funzione lineare nella variabile tempo e che aumenta, in ogni unità di tempo, di una quantità pari alla lunghezza media delle istruzioni diviso il loro tempo medio di esecuzione. In altre parole, l'ordine dinamico secondo cui il processore esegue le istruzioni è quello con cui appaiono nel listato del programma, come mostrato dalla Figura 5.36(a). Se un programma contiene salti, questa semplice relazione tra ordinamento delle istruzioni in memoria e ordine di esecuzione non vale più. In presenza di salti il program counter non è più una funzione monotona crescente nel tempo, come si evince dalla Figura 5.36(b). In ragione di ciò, diventa difficile visualizzare la sequenza di esecuzione delle istruzioni a partire dal listato.

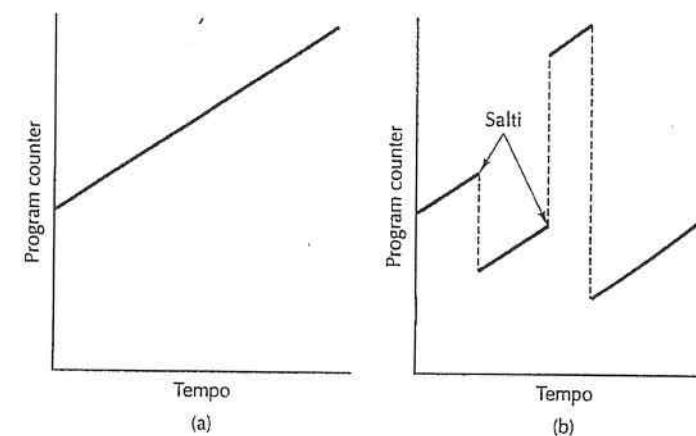


Figura 5.36 Valore del program counter in funzione del tempo. (a) Senza salti. (b) Con salti.

Quando i programmati incontrano difficoltà nel tener traccia della sequenza di esecuzione delle istruzioni da parte del processore, commettono errori con più facilità. Questa osservazione ha portato Dijkstra a scrivere una nota (1968a), allora controversa, dal titolo "GO TO Statement Considered Harmful" ("L'istruzione GOTO è da considerarsi dannosa"), in cui suggeriva di evitare le istruzioni di goto.

Da quel testo è scaturita la rivoluzione della programmazione strutturata, tra i cui dogmi c'è la sostituzione di tutti i comandi goto con forme di controllo più strutturate, quali i cicli. Naturalmente quando questi programmi vengono compilati in programmi di livello 2, questi ultimi possono contenere molti salti, dal momento che l'implementazione di if, while e di altre strutture di controllo di alto livello richiede la capacità di saltare da una parte all'altra.

5.6.2 Procedure

Le procedure costituiscono la tecnica più importante per la strutturazione dei programmi. Da un certo punto di vista una chiamata di procedura altera il flusso esecutivo tanto quanto un salto, ma a differenza di un salto alla terminazione del suo compito la procedura ripassa il controllo al comando o all'istruzione che segue la sua chiamata.

Tuttavia, da un altro punto di vista, il corpo di una procedura può essere visto come la definizione di una nuova istruzione di alto livello. Così facendo, la chiamata di procedura è concepibile come un'istruzione singola, senza riguardo alla sua effettiva complessità. Per la comprensione di una porzione di codice contenente una chiamata di procedura è sufficiente sapere *che cosa fa*, non *come lo fa*.

Una tipologia di procedure particolarmente interessanti è quella delle **procedure ricorsive**, ovvero quelle procedure che richiamano se stesse direttamente o indirettamente attraverso una successione di altre procedure. Lo studio delle procedure ricorsive favorisce una profonda comprensione del modo in cui sono implementate le chiamate di procedura, e della natura delle variabili locali. Diamo ora un esempio di procedura ricorsiva.

Quello della "torre di Hanoi" è un vecchio problema che ammette una semplice soluzione basata sulla ricorsione. In un monastero di Hanoi ci sono tre pioli d'oro: sul primo sono impilati, attraverso un foro apposito, 64 dischi d'oro. I dischi sono di dimensioni diverse e ciascuno è poggiato su di un disco di diametro maggiore del proprio. Gli altri due pioli sono inizialmente vuoti. I monaci del monastero hanno il compito di trasferire tutti i dischi (uno alla volta) sul terzo piolo, ma senza mai poggiare un disco di dimensioni maggiori su uno più piccolo. Si dice che quando avranno evaso il loro compito, il mondo avrà fine. Se volete sperimentare direttamente potete provare a usare dei dischi qualunque (e meno di 64), ma una volta che avrete risolto il problema non aspettatevi che succeda alcunché. Per ottenere l'effetto "fine del mondo" dovete usarne 64 e tutti d'oro. La Figura 5.37 mostra la configurazione iniziale con 5 dischi.

La soluzione consiste nel muovere gli n dischi dal piolo 1 al piolo 3 spostandone prima $n - 1$ dal piolo 1 al 2, quindi impilando il disco rimanente sul piolo 3 e infine spostando gli $n - 1$ dischi del piolo 2 sul piolo 3. La Figura 5.38 illustra questa soluzione (per $n = 3$).

Per risolvere il problema abbiamo bisogno di una procedura per spostare n dischi dal piolo i al piolo j . La sua invocazione

```
towers(n, i, j)
```

produce la visualizzazione della soluzione. Per prima cosa la procedura effettua il test $n = 1$. Se così fosse la soluzione sarebbe banale, basterebbe spostare il disco da i a j . Se $n \neq 1$ allora la soluzione consiste nelle tre fasi già illustrate, ciascuna delle quali comporta una chiamata di procedura ricorsiva.

La soluzione completa è data dalla Figura 5.39. La chiamata

```
towers(3, 1, 3)
```

che risolve il problema della Figura 5.38, provoca altre tre chiamate, e cioè

```
towers(2, 1, 2)
towers(1, 1, 3)
towers(2, 2, 3)
```

La prima e la terza causano a loro volta tre chiamate, per un totale di sette.

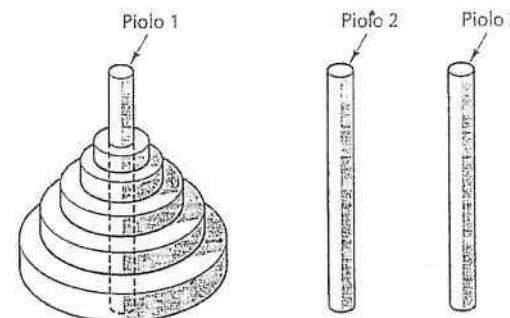


Figura 5.37 Configurazione iniziale del problema della torre di Hanoi con cinque dischi.

Al fine di poter gestire le procedure ricorsive abbiamo bisogno di uno stack per memorizzare i parametri e le variabili locali a ogni invocazione, analogamente a quanto mostrato nel caso dell'IJVM. Ogni volta che una procedura viene chiamata, in cima allo stack viene allocato un record d'attivazione per la procedura stessa. Il record d'attivazione di creazione più recente è quello in uso corrente. Nei nostri esempi lo stack cresce verso l'alto, dagli indirizzi di memoria più piccoli ai più grandi, proprio come nell'IJVM. Perciò il record d'attivazione più recente è caratterizzato da un indirizzo maggiore di tutti gli altri. Oltre al puntatore allo stack, che punta alla cima della pila, risulta spesso conveniente avere un puntatore al record d'attivazione, FP (*frame pointer*), che punta a una locazione nota del record d'attivazione (per esempio il puntatore di collegamento, come nell'IJVM, o la prima variabile locale).

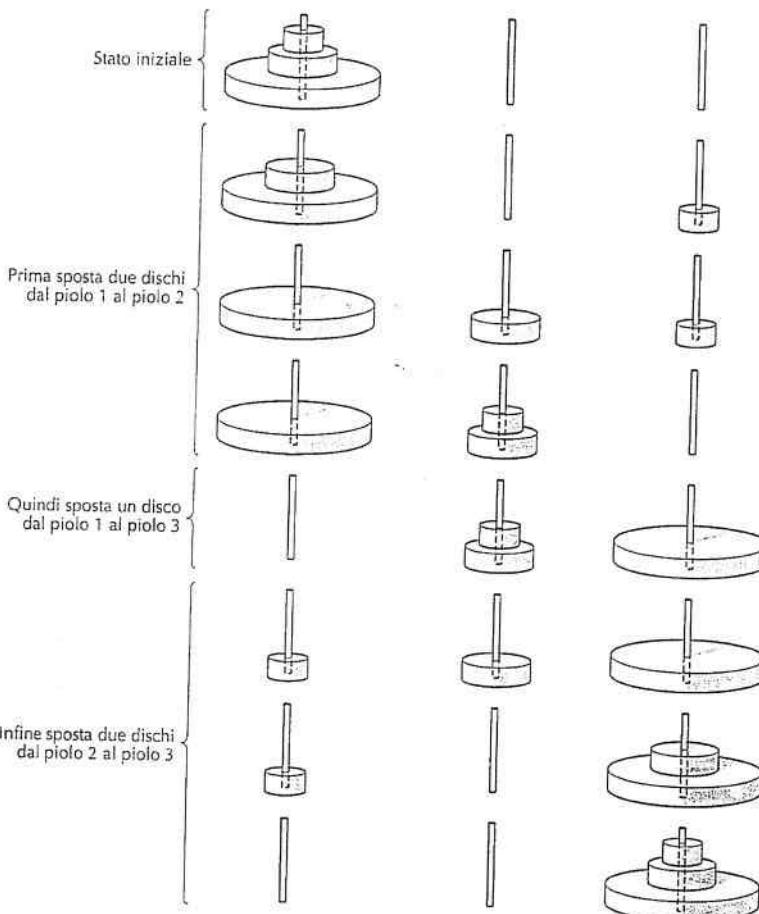


Figura 5.38 Passi necessari per la soluzione del problema della torre di Hanoi con tre dischi.

La Figura 5.40 mostra il record d'attivazione di una macchina con parole di 32 bit. La chiamata originale `towers` impila `n`, `i` e `j` in cima allo stack ed esegue quindi un'istruzione `CALL` che impila l'indirizzo di ritorno sullo stack, all'indirizzo 1012. Al momento dell'ingresso, la procedura memorizza nello stack il vecchio valore di `FP` alla locazione 1016 e incrementa il puntatore allo stack per allocare spazio sufficiente alle variabili locali. Poiché c'è una sola variabile locale di 32 bit (`k`), `SP` è incrementato di 4 in 4 fino a 1020. La Figura 5.40(a) mostra la situazione dello stack al termine di queste operazioni.

```
public void towers(int n, int i, int j) {
    int k;
    if (n == 1)
        System.out.println("Sposta un disco da " + i + " a " + j);
    else {
        k = 6 - i - j;
        towers(n - 1, i, k);
        towers(1, i, j);
        towers(n - 1, k, j);
    }
}
```

Figura 5.39 Procedura per la soluzione del problema della torre di Hanoi.

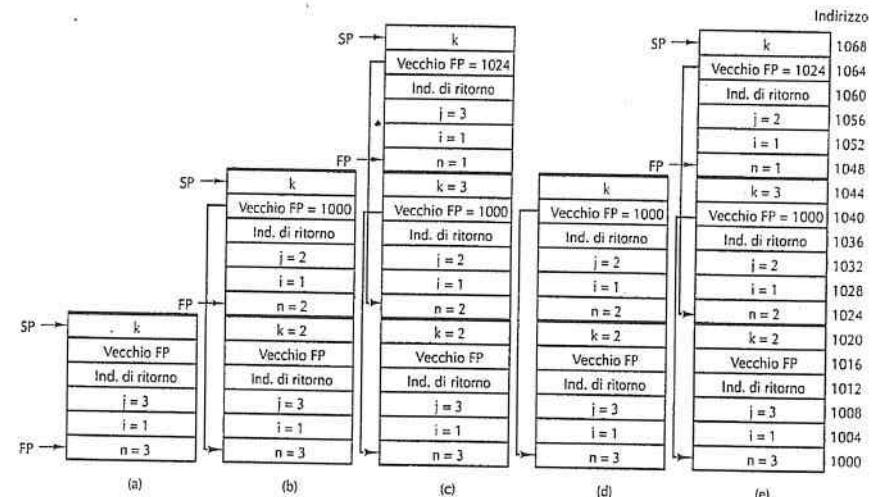


Figura 5.40 Lo stack in alcuni momenti dell'esecuzione del codice della Figura 5.39.

Le prime cose che una procedura invocata deve fare sono il salvataggio del vecchio `FP` (così che possa essere ripristinato all'uscita della procedura), la copia di `SP` in `FP` e l'eventuale incremento di `FP` di una parola, a seconda della locazione cui puntare nel nuovo record d'attivazione. In questo esempio `FP` punta alla prima variabile locale, ma nell'IJVM LV indirizzava il puntatore di collegamento. La gestione del puntatore al record d'attivazione può essere leggermente diversa su macchine distinte, a seconda che abbia l'indirizzo della base del record d'attivazione, come nella Figura 5.40, o della sua cima oppure di altro. A tal fine è utile confrontare le Figure 5.40 e 4.12 per comprendere due modi alternativi di gestire il puntatore di collegamento. Ci sono anche altri modi

possibili. Comunque sia, il punto chiave è poter effettuare un ritorno da procedura e ripristinare lo stato dello stack alla situazione in cui si trovava appena prima dell'inizio della procedura corrente.

Il codice che si preoccupa di salvare il vecchio FP, che stabilisce il nuovo FP e accresce il puntatore allo stack per riservare spazio alle variabili locali si chiama **prologo della procedura** (*procedure prolog*). All'uscita dalla procedura si rende invece necessario ripulire lo stack, che rappresenta la fase di **epilogo della procedura**. Due delle caratteristiche più importanti di ogni computer sono la brevità e la velocità dei meccanismi di prologo ed epilogo delle procedure. Se sono lunghi e lenti, le chiamate di procedura ne soffriranno e i programmati devoti all'efficienza impareranno a evitare di scrivere molte procedure brevi, cui preferiranno programmi lunghi, monolitici e poco strutturati. Le istruzioni **ENTER** e **LEAVE** del Core i7 sono state progettate proprio per far funzionare i prologhi e gli epiloghi delle procedure in modo efficiente. Naturalmente queste istruzioni hanno un loro modello di gestione del puntatore al record d'attivazione, per cui se il compilatore ha un modello diverso deve fare a meno di usarle.

Torniamo ora al problema della torre di Hanoi. Ogni chiamata di procedura aggiunge un nuovo record d'attivazione allo stack che verrà rimosso al suo ritorno. Per illustrare l'uso dello stack nell'implementazione delle procedure ricorsive, tracciamo le chiamate a partire da

```
towers(3, 1, 3)
```

La Figura 5.40(a) mostra lo stack appena prima della chiamata di questa procedura. Per prima cosa la procedura controlla il valore di n , e quando scopre che $n = 3$, assegna k e opera la chiamata

```
towers(2, 1, 2)
```

Al completamento di questa chiamata lo stack è quello della Figura 5.40(b) e la procedura ricomincia dall'inizio (a ogni chiamata di procedura l'esecuzione riparte dall'inizio). Anche questa volta il confronto n è diverso da 1, perciò la procedura inizializza il proprio k e chiama

```
towers(1, 1, 3)
```

Ora lo stack è nella situazione della Figura 5.40(c) e il program counter punta all'inizio della procedura. Questa volta però $n = 1$ e viene visualizzata una riga di risultato. Dopo di che la procedura rientra e rimuove un record d'attivazione, ripristinando FP e SP come indicato nella Figura 5.43(d). Successivamente l'esecuzione continua dall'indirizzo di ritorno, che è la seconda chiamata:

```
towers(1, 1, 2)
```

Questa provoca l'aggiunta di un nuovo record allo stack come mostrato nella Figura 5.40(e). Quindi viene visualizzata una nuova riga di risultato e il record d'attivazione è rimosso dallo stack. Le chiamate di procedura proseguono con questo andamento, finché non viene completata l'esecuzione della chiamata originaria, al che viene rimosso

dallo stack anche il record d'attivazione della Figura 5.40(a). Per capire davvero come funziona la ricorsione consigliamo di simulare con carta e penna l'intera esecuzione di

```
towers(3, 1, 3)
```

5.6.3 Coroutine

Nell'usuale sequenza di chiamata c'è una chiara distinzione tra procedura chiamante e procedura chiamata. Si consideri la Figura 5.41, dove la procedura A chiama la procedura B.

La procedura B svolge dei calcoli, e restituisce il controllo all'altra. A prima vista sembrerebbe una situazione simmetrica, perché né A né B costituiscono il programma principale, ma due procedure (la procedura A potrebbe essere stata invocata dal programma principale, ma ciò è irrilevante). Il controllo passa da A a B, durante la chiamata, mentre durante il rientro è trasferito da B ad A.

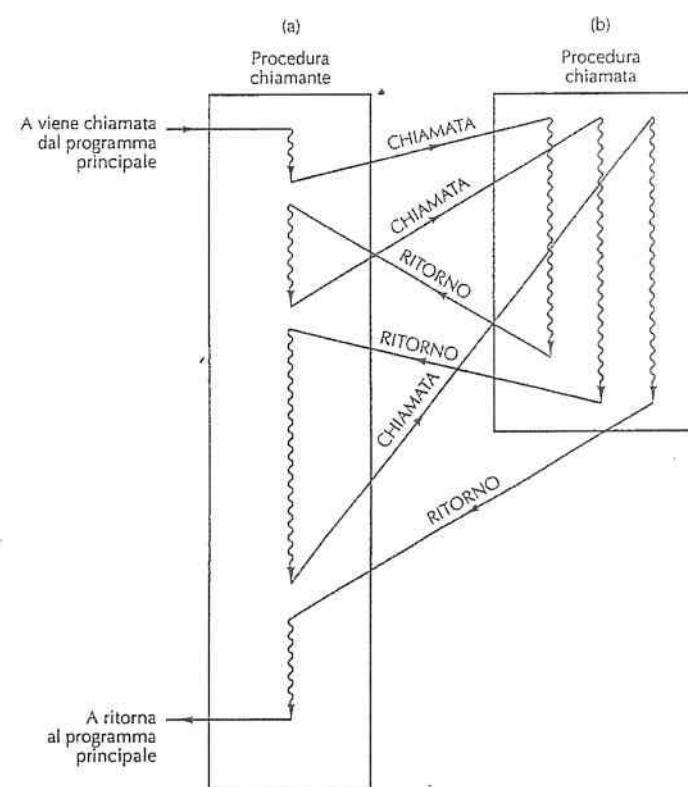


Figura 5.41 L'esecuzione di una procedura invocata inizia sempre dalla sua prima istruzione.

L'asimmetria nasce dal fatto che, nel primo caso l'esecuzione della procedura *B* comincia dal suo inizio, mentre nel secondo caso l'esecuzione non comincia dall'inizio di *A*, ma dall'istruzione immediatamente successiva alla chiamata. Se poi *A* chiama ancora *B*, l'esecuzione riparte dall'inizio di *B*, non dall'istruzione che segue il ritorno precedente. Ogniqualvolta *A* chiama *B* quest'ultima ricomincia dall'inizio, mentre *A* non ricomincia mai da capo, ma continua semplicemente ad andare avanti.

Questa differenza si riflette nel modo in cui il controllo è passato tra *A* e *B*. Per chiamare *B*, la procedura *A* si avvale dell'istruzione di chiamata di procedura, che memorizza l'indirizzo di ritorno (cioè l'indirizzo dell'istruzione che segue la chiamata) in un qualche luogo adatto allo scopo, per esempio in cima allo stack. Quindi pone l'indirizzo di *B* nel program counter, completando così la chiamata. Quando *B* ritorna, non usa l'istruzione di chiamata ma si avvale dell'istruzione di ritorno, che semplicemente fa il pop dell'indirizzo di ritorno dallo stack e lo copia nel program counter.

Alle volte è utile avere due procedure che si chiamano a vicenda nel modo illustrato nella Figura 5.42. Come in precedenza, quando *B* ritorna ad *A*, *B* salta all'istruzione appena successiva alla sua chiamata. Quando è invece *A* a passare il controllo a *B*, non salta all'inizio di *B* (tranne la prima volta) bensì all'istruzione successiva al "ritorno" più recente, vale a dire successiva alla chiamata di *A* più recente. Due procedure che si comportano in questo modo prendono il nome di *coroutine*.

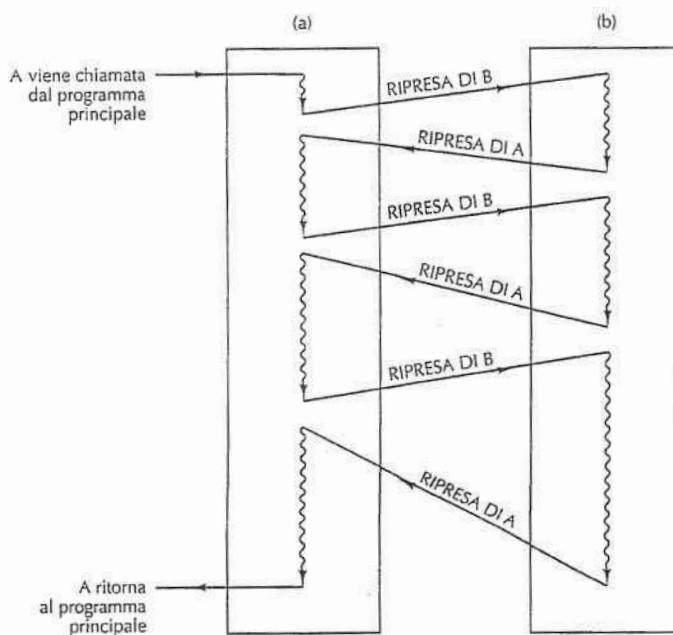


Figura 5.42 Alla ripresa di una coroutine, la sua esecuzione comincia da dove era stata interrotta, non dall'inizio.

Le coroutine sono usate comunemente per la simulazione dell'elaborazione parallela su singola CPU. Ogni coroutine gira in maniera pseudo-parallela alle altre, come se disponesse di una propria CPU. Questo stile di programmazione semplifica la scrittura di alcune applicazioni; inoltre è utile per la validazione del software destinato a girare su sistemi multiprocessore.

Le istruzioni CALL e RETURN non funzionano per le chiamate di coroutine perché, anche se l'indirizzo del salto viene recuperato dallo stack come per un'istruzione di ritorno, in questo caso è la stessa chiamata di coroutine a salvare l'indirizzo di ritorno per un successivo ritorno a essa. Sarebbe bello se esistesse un'istruzione per scambiare la cima dello stack con il program counter. In particolare questa istruzione effettuerebbe per prima cosa il pop del vecchio indirizzo di ritorno dallo stack verso un registro interno, quindi eseguirebbe una push del program counter sullo stack e infine copierebbe il contenuto del registro interno nel program counter. Dal momento che verrebbero effettuate sullo stack una push e un pop di una parola ciascuna, il puntatore allo stack non si sposterebbe. Questa istruzione si riscontra di rado, perciò viene spesso simulata per mezzo di numerose istruzioni.

5.6.4 Trap

Una trap ("trappola") è una specie di chiamata di procedura automatica effettuata quando si verificano certe condizioni causate da un programma e che sono in genere eventi rilevanti, ma di rara occorrenza. Ne è un buon esempio l'overflow: in molti computer, se il risultato di un'operazione aritmetica eccede il più grande numero rappresentabile, si verifica una trap, ovvero il controllo del flusso viene interrotto e riprende da una locazione di memoria prefissata, invece di proseguire in sequenza. In tale locazione di memoria si trova l'indirizzo per un salto a una procedura detta gestore di trap, che svolge le azioni appropriate al caso, come la stampa di un messaggio di errore. Viceversa non si verifica nessuna trap se il risultato di un'operazione è nell'intervallo dei numeri rappresentati dalla macchina.

Il concetto chiave delle trap è che sono fatte scattare da condizioni eccezionali causate dal programma stesso e rilevate dall'hardware o dal microprogramma. Un metodo alternativo per la gestione dell'overflow è di avere un registro di 1 bit asserito ogniqualvolta si verifica un overflow. Se un programmatore vuole controllare la presenza di un overflow deve includere un'istruzione esplicita di "salto se overflow asserito" dopo ogni istruzione aritmetica. Si tratta di una soluzione dispendiosa sia in tempo sia in spazio. Le trap fanno risparmiare tempo di esecuzione e memoria rispetto alla verifica affidata al controllo esplicito del programmatore.

Le trap potrebbero essere implementate tramite un test esplicito effettuato dal microprogramma (o dall'hardware); se viene rilevato un overflow, l'indirizzo di trap è caricato nel program counter. Ciò che fa scattare una trap a un certo livello potrebbe essere tenuto sotto controllo da un programma a un livello più basso. Il test effettuato dal microprogramma è ancora conveniente in termini di tempo rispetto al test svolto dal programmatore, perché può essere sovrapposto a qualche altra operazione. Inoltre consente un risparmio di memoria, perché basta che sia effettuato una sola volta, per esem-

pio nel ciclo principale del microprogramma, indipendentemente dal numero d'istruzioni aritmetiche che ci sono nel programma principale.

Alcune delle condizioni che causano comunemente trap sono gli overflow e gli underflow (interi o in virgola mobile), le violazioni di protezione, gli opcode non definiti, gli overflow di stack, i tentativi di utilizzare dispositivi di I/O inesistenti, i tentativi di fetch di una parola da un indirizzo dispari, la divisione per zero.

5.6.5 Interrupt

Gli interrupt ("interruzioni") sono cambiamenti nel flusso esecutivo causati non dal programma in esecuzione, ma da qualche altro problema, in genere determinato dall'I/O. Per esempio un programma potrebbe ordinare al disco di iniziare il trasferimento delle informazioni e di inviare un interrupt non appena il trasferimento termini. Come le trap, gli interrupt interrompono il programma in esecuzione e trasferiscono il controllo a un gestore deputato a svolgere le azioni appropriate. Al loro compimento, il gestore di interrupt restituisce il controllo al programma interrotto. È suo compito far riprendere il processo interrotto esattamente dallo stesso stato in cui si trovava al momento dell'interruzione, il che implica il ripristino di tutti i registri interni allo stato precedente all'interrupt.

La differenza essenziale tra le trap e gli interrupt è che le *trap* sono sincrone al programma e gli *interrupt* sono asincroni. Se un programma viene rieseguito un milione di volte con lo stesso input, le trap si verificheranno sempre nello stesso punto, mentre gli interrupt possono variare a seconda, per esempio, del momento in cui viene premuto il tasto d'invio al terminale. La ragione alla base della riproducibilità delle trap e all'irriproducibilità degli interrupt è che le trap sono causate direttamente dal programma, gli interrupt, invece, sono causati dal programma per lo più indirettamente.

Per comprenderne meglio il funzionamento consideriamo un tipico esempio di interrupt: un calcolatore deve scrivere sullo schermo una riga di caratteri. Per prima cosa il software di sistema raccoglie in un buffer tutti i caratteri da scrivere, inizializza una variabile globale *ptr* in modo che punti all'inizio del buffer e quindi assegna a una seconda variabile globale *count* il numero di caratteri da visualizzare. Successivamente verifica che il terminale sia pronto e, in caso affermativo, invia il primo carattere (per mezzo di registri come quelli della Figura 5.30). Dopo aver fatto partire l'attività di I/O, la CPU è libera di eseguire un altro programma o di svolgere lavoro di altro genere.

Nel volgere di poco tempo, il carattere viene visualizzato e l'interrupt può essere lanciato. In forma semplificata, i passi si succedono nell'ordine seguente.

AZIONI HARDWARE

- Il controllore del dispositivo attiva una linea di interrupt sul bus di sistema per dar via alla sequenza di interrupt.
- Non appena pronta a gestire l'interrupt, la CPU attiva sul bus un segnale di conferma (acknowledgment) dell'interrupt.
- Quando il controllore del dispositivo vede confermata la ricezione del proprio segnale di interrupt, invia sulla linea dati un piccolo intero che lo identifica. Questo numero si chiama **vettore di interrupt**.

- La CPU preleva il vettore di interrupt dal bus e lo salva temporaneamente.
- La CPU impila il program counter e il registro PSW sullo stack.
- Quindi la CPU usa il vettore di interrupt come indice per individuare il nuovo program counter all'interno di una tabella posta all'inizio della memoria. Per esempio, se il program counter è di 4 byte, il vettore di interrupt *n* corrisponde all'indirizzo 4 *n*. Questo nuovo program counter punta all'inizio della routine di servizio dell'interrupt per il controllore che l'ha lanciato. Spesso anche PSW viene caricato o modificato (per esempio per disabilitare ulteriori interrupt).

AZIONI SOFTWARE

- La routine di servizio dell'interrupt comincia con il salvare (sullo stack o in una tabella di sistema) tutti i registri che utilizza per poterli ripristinare successivamente.
- In genere ogni vettore di interrupt è condiviso da tutti i dispositivi dello stesso tipo, perciò non identifica univocamente il terminale che ha causato l'interrupt. Si può risalire al numero del terminale attraverso la lettura di alcuni registri di dispositivo.
- A questo punto è possibile leggere ogni altra informazione sull'interrupt, come il codice di stato.
- Nel caso si fosse verificato un errore di I/O, lo si può gestire da questo momento.
- Le variabili globali *ptr* e *count* vengono aggiornate; la prima è incrementata perché punti al byte successivo, la seconda è decrementata a indicare che manca un byte in meno da visualizzare. Se *count* è ancora maggiore di 0, ci sono altri caratteri da visualizzare, e quello ora puntato da *ptr* viene copiato nel registro buffer di output.
- Se richiesto, viene inviato un codice speciale per specificare al dispositivo o al controllore dell'interrupt che l'interrupt è stato trattato.
- Ripristino di tutti i registri salvati.
- Esecuzione dell'istruzione RITORNO DA INTERRUPT che ripristina la modalità e lo stato della CPU a prima del sollevamento dell'interrupt. Infine il computer riprende la sua attività dal punto in cui era stato interrotto.

Un concetto chiave legato agli interrupt è la **trasparenza**. Al verificarsi di un interrupt si intraprendono alcune azioni e si procede con l'esecuzione di un certo codice, ma quando tutto è finito il computer deve ripartire esattamente dallo stato che aveva prima dell'interrupt. Una routine di interrupt che manifesta questa proprietà si dice *trasparente*. Avere interrupt trasparenti ne semplifica la comprensione.

Se un computer ha un solo dispositivo di I/O, gli interrupt funzionano sempre nel modo appena descritto, e non resta da aggiungere nulla a quanto detto. D'altro canto un grosso calcolatore può disporre di numerosi dispositivi di I/O, molti dei quali attivi nello stesso momento e magari per conto di utenti diversi. Esiste una probabilità non nulla che un secondo dispositivo di I/O voglia generare il *proprio* interrupt mentre una routine di interrupt è già in esecuzione.

In questa evenienza si possono prevedere due strategie. La prima è far sì che ogni routine di interrupt per prima cosa disabiliti eventuali interrupt successivi, ancor prima di salvare i registri. Questo approccio mantiene lo schema semplice, visto che gli interrupt sono trattati in modo strettamente sequenziale, ma può generare problemi ai dispositivi che non tollerano attese oltre una certa durata. Per esempio, su una linea di comunicazione a 9600 bps arrivano caratteri ogni 1042 µs, pronti o meno a riceverli. Se il primo carattere non è stato ancora elaborato, quando arriva il secondo è possibile che alcuni dati vadano persi.

Se un calcolatore possiede dispositivi di I/O per i quali la gestione del tempo è critica, una migliore strategia consiste nell'assegnare una priorità a ogni dispositivo di I/O: alta per i dispositivi molto critici, bassa per quelli meno critici. Allo stesso modo anche la CPU dovrebbe definire alcune priorità, in genere determinate da un campo in PSW. Quando un dispositivo di priorità n causa un'interruzione, anche la routine di interrupt dovrebbe girare a livello di priorità n .

Durante l'esecuzione di una routine di interrupt di priorità n , ogni tentativo di interruzione di dispositivi di priorità inferiore viene ignorato fintanto che la routine non sia stata completata e che la CPU non sia tornata a eseguire codice di priorità più bassa. Viceversa gli interrupt provenienti da dispositivi a priorità maggiore dovrebbero essere trattati senza attesa.

Essendo le stesse routine di interrupt soggette a interruzioni, il modo migliore per mantenerne una gestione corretta è far sì che tutti gli interrupt siano trasparenti. Si consideri un semplice esempio di interrupt multipli. Un computer ha tre dispositivi di I/O, una stampante, un disco e una linea (seriale) RS232, rispettivamente con priorità 2, 4 e 5. Al tempo $t = 0$ è in esecuzione un programma dell'utente, ma all'improvviso, al tempo $t = 10$, si verifica un interrupt dalla stampante. Viene allora fatta partire la routine di servizio dell'interrupt (*Interrupt Service Routine*, ISR) della stampante, come mostrato nella Figura 5.43.

Al tempo $t = 15$, la linea RS232 invoca attenzione e genera un interrupt. Dal momento che la linea RS232 ha priorità maggiore (5) della stampante (2), il suo interrupt si verifica. Lo stato della macchina, impegnata al momento nell'esecuzione della routine di servizio dell'interrupt della stampante, viene impilato sullo stack e parte l'esecuzione della routine di servizio dell'interrupt da RS232.

Poco dopo, all'istante $t = 20$, il disco richiede a sua volta di essere servito. Tuttavia la sua priorità (4) è minore di quella della routine di interrupt correntemente in esecuzione (5), perciò l'hardware della CPU non conferma l'interrupt, e questo resta sospeso in attesa. All'istante $t = 25$ la routine della linea RS232 termina e lo stato viene ripristinato al momento precedente l'interruzione da parte di RS232, ossia ritorna all'esecuzione della routine di servizio della stampante di priorità 2. Non appena la CPU passa alla priorità 2 e prima che ne venga eseguita una sola istruzione, il controllo passa all'interrupt del disco di priorità 4 e viene eseguita la sua routine di servizio. Al termine di essa, la routine della stampante ottiene di poter proseguire. Infine, al tempo $t = 40$, tutte le routine di servizio degli interrupt sono state completate e il programma dell'utente riprende da dove era stato interrotto.

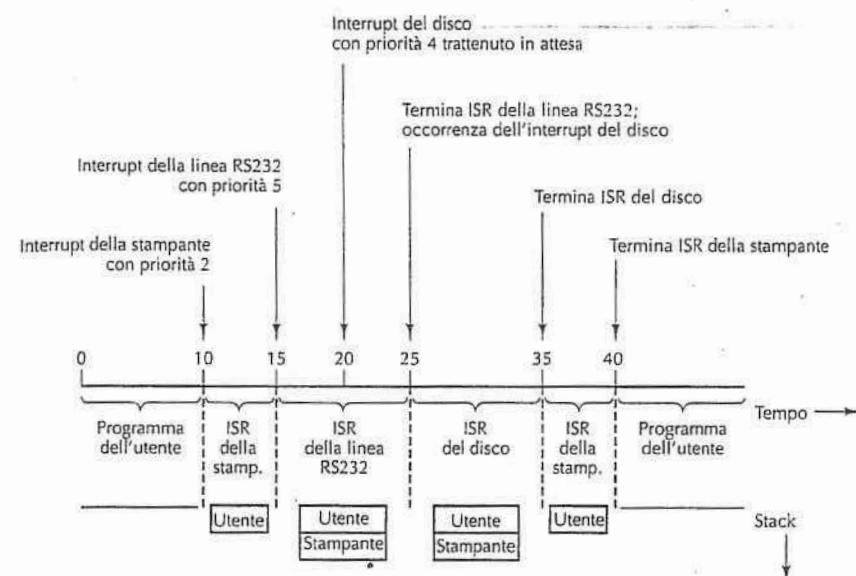


Figura 5.43 Sequenza di interrupt.

A partire dall'8088, tutte le CPU Intel sono state dotate di due livelli (priorità) di interrupt: mascherabile e non mascherabile. Gli interrupt non mascherabili sono usati in genere solo per segnalare eventi quasi catastrofici, come gli errori di parità della memoria. Tutti i dispositivi di I/O usano interrupt mascherabili.

Quando un dispositivo di I/O lancia un interrupt, la CPU usa il vettore di interrupt per indicizzare una tabella di 256 elementi al fine di trovare l'indirizzo della sua routine di servizio. Gli elementi della tabella sono descrittori di segmenti di 8 byte e la tabella può trovarsi ovunque in memoria. Un registro globale punta al suo inizio.

Essendoci un solo livello di interrupt utilizzabile, la CPU non ha modo di far sì che un dispositivo di priorità maggiore interrompa una routine di servizio di priorità media, e di garantire al contempo che ciò sia vietato a un dispositivo di priorità bassa. Per risolvere questo problema, le CPU Intel sono usate spesso in congiunzione con un controllore esterno di interrupt (per esempio un 8259A). Quando sopraggiunge il primo interrupt, diciamo di priorità n , la CPU viene interrotta. Se poi segue un interrupt di priorità maggiore, il controllore di interrupt causa una seconda interruzione. Se invece il secondo interrupt è di livello inferiore, viene trattenuto fino al completamento del primo. Affinché questo schema funzioni, il controllore di interrupt deve sapere quando termina la routine di servizio corrente, così la CPU deve inviare un comando quando il trattamento dell'interrupt corrente viene completato.

5.7 Un esempio: le torri di Hanoi

Dopo aver studiato l'ISA di tre macchine diverse possiamo ora trarre le somme presentando l'esempio di programma delle torri di Hanoi nel caso delle due macchine più complesse. Abbiamo già offerto una versione Java di questo esempio nella Figura 5.39. Nei paragrafi seguenti proponiamo due programmi in codice assemblativo per le torri di Hanoi.

Ricorreremo a un piccolo espediente: invece di fornire la traduzione della versione Java per il Core i7 e per l'OMAP4430, ne daremo una traduzione dal C per ovviare ad alcuni problemi dell'I/O di Java. La sola differenza è la sostituzione della chiamata Java a `printf` con l'istruzione C standard

```
printf("Move a disk from %d to %d\n", i, j)
```

A questo proposito non è rilevante la sintassi delle stringhe di formattazione della `printf` (fondamentalmente una stringa è interpretata letteralmente a eccezione della sequenza `%d`, che specifica il formato di visualizzazione decimale per l'intero successivo). Ciò che qui conta è che la procedura viene chiamata con tre parametri: una stringa di formattazione e due interi. La ragione dell'utilizzo della versione C per il Core i7 e per l'OMAP4430 è che la libreria Java di I/O non è disponibile in forma nativa per queste macchine, mentre la libreria C lo è. La differenza è minima e coinvolge la sola istruzione di stampa sullo schermo.

5.7.1 Le torri di Hanoi nel linguaggio assemblativo del Core i7

La Figura 5.44 fornisce una possibile traduzione della versione C delle torri di Hanoi per il Core i7; buona parte del codice è di comprensione immediata. Il registro EBP è usato come puntatore al record d'attivazione. Le prime due parole sono direttive per il linker, così il primo vero parametro, *n* (o *N* in questo caso, non essendo MASM case sensitive¹) si trova alla posizione EBP + 8, seguito da *i* e *j* alle posizioni EBP + 12 e EBP + 16, rispettivamente. La variabile locale *k* si trova in EBP + 20.

La procedura comincia con l'attivare un nuovo record alla fine del precedente. Per far ciò copia ESP nel puntatore al record d'attivazione, EBP. Quindi confronta *n* a 1 e salta alla clausola *else* se *n* > 1. Il ramo *then* impila sullo stack tre valori: l'indirizzo della stringa di formattazione, *i* e *j*, chiama `printf`. I parametri sono impilati in ordine inverso, come richiesto dai programmi C. Ciò si rende necessario perché il puntatore alla stringa di formattazione si trovi in cima allo stack. Poiché `printf` ha un numero variabile di parametri, se i parametri fossero stati impilati in ordine, `printf` non avrebbe avuto modo di stabilire quante locazioni scorrere lungo lo stack per trovare la stringa di formattazione. Dopo la chiamata si somma 12 a ESP per rimuovere i parametri dallo stack. Ovviamamente essi non vengono davvero cancellati dalla memoria, ma la modifica di ESP li rende inaccessibili alle normali operazioni sullo stack.

La clausola *else*, che comincia a L1, è di facile comprensione. Per prima cosa calcola $6 - i - j$ e memorizza il valore in *k*. Qualsiasi sia il valore di *i* e *j*, il terzo piolo è

sempre $6 - i - j$. Memorizzandone il valore in *k* ci si risparmia di doverlo ricalcolare una seconda volta. Successivamente la procedura richiama se stessa tre volte, con parametri ogni volta differenti. Dopo ogni chiamata lo stack viene ripulito.

Le procedure ricorsive spesso confondono chi vi si avvicina per la prima volta, ma risultano facili da capire quando osservate a questo livello. Tutto ciò che succede è l'aggiunta dei parametri in cima allo stack e la chiamata stessa di procedura.

```
; compila per il Core i7
; esporta 'towers'
; importa printf
; salva EBP (puntatore al record d'attivazione) e decrementa ESP
; imposta nuovo puntatore al record al di sopra di ESP
; if (n == 1)
; salta se n è diverso da 1
; printf "...", i, j;
; si noti che i parametri i, j e la stringa
; di formattazione sono impilati sullo stack
; in ordine inverso. È la convenzione per le chiamate C
; è l'indirizzo della stringa formattato
; chiama printf
; rimuove i parametri dallo stack
; abbiamo concluso
; comincia k = 6 - i - j
; EAX = 6 - i
; EAX = 6 - i - j
; k = EAX
; comincia towers(n - 1, i, k)
; EAX = i
; push di i
; EAX = n
; EAX = n - 1
; push di n - 1
; chiama towers(n - 1, i, 6 - i - j)
; rimuove i parametri dallo stack
; comincia towers(1, i, j)
; push di j
; EAX = i
; push di i
; push di 1
; chiama towers(1, i, j)
; rimuove i parametri dallo stack
; comincia towers(n - 1, 6 - i - j, j)
; push di j
; EAX = k
; push di k
; EAX = n
; EAX = n - 1
; push di n - 1
; chiama towers(n - 1, 6 - i - j, j)
; modifica il puntatore allo stack
; preparazione all'uscita
; ritorno al chiamante
DB "Move disk from %d to %d\n" ; stringa di formattazione
```

¹ Un linguaggio si definisce *case sensitive* se distingue tra caratteri minuscoli e maiuscoli, ovvero se considera *a* e *A* due identificativi diversi (N.d.T.).

Figura 5.44 Le torri di Hanoi nel linguaggio del Core i7.

5.7.2 Le torri di Hanoi nel linguaggio assemblativo dell'OMAP4430 ARM

Ora riproviamo la traduzione, questa volta per l'OMAP4430 ARM. Il codice è elencato nella Figura 5.45. Dal momento che il codice dell'OMAP4430 è particolarmente illeggibile, anche a livello del codice assemblativo e pur dopo molta pratica, ci siamo presi la libertà di cominciare con il definire alcuni simboli per renderlo più chiaro.

```
#define Param0          r0
#define Param1          r1
#define Param2          r2
#define FormatPtr       r0
#define k               r7
#define n_minus_1       r5

.text
towers: push {r3, r4, r5, r6, r7, lr}           @ Salva l'indirizzo di ritorno e i registri usati
        mov r4, Pa ram1
        mov r6, Pa ram2
        cmp Pa ram0, #1
        bne else
        movw FormatPtr, #:lower16:format
        movt FormatPtr, #:upper16:format
        bl printf
        pop {r3, r4, r5, r6, r7, pc}

else:   rsb k, r1, #6
        subs k, k, r2
        add n_minus_1, r0, #-1
        mov r0, n_minus_1
        mov r2, k
        bl towers
        mov r0, #1
        mov r1, r4
        mov r2, r6
        bl towers
        mov r0, n_minus_1
        mov r1, k
        mov r2, r6
        bl towers
        pop {r3, r4, r5, r6, r7, pc}
        @ Chiama towers(n-1, i, k)
        @ Calcola (n-1) per la chiamata ricorsiva
        @ Chiamata towers(n-1, i, j)
        @ Chiamata towers(1, k, j)
        @ Chiamata towers(n-1, k, j)
        @ Ripristina i registri usati e ritorna al chiamante
        @ Salva l'indirizzo di ritorno

main:  push {lr}
        mov Param0, #3
        mov Param1, #1
        mov Param2, Pa ram0
        bl towers
        pop {pc}
        @ Chiama towers(3, 1, 3)
        @ Preleva l'indirizzo di ritorno e ritorna al chiamante

format: .ascii "Move a disk from %d to %d\n\0"
```

Figura 5.45 Le torri di Hanoi nel linguaggio della CPU ARM OMAP4430.

Affinché possa funzionare, il programma deve essere fatto passare prima del suo assemblaggio attraverso un programma che si chiama *cpp*, un preprocessore C. Inoltre, in questo caso abbiamo usato lettere minuscole, perché l'assemblatore dell'OMAP4430 ARM ne richiede l'uso (casomai il lettore volesse provare a immettere il programma in una macchina OMAP4430).

Dal punto di vista algoritmico la versione per l'OMAP4430 è identica a quella per il Core i7. Entrambe cominciano con l'esaminare *n* e saltano alla clausola *else* se *n* > 1. La difficoltà principale nella versione per ARM è dovuta ad alcune proprietà dell'ISA.

Per cominciare, l'OMAP4430 deve passare l'indirizzo della stringa di formattazione alla *printf*, ma la macchina non può limitarsi a copiare l'indirizzo nel registro contenente il parametro in uscita, perché non c'è modo, con una sola istruzione, di mettere una costante di 32 bit in un registro, ma ce ne vogliono due: una *MOVW* e una *MOVT*.

Bisogna poi osservare che le operazioni sullo stack sono gestite automaticamente dalle istruzioni di *PUSH* e *POP* all'inizio e alla fine delle funzioni. Queste istruzioni gestiscono anche il salvataggio e il rispristino dell'indirizzo di ritorno, salvando LP quando si entra nella funzione e ripristinandolo prima dell'uscita.

5.8 Architettura IA-64 e Itanium 2

Intorno all'anno 2000 alcuni ingegneri di Intel iniziarono a pensare che la società stava arrivando al punto di aver spremuto del tutto la linea di processori IA-32. I nuovi modelli godevano ancora dei vantaggi portati dagli avanzamenti tecnologici, ovvero della più piccola dimensione dei transistor (che porta a una maggiore velocità di clock). Tuttavia diventava sempre più difficile escogitare nuovi stratagemmi per velocizzare ulteriormente le implementazioni, dato che i vincoli imposti dall'ISA IA-32 si rivelavano sempre più costrittivi.

Secondo alcuni ingegneri l'unica soluzione realistica era abbandonare IA-32 come linea principale di sviluppo e spostarsi verso un ISA completamente nuovo. È proprio ciò che Intel ha iniziato a fare: vi erano infatti progetti per due nuove linee. EMT-64 è un rifacimento ampliato del tradizionale ISA Pentium, con registri di 64 bit e spazio degli indirizzi a 64 bit. Questo nuovo ISA risolve il problema dello spazio degli indirizzi, ma presenta ancora le complicazioni implementative dei suoi predecessori. In pratica può essere visto come un Pentium esteso.

L'altra nuova architettura, sviluppata congiuntamente da Intel e Hewlett Packard, venne chiamata IA-64. Si tratta di una macchina completamente a 64 bit, non di un'estensione di una macchina a 32 bit. Inoltre si differenzia radicalmente dall'architettura IA-32 per molti aspetti. Inizialmente si rivolge al mercato dei server di fascia alta, ma Intel sperava di poter raggiungere anche il mercato dei desktop. Ciò non avvenne: nonostante i suoi difetti, gli acquirenti rifiutarono di abbandonare l'architettura IA-32. A ogni modo, la sua architettura è così diversa da tutto quanto fin qui studiato che merita una trattazione a parte. La prima implementazione dell'architettura IA-64 è la serie Itanium. Nel resto del paragrafo analizzeremo l'architettura IA-64 e la CPU Itanium 2 che l'implementa.

5.8.1 Il problema dell'ISA IA-32

Prima di addentrarci nei dettagli di IA-64 e dell'Itanium 2 è utile ricordare gli aspetti negativi dell'ISA IA-32 per capire quali sono i problemi che Intel ha inteso risolvere con la nuova architettura. La questione fondamentale è che IA-32 è un ISA datato, le cui proprietà sono inadatte alla tecnologia corrente. È un ISA CISC con istruzioni di lunghezza variabile e una miriade di formati differenti difficili da decodificare velocemente. La tecnologia attuale lavora meglio con ISA RISC che hanno una sola lunghezza per le istruzioni e opcode di lunghezza fissa facili da decodificare. Le istruzioni IA-32 possono essere suddivise in microistruzioni di tipo RISC in fase di esecuzione, ma ciò richiede hardware (cioè superficie del chip), porta via tempo e contribuisce alla complessità del progetto. Questi sono i primi punti a sfavore.

Inoltre IA-32 è un ISA orientato alla memoria ed è a due indirizzi. La maggior parte delle istruzioni riferenzia la memoria e la maggior parte dei programmati e dei compilatori non si cura di far riferimento continuo alla memoria. La tecnologia attuale favorisce gli ISA load/store che accedono alla memoria solo per recuperare gli operandi e trasferirli nei registri, altrimenti effettuano tutta la computazione usando istruzioni a tre indirizzi di registri. Per di più questa deficienza peggiorerà con il passare del tempo, visto che la frequenza di clock delle CPU cresce molto più velocemente rispetto alla velocità della memoria. Questo è il secondo punto a sfavore.

Inoltre IA-32 dispone di un insieme di registri piccolo e irregolare. Non solo questo imbriglia i compilatori, ma il numero esiguo di registri d'uso generale (quattro o sei, a seconda che si contino anche EDI ed ESI) esige che i risultati intermedi siano riversati continuamente in memoria, risultando in un sovrannumero di accessi anche laddove non sarebbero necessari. E con questo terzo punto a sfavore IA-32 perde il primo tempo della partita.

Cominciamo ora il secondo tempo. Il numero limitato di registri causa molte dipendenze, in special modo dipendenze WAR non necessarie, perché i risultati devono essere messi da qualche parte e non c'è più spazio nei registri. Per aggirare la mancanza di registri bisogna che l'implementazione gestisca la rinomina internamente, un artificio tra i peggiori, per nascondere i registri all'interno del buffer di riordinamento. Per evitare di bloccarsi frequentemente a seguito di miss della cache, le istruzioni devono essere eseguite fuori sequenza. Tuttavia la semantica di IA-32 specifica precise interruzioni di modo tale che le istruzioni fuori sequenza siano ritirate in ordine. Tutto ciò richiede però altro hardware molto complesso. Quarto punto a sfavore.

Per svolgere velocemente tutte queste operazioni occorre una pipeline con molti stadi. A sua volta, l'utilizzo di queste pipeline comporta che le istruzioni impieghino molti cicli per venir completate. Di conseguenza per garantire che le istruzioni introdotte nelle pipeline siano quelle giuste è essenziale una predizione dei salti molto precisa. Una predizione sbagliata richiede lo svuotamento della pipeline, un'operazione molto costosa, perciò anche un tasso di predizioni erronee abbastanza basso può causare un degrado sostanziale delle prestazioni. Quinto punto a sfavore.

Per alleviare il problema delle predizioni sbagliate il processore deve effettuare l'esecuzione speculativa, con tutti i problemi che comporta, specie quando i riferimenti alla memoria causano un'eccezione. Sesto punto a sfavore.

Non occorre continuare per capire che ci troviamo davanti a un vero problema. Non abbiamo neanche menzionato il fatto che gli indirizzi di 32 bit di IA-32 limitano i singoli programmi a 4 GB di memoria, un limite problematico su server di fascia alta. EMT-64 risolve questo problema, ma non tutti gli altri.

Dopo tutto la situazione di IA-32 può essere paragonata con ragione allo stato della meccanica celeste appena prima di Copernico. La teoria astronomica allora dominante stabiliva che la Terra fosse immobile e che i pianeti ruotassero con epicicli attorno a essa. D'altra parte, al migliorare delle osservazioni e al crescere delle discrepanze tra i dati osservati e il modello, furono aggiunti epicicli a epicicli finché l'intero modello collassò a causa della sua complessità interna.

Intel si trova ora nello stesso guaio. Una porzione enorme dei transistor del Pentium 4 è destinata a scomporre le istruzioni CISC, stabilire ciò che può essere svolto in parallelo, risolvere conflitti, fare predizioni, rimediare alle conseguenze di predizioni scorrette e ad altra ordinaria amministrazione, lasciando una frazione sorprendentemente piccola per svolgere i compiti che l'utente ha effettivamente richiesto. La conclusione cui è giunta Intel è l'unica sensata: buttare via tutto (IA-32) e ricominciare da capo dopo aver fatto tabula rasa (IA-64). EMT-64 consente un certo margine di respiro, ma si tratta davvero di nascondere il problema della complessità sotto un tappeto di novità.

5.8.2 Modello IA-64 e calcolo che utilizza il parallelismo esplicito

L'idea chiave alla base di IA-64 è di spostare il carico di lavoro dalla fase di esecuzione alla fase di compilazione. Nel Core i7, durante l'esecuzione la CPU riordina le istruzioni, rinomina i registri, fa lo scheduling delle unità funzionali e svolge molto altro lavoro per stabilire come occupare pienamente tutte le risorse hardware. Nel modello IA-64 il compilatore prevede tutto ciò e produce un programma che può essere eseguito così com'è, senza che l'hardware debba destreggiarsi tra tutti quei dettagli durante l'esecuzione. Per esempio, invece di dire al compilatore che la macchina ha otto registri quando in realtà ne ha 128, per poi cercare di evitare le dipendenze durante l'esecuzione, il modello IA-64 comunica al compilatore il numero effettivo di registri della macchina, così da poter produrre programmi senza conflitti tra registri. Analogamente, in questo modello il compilatore tiene traccia delle unità funzionali occupate e non emette istruzioni che usano unità funzionali non disponibili. Il modello che rende il parallelismo sottostante nell'hardware visibile al compilatore si chiama EPIC (*Explicitly Parallel Instruction Computing*, "calcolo che utilizza il parallelismo esplicito"). EPIC può essere considerato in un certo senso il successore di RISC.

Il modello IA-64 presenta numerose funzionalità per incrementare le prestazioni. Tra queste ce ne sono alcune per ridurre i riferimenti in memoria, per lo scheduling delle istruzioni, per ridurre i salti condizionati e altro. Vediamole ora una per una, riferendo come sono implementate nell'Itanium 2.

5.8.3 Riduzione degli accessi in memoria

L'Itanium 2 ha un modello di memoria semplice. La memoria consiste di 2^{64} byte di memoria lineare. Ci sono istruzioni per accedere a unità di memoria di 1, 2, 4, 16 e 10

byte, quest'ultima per i numeri in virgola mobile IEEE 745 di 80 bit. I riferimenti in memoria non devono essere allineati alle loro estremità naturali, ma se non lo sono ciò comporta un degrado delle prestazioni. L'ordinamento della memoria può essere sia little-endian sia big-endian, specificato tramite il valore di un bit appartenente a un registro a disposizione del sistema operativo.

Nei computer moderni l'accesso alla memoria costituisce un collo di bottiglia molto stretto, perché le CPU sono molto più veloci della memoria. Un modo per ridurre gli accessi alla memoria è disporre di una grande cache di primo livello sul chip e di una cache di secondo livello ancora più grande, vicina al chip. Oltre all'uso delle cache ci sono altri modi per ridurre gli accessi in memoria, alcuni dei quali sono sfruttati da IA-64.

Il modo migliore per velocizzare i riferimenti in memoria è... non usarli per niente. L'implementazione Itanium 2 del modello IA-64 ha 128 registri d'uso generale, di 64 bit ciascuno. I primi 32 registri sono statici, mentre i rimanenti 96 sono usati come registri di stack, in un modo molto simile allo schema a finestra di registri di altri processori RISC come l'UltraSPARC. A differenza dell'UltraSPARC il numero di registri visibili al programma è però variabile e può cambiare da procedura a procedura. Così ogni procedura ha accesso a 32 registri statici più un numero (variabile) di registri allocati dinamicamente.

All'atto della chiamata di una procedura il puntatore allo stack è incrementato, affinché i parametri d'ingresso siano visibili nei registri, ma non viene allocato alcun registro per le variabili locali. È la procedura stessa a decidere quanti registri necessita, e per allocarli incrementa lo stack pointer. Questi registri non devono essere salvati all'ingresso né ripristinati all'uscita, anche se la procedura deve aver cura di salvare e poi ripristinare i registri statici da modificare. Grazie al numero variabile di registri disponibili, legato alle reali esigenze di ogni procedura, si evita lo spreco dei già pochi registri; le chiamate di procedura possono così innestarsi a maggior profondità prima che si debbano riversare i registri in memoria.

L'Itanium 2 dispone anche di 128 registri in virgola mobile che rispettano il formato IEEE 745. Non vengono usati come registri di stack, ma il loro numero abbondante consente di memorizzare i risultati intermedi di molte operazioni in virgola mobile, senza bisogno di salvarli temporaneamente in memoria.

Ci sono anche 64 registri predicatori di 1 bit, otto registri di salto e 128 registri per applicazioni specifiche usati per vari propositi, come il passaggio di parametri tra programmi applicativi e il sistema operativo. La Figura 5.46 dà una panoramica dei registri dell'Itanium 2.

5.8.4 Scheduling delle istruzioni

Uno dei problemi principali del Core i7 è la difficoltà insita nel trovare uno scheduling delle varie istruzioni, a favore delle diverse unità funzionali, che eviti le dipendenze. Ci vogliono meccanismi esageratamente complicati per gestire tutti questi aspetti in fase d'esecuzione, e così un'ingente porzione dell'area del chip è dedicata al loro trattamento. IA-64 e Itanium 2 evitano questi problemi scaricando il lavoro sul compilatore. L'idea chiave è che un programma consista in una sequenza di **gruppi d'istruzioni**.

Entro un certo limite le istruzioni di un gruppo non sono mai in conflitto tra di loro, non usano più unità funzionali e risorse di quelle a disposizione della macchina, non contengono dipendenze RAW e WAW, ma solo dipendenze WAR limitate. I gruppi consecutivi d'istruzioni danno l'impressione di un'esecuzione strettamente sequenziale, laddove l'esecuzione del secondo gruppo non comincia finché non sia stato completato il primo. Tuttavia la CPU può iniziare a elaborare (in parte) il secondo gruppo non appena lo reputi sicuro.

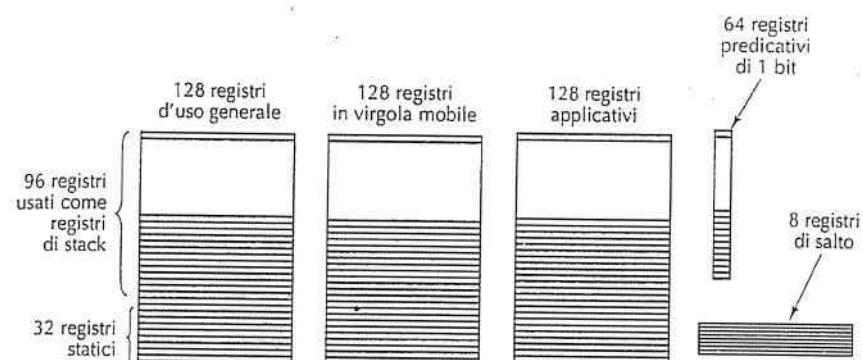


Figura 5.46 Registri di Itanium 2.

Una conseguenza di queste regole è che la CPU è libera di scegliere l'ordine di esecuzione delle istruzioni (inclusa l'esecuzione parallela), senza timore di conflitti. Il comportamento di un programma con un gruppo d'istruzioni che viola le regole non è definito. Spetta al compilatore riordinare il codice assemblativo generato dal programma sorgente al fine di soddisfare tutti i requisiti. In fase di sviluppo dell'applicazione il compilatore può mettere ciascuna istruzione in un gruppo per ottenere una compilazione rapida; è una soluzione facile, ma che porta a prestazioni molto scarse. Al momento di produrre il codice definitivo il compilatore può impiegare molto tempo per ottimizzarlo.

Le istruzioni sono organizzate in pacchetti d'istruzioni (*bundle*) di 128 bit, come mostrato nella parte alta della Figura 5.47. Ogni pacchetto contiene tre istruzioni di 41 bit e un campo template di 5 bit. Un gruppo d'istruzioni non deve prendere necessariamente un numero intero di pacchetti; può cominciare e terminare nel mezzo di un pacchetto.

Esistono più di cento formati d'istruzioni. La Figura 5.47 ne mostra uno tipico per le operazioni della ALU, come la ADD che somma due registri in un terzo. Il primo campo GRUPPO DI OPERAZIONI è il gruppo principale e specifica in sostanza la classe generale di appartenenza dell'istruzione, per esempio le operazioni ALU intere. Il campo successivo, TIPO DI OPERAZIONE, restituisce la particolare operazione richiesta, per esempio ADD o SUB. Quindi vengono i tre campi registro e infine c'è il REGISTRO PREDICATIVO, trattato a breve.



Figura 5.47 Pacchetto d'istruzioni IA-64 contenente tre istruzioni.

Il template del pacchetto specifica le unità funzionali necessarie al pacchetto e anche l'eventuale presenza e posizione di un'estremità del pacchetto d'istruzioni. Le unità funzionali principali sono le ALU (intera e non), le operazioni di memoria, le operazioni floating point e i salti. Ovviamente l'ortogonalità completa con sei unità e tre istruzioni richiederebbe 216 combinazioni, più altre 216 per indicare un marcatore di gruppo d'istruzioni dopo l'istruzione 0, altre 216 per indicare un marcatore di gruppo d'istruzioni dopo l'istruzione 1, e ancora 216 per indicare un marcatore di gruppo d'istruzioni dopo l'istruzione 2. Con soli 5 bit disponibili è consentito solo un numero molto limitato di queste combinazioni. C'è da dire che, se anche ci fosse un modo di specificare tre istruzioni in virgola mobile in un pacchetto d'istruzioni, non servirebbe a nulla, perché la CPU non può avviare simultaneamente tre istruzioni in virgola mobile. Le combinazioni previste sono quelle effettivamente attuabili.

5.8.5 Riduzione dei salti condizionati: attribuzione di predici

Un'altra caratteristica importante di IA-64 è il modo nuovo di trattare i salti condizionati. Se ci fosse un modo di liberarsene quasi del tutto, le CPU ne beneficierebbero grandemente in semplicità e velocità. Di primo acchito si direbbe impossibile sbarazzarsi dei salti condizionati, perché i programmi sono pieni d'istruzioni if, e invece IA-64 usa una tecnica, detta **attribuzione di predici** (*predication*), che può ridurre di molto il loro numero (August et al., 1998; Hwu, 1998). Descriviamola brevemente.

Nelle architetture tradizionali tutte le istruzioni sono incondizionate, nel senso che quando la CPU raggiunge un'istruzione non fa altro che svolgerla. Non c'è alcun dubbio del tipo "fare o non fare, questo è il dilemma". Al contrario, in un'architettura predicativa le istruzioni contengono condizioni (predicati) che stabiliscono se devono essere eseguite o meno. Questo cambio di paradigma da istruzioni incondizionate a istruzioni predicative rende possibile la riduzione di (molti) salti condizionati. Invece di scegliere tra due sequenze d'istruzioni incondizionate, si fondono tutte le istruzioni in una sola sequenza d'istruzioni predicative, usando predici diversi per istruzioni diverse.

Per vedere come funziona l'attribuzione di predici consideriamo la Figura 5.48, che mostra un semplice esempio di esecuzione condizionata, concetto precursore dell'attribuzione di predici. La Figura 5.48(a) contiene un'istruzione if, mentre la Figura 5.48(b) ne mostra la traduzione in tre istruzioni: un confronto, un salto condizionato e un trasferimento.

Nella Figura 5.48(c) ci liberiamo del salto condizionato grazie a una nuova istruzione, CMOVZ, che è un trasferimento condizionato. Il suo funzionamento è controllare se il terzo registro, R1, vale 0; se è così copia R3 in R2, altrimenti non esegue nulla.

if (R1 == 0) R2 = R3;	CMP R1,0 BNE L1 MOV R2,R3	CMOVZ R2,R3,R1
L1:	(a)	(b)

Figura 5.48 (a) Un'istruzione if. (b) Codice assemblativo per (a). (c) Un'istruzione condizionata.

Una volta resa disponibile un'istruzione che può copiare dati quando un certo registro vale 0, ci vuole poco per avere un'istruzione che trasferisce dati quando un certo registro non è 0: chiamiamola CMOVN. Con queste due istruzioni siamo sulla buona strada per l'esecuzione pienamente condizionata. Si immagini un'istruzione if con molti assegnamenti nel ramo then e molti altri nel ramo else. L'intera istruzione può essere tradotta in codice che imposta un certo registro a 0 se la condizione è falsa e a un altro valore se è vera. A seguire, il ramo then può essere compilato in una sequenza d'istruzioni CMOVN e il ramo else in una sequenza d'istruzioni CMOVZ.

Tutte queste istruzioni, l'impostazione del registro, le CMOVN e le CMOVZ, formano un singolo blocco elementare senza salti condizionati. L'ordine delle istruzioni può addirittura essere modificato, sia dal compilatore (che può anticipare gli assegnamenti prima del test) sia durante l'esecuzione. L'unico limite è che il valore della condizione deve essere noto prima che le istruzioni condizionate siano ritirate (verso la fine della pipeline). La Figura 5.49 illustra un semplice esempio con un ramo then e un ramo else.

if (R1 == 0) { R2 = R3; R4 = R5; } else { R6 = R7; R8 = R9; }	CMP R1,0 BNE L1 MOV R2,R3 MOV R4,R5 BR L2 L1: MOV R6,R7 MOV R8,R9 L2:	CMOVZ R2,R3,R1 CMOVZ R4,R5,R1 CMOVN R6,R7,R1 CMOVN R8,R9,R1
	(a)	(b)

Figura 5.49 (a) Un'istruzione if. (b) Codice assemblativo per (a). (c) Esecuzione condizionata.

Fin qui abbiamo presentato istruzioni condizionate molto semplici (appartenenti in realtà all'ISA IA-32), ma si tenga presente che tutte le istruzioni di IA-64 sono predicative. Ciò significa che l'esecuzione di ogni istruzione può essere resa condizionata. I 6 bit addizionali cui si è fatto riferimento prima servono a selezionare uno dei 64 registri predicatori di 1 bit. Perciò un'istruzione `if` sarà compilata in codice che imposta a 1 uno dei registri predicatori se la condizione è vera, e a 0 viceversa. Allo stesso tempo imposta automaticamente un altro registro predicativo al valore opposto. Grazie all'attribuzione di prediciati le istruzioni macchina che costituiscono le clausole `then` ed `else` possono essere fuse in un solo flusso d'istruzioni, dove le prime si avvalgono del predicato originale, e le seconde del suo opposto. Nel momento del controllo verrà eseguito un solo un insieme di istruzioni.

Per quanto semplice, l'esempio della Figura 5.50 mostra l'idea fondamentale di come l'attribuzione di prediciati sia utilizzabile per l'eliminazione dei salti. L'istruzione `CMPEQ` confronta due registri e asserisce il registro predicativo `P4` se sono uguali, lo azzerà se sono diversi. Inoltre imposta un registro accoppiato, diciamo `P5`, secondo la condizione complementare, dopo di che è possibile elencare di seguito le istruzioni delle parti `then` ed `else`, ciascuna condizionata da un certo registro predicativo (indicato tra parentesi angolari). Lo schema si può applicare a qualsiasi tipo d'istruzione, purché opportunamente predicata.

IA-64 conduce questa idea alle sue estreme conseguenze, disponendo d'istruzioni di confronto che impostano registri predicatori e d'istruzioni aritmetiche e non, la cui esecuzione dipende da un certo registro predicativo. Le istruzioni predicatorie possono essere inserite nella pipeline in sequenza, senza stalli né altri problemi, ragion per cui sono così utili.

L'attribuzione di prediciati su IA-64 avviene facendo sì che tutte le istruzioni siano eseguite. In fondo alla pipeline, al momento di ritirare un'istruzione si verifica se il suo predicato è vero. In caso affermativo l'istruzione è ritirata normalmente e il suo risultato è salvato nel registro destinazione. Se invece il predicato è falso, non si effettua alcun write-back e l'istruzione non ha effetto. L'attribuzione di prediciati è esaminata in maggior dettaglio in Dulong (1998).

if ($R1 == R2$) $R3 = R4 + R5$; else $R6 = R4 - R5$	CMP R1,R2 BNE L1 MOV R3,R4 ADD R3,R5 BR L2	CMPEQ R1,R2,P4 <P4> ADD R3,R4,R5 <P5> SUB R6,R4,R5
(a)	(b)	(c)

Figura 5.50 (a) Un'istruzione `if`. (b) Codice assemblativo per (a). (c) Esecuzione predicativa.

5.8.6 Caricamenti speculativi

Un'altra caratteristica di IA-64 che ne velocizza l'esecuzione è la presenza di LOAD speculativi. Se un'istruzione `LOAD` è speculativa e non va a buon fine, invece di causare un'eccezione, semplicemente si ferma e viene asserito un bit associato al registro da caricare che lo segnala come invalido. Si tratta proprio del poison bit introdotto nel Capitolo 4. Se il registro associato al poison bit viene usato successivamente, allora viene sollevata un'eccezione, che altrimenti non si verifica.

Il modo usuale di impiegare la speculazione è di permettere al compilatore di anticipare istruzioni `LOAD` in posizioni antecedenti al loro effettivo bisogno. Se cominciano prima è probabile che il loro risultato sia disponibile al momento opportuno. Il compilatore inserisce un'istruzione `CHECK` laddove necessita di un registro appena caricato. Se il valore è disponibile la `CHECK` si comporta come una `NOP` e l'esecuzione prosegue immediatamente. Se invece il valore non è ancora disponibile, l'istruzione successiva deve andare in stallo. Se si è verificata un'eccezione e il poison bit è asserito, anche l'eccezione sospesa viene sollevata.

In conclusione, le macchine che implementano l'architettura IA-64 devono la loro velocità a diversi fattori. Al centro c'è il motore RISC a tre indirizzi, con pipeline, di tipo load/store e che è allo stato dell'arte della materia. È già un miglioramento notevole rispetto alla complessa architettura IA-32.

In più IA-64 è provvista di un modello di parallelismo esplicito che richiede un compilatore in grado di stabilire quali istruzioni eseguire contemporaneamente senza conflitti e quindi capace di raggrupparle in pacchetti d'istruzioni. Così facendo la CPU può limitarsi alla pura gestione dello scheduling di un pacchetto senza inutili ripensamenti. Spostare il lavoro dall'esecuzione al momento della compilazione porta sempre a un successo.

Inoltre, l'attribuzione di prediciati consente di fondere le istruzioni di entrambi i rami d'un'istruzione `if` in un solo flusso, eliminando i salti condizionati, e quindi la necessità di predire quale ramo verrà percorso. Infine le LOAD speculative rendono possibile il fetch anticipato degli operandi, senza penalità in caso risultassero inutili.

Nel complesso l'architettura di Itanium è un progetto impressionante che sembra servire al meglio gli architetti e gli utenti, ma... avete un Itanium sul vostro computer? Ne abbiamo uno noi sul nostro? Conoscete qualcuno che ne possiede uno? Le risposte sono: no, no, no e (probabilmente) no. Più di due lustri dopo la sua introduzione, la sua adozione può essere descritta, in maniera politicamente corretta, come "mancata". Ciononostante, Intel continua a commissionare la produzione di sistemi basati su Itanium, anche se il loro utilizzo è limitato a server di alta fascia.

Torniamo alle motivazioni che in origine hanno portato alla creazione dell'architettura IA-64. Itanium fu progettato per sopportare alle numerose mancanze dell'architettura IA-32. Dato che la nuova architettura non è stata adottata, come ha potuto Intel affrontare queste mancanze? Come vedremo nel capitolo 8, la chiave di volta che ha permesso di portare avanti l'architettura IA-32 non è stata la riprogettazione dell'ISA, ma piuttosto il passaggio al calcolo parallelo, attraverso il progetto di chip multiprocessore. Per maggiori informazioni sull'Itanium 2 e sulla sua micro-architettura si veda McNairy e Soltis, 2003; Rusu *et al.*, 2000.

5.9 Riepilogo

L'architettura dell'insieme d'istruzioni è ciò a cui molte persone pensano in temini di "linguaggio macchina", anche se nelle macchine CISC è generalmente costruita su un livello più basso di microcodice. A questo livello la macchina ha una memoria orientata al byte o alla parola, costituita da decine di megabyte, e dispone d'istruzioni quali MOVE, ADD e BEQ.

Gran parte dei calcolatori moderni dispone di una memoria organizzata come una sequenza di byte, con gruppi di 4 o 8 byte a formare le parole. In genere ci sono tra gli 8 e i 32 registri, ciascuno lungo una parola. Su alcune macchine (come il Core i7) i riferimenti in memoria non devono essere allineati alle estremità naturali delle parole di memoria, mentre su altri (come l'OMAP4430 ARM) ciò è richiesto. Anche se non è richiesto l'allineamento delle parole le prestazioni sono migliori se le parole sono allineate.

Le istruzioni hanno in genere uno, due o tre operandi, indirizzati in modo immediato, diretto, a registro, indicizzato o in altri modi. Alcune macchine dispongono di un gran numero di complesse modalità d'indirizzamento. In molti casi, i compilatori non sono in grado di usarle in maniera efficace, e alcune modalità restano inutilizzate. Ci sono in genere istruzioni per trasferire dati, operazioni unarie e binarie, comprese le operazioni aritmetiche e quelle booleane, salti, chiamate di procedura, cicli e alle volte istruzioni per l'I/O. Le istruzioni in genere trasferiscono una parola dalla memoria in un registro (o viceversa), sommano, sottraggono, moltiplicano o dividono due registri o un registro e una parola di memoria, oppure confrontano due elementi nei registri o in memoria. Non è insolito trovare computer con più di 200 istruzioni nel loro repertorio. Le macchine CISC ne hanno spesso molte di più.

Il controllo del flusso a livello 2 si realizza tramite una varietà di primitive, inclusi i salti, le chiamate di procedura, le chiamate di routine, le trap e gli interrupt. I salti sono usati per concludere una sequenza d'istruzioni e cominciare una nuova a una diversa locazione di memoria (potenzialmente distante). Le procedure sono un'astrazione di questo meccanismo e consentono di isolare una parte di un programma come un'unità che può essere richiamata da molti punti diversi. L'astrazione ottenuta con l'uso delle procedure, in una qualsiasi forma, è la base della programmazione moderna. Senza le procedure (o le loro equivalenti) sarebbe impossibile scrivere qualsiasi software moderno. Le routine permettono l'esecuzione simultanea di due thread di controllo. Le trap sono utilizzate per segnalare situazioni eccezionali, come l'overflow aritmetico. Gli interrupt permettono all'I/O di interagire concorrentemente con la computazione principale: la CPU riceve un segnale non appena l'I/O ha completato la sua attività.

Le torri di Hanoi costituiscono un problema semplice e divertente che ammette l'elegante soluzione ricorsiva che abbiamo esaminato. Sono anche state trovate soluzioni iterative al problema, ma queste sono molto più complicate e meno eleganti di quella ricorsiva che abbiamo visto.

Infine, l'architettura IA-64 usa il modello di calcolo EPIC per facilitare l'uso del parallelismo da parte dei programmi. Per migliorare le prestazioni si avvale di gruppi d'istruzioni, dell'attribuzione di predici e di LOAD speculative. In definitiva potrebbe rappresentare un significativo miglioramento del Core i7, anche se scarica gran parte del

peso della parallelizzazione sul compilatore. A ogni modo, è sempre meglio che il lavoro sia eseguito in fase di compilazione piuttosto che in fase di esecuzione.

PROBLEMI

- Una parola di un computer little-endian con parole di 32 bit contiene il valore numerico 3. Se viene trasmessa a un computer big-endian byte per byte e ivi memorizzata, con il byte 0 al posto del byte 0, il byte 1 al posto del byte 1, e così via, quale sarà il suo valore numerico nella macchina big-endian quando viene letta come un intero di 32 bit?
- In passato diversi computer e sistemi operativi hanno utilizzato spazi separati per dati e istruzioni, permettendo fino a 2^k indirizzi per il programma e lo stesso per i dati, con l'utilizzo di indirizzi di k bit. Per esempio, per $k=32$ un programma poteva accedere a 4GB di istruzioni e 4GB di dati, per un totale di 8 GB di spazio di indirizzamento. Visto che quando si utilizza questo schema un programma non può sovrascrivere se stesso, come può il sistema operativo caricare i programmi in memoria?
- Si progettati un opcode espandibile, in cui le seguenti istruzioni possano essere tutte codificate in istruzioni di 36 bit:
 - 15 istruzioni con due indirizzi di 12 bit e un identificativo di registro di 4 bit
 - 650 istruzioni con un indirizzo di 12 bit e un identificativo di registro di 4 bit
 - 80 istruzioni senza indirizzi né registri.
- Una certa macchina dispone d'istruzioni di 16 bit e d'indirizzi di 6 bit. Alcune istruzioni contengono un indirizzo, altre due. Se ci sono n istruzioni a due indirizzi, qual è il numero massimo d'istruzioni a un indirizzo?
- È possibile progettare un opcode espandibile che consenta di codificare le seguenti istruzioni in istruzioni di 12 bit? Un registro è identificato da 3 bit.
 - 4 istruzioni con tre registri
 - 255 istruzioni con un registro
 - 16 istruzioni senza registri
- Dati i seguenti valori di memoria e una macchina a un indirizzo dotata di accumulatore, quali valori sono caricati all'interno dell'accumulatore dalle istruzioni specificate?
 - la parola 20 contiene 40
 - la parola 30 contiene 50
 - la parola 40 contiene 60
 - la parola 50 contiene 70
 - LOAD IMMEDIATE 20
 - LOAD DIRECT 20
 - LOAD INDIRECT 20
 - LOAD IMMEDIATE 30
 - LOAD DIRECT 30
 - LOAD INDIRECT 30
- Si confrontino le macchine a 0, 1, 2 e 3 indirizzi scrivendo un programma per il calcolo di $X = (A + B \times C) / (D - E \times F)$ per ciascuna delle quattro macchine. Le istruzioni disponibili sono:

0 Indirizzi	1 Indirizzo	2 Indirizzi	3 Indirizzi
PUSH M	LOAD M	MOV (X = Y)	MOV (X = Y)
POP M	STORE M	ADD (X = X + Y)	ADD (X = X + Y)
		SUB (X = X - Y)	SUB (X = X - Y)

SUB	SUB M	MUL (X = X * Y)	MUL (X = X * Y)
MUL	MUL M	DIV (X = X / Y)	DIV (X = X / Y)
DIV	DIV M		

M è un indirizzo di memoria di 16 bit, *X*, *Y* e *Z* sono o indirizzi di 16 bit o registri di 4 bit. La macchina a 0 indirizzi usa uno stack, la macchina a 1 indirizzo usa un accumulatore, mentre le altre due hanno 16 registri e istruzioni per operare su tutte le possibili combinazioni di locazioni di memoria e registri. SUB *X*, *Y* sottrae *Y* da *X* e SUB *X*, *Y*, *Z* sottrae *Z* da *Y* e pone il risultato in *X*. Con opcode di 8 bit e lunghezze d'istruzioni che sono multiple di 4 bit, di quanti bit ha bisogno ciascuna macchina per calcolare *X*?

8. Si escogiti un meccanismo d'indirizzamento che consenta di specificare, in uno spazio d'indirizzi molto grande, un insieme arbitrario di 64 indirizzi non necessariamente contigui, mediante un campo di 6 bit.
9. Si indichi uno svantaggio del codice auto-modificante non menzionato nel testo.
10. Si convertano le formule seguenti da notazione infissa a polacca inversa.
 - a. A + B + C + D - E
 - b. (A - B) × (C + D) + E
 - c. (A × B) + (C × D) + E
 - d. (A - B) × (((C - D × E) / F) / G) × H.
11. Quali delle seguenti coppie di formule in notazione polacca inversa sono matematicamente equivalenti?
 - a. A B + C + e A B C ++
 - b. A B - C - e A B C --
 - c. A B × C + e A B C + ×
12. Si convertano le seguenti formule da notazione polacca inversa a notazione infissa.
 - a. A B - C + D ×
 - b. A B / C D / +
 - c. A B C D E + × × /
 - d. A B C D E × F / + G - H / × +
13. Si scrivano tre formule in notazione polacca inversa che non possono essere convertite in notazione infissa.
14. Si convertano le seguenti formule booleane infisse in notazione polacca inversa.
 - a. (A AND B) OR C
 - b. (A OR B) AND (A OR C)
 - c. (A AND B) OR (C AND D)
15. Si converta la seguente formula infissa in notazione polacca inversa e si scriva del codice IJVM per valutarla.

$$(5 \times 2 + 7) - (4 / 2 + 1)$$
16. Quanti registri ci sono nella macchina che ha i formati d'istruzione della Figura 5.24?
17. Nella Figura 5.24 il bit 23 è usato per distinguere il formato 1 dal formato 2. Non c'è alcun bit per distinguere il formato 3. Come fa l'hardware a sapere quando usarlo?
18. Capita spesso nella programmazione di dover determinare dove si posiziona una variabile *X* rispetto a un intervallo [A, B]. Se si dispone di un'istruzione a tre indirizzi con operandi A, B e X, quanti bit codici di condizione si dovranno impostare per rappresentare il risultato del confronto?
19. Si descrivano un vantaggio e uno svantaggio dell'indirizzamento relativo al program counter.
20. Il Core i7 ha un bit codice di condizione che tiene traccia del riporto oltre il terzo bit a seguito di un'operazione aritmetica. A che cosa serve?
21. Un tuo amico ti ha appena svegliato telefonandoti nel cuore della notte e ti ha messo a parte delle sua nuova idea brillante: un'istruzione con due opcode. Gli consigli di correre all'ufficio brevetti o di tornare sui banchi di scuola?

22. Confronti del tipo
`if (k == 0) ...
if (a > b) ...
if (k < 5) ...`
 sono comuni in programmazione. Si progettati un'istruzione che effettui questi test in modo efficiente: quali sono i campi dell'istruzione?
23. Dato il numero binario di 16 bit 1001 0101 1100 0011, si mostrino gli effetti delle seguenti operazioni:
 - a. uno scorrimento verso destra di 4 bit con riempimento di zeri
 - b. uno scorrimento verso destra di 4 bit con estensione del segno
 - c. uno scorrimento verso sinistra di 4 bit
 - d. una rotazione verso sinistra di 4 bit
 - e. una rotazione verso destra di 4 bit.
24. Come si può azzerare una parola di memoria su una macchina priva d'istruzione CLR?
25. Si calcoli il valore dell'espressione booleana (A AND B) OR C per

$$\begin{aligned} A &= 1101\ 0000\ 1010\ 0011 \\ B &= 1111\ 1111\ 0000\ 1111 \\ C &= 0000\ 0000\ 0010\ 0000 \end{aligned}$$
26. Si escogiti un modo per scambiare il valore di due variabili *A* e *B* senza usare una terza variabile o registro.
Suggerimento: si pensi all'operazione di OR ESCLUSIVO.
27. Su un certo computer è possibile svolgere le seguenti operazioni in meno tempo di quanto impiegato da una moltiplicazione: trasferire un numero da un registro all'altro, far scorrere entrambi di un certo quantitativo di bit e sommare i risultati. Sotto quali condizioni questa sequenza d'istruzioni è utile per calcolare "costante × variabile"?
28. Macchine diverse hanno diverse densità d'istruzioni (numero di byte richiesti per svolgere un certo calcolo). Tradurre i seguenti frammenti di codice Java nel linguaggio assemblativo del Core i7 e in linguaggio IJVM. Calcolare quindi il numero di byte richiesti da ciascuna macchina per calcolare le espressioni. Si ipotizzi che *i* e *j* sono variabili locali in memoria, per il resto si facciano sempre le assunzioni più ottimistiche
 - a. *i* = 3;
 - b. *i* = *j*;
 - c. *i* = *j* - 1;
29. Le istruzioni di ciclo illustrate nel testo servono per la gestione di cicli for. Progettare invece un'istruzione che potrebbe essere utile per gestire comuni cicli while.
30. Se i monaci di Hanoi possono spostare un disco al minuto (non hanno fretta di portare a termine il compito perché le loro professionalità sono poco apprezzate sul mercato del lavoro di Hanoi), quanto impiegheranno a risolvere il problema con 64 dischi? Estrarre il risultato in anni.
31. Per quale motivo i dispositivi di I/O inviano il vettore di interrupt sul bus? Non potrebbero invece memorizzare questa informazione in una tabella di memoria?
32. Un certo computer usa il meccanismo DMA per leggere dal disco. Il disco ha 64 settori di 512 byte per ogni traccia. Il tempo di rotazione del disco è 16 ms. Il bus è largo 16 bit e ogni trasferimento sul bus impiega 500 ns. Un'istruzione di CPU richiede in media due cicli di bus. In che misura la CPU viene rallentata dal DMA?
33. Il trasferimento con DMA descritto nella figura 5.32 richiede 2 trasferimenti su bus per spostare i dati tra un dispositivo di I/O e la memoria. Si descriva come si possono migliorare le prestazioni del DMA utilizzando l'architettura del bus mostrata nella Figura 3.35.