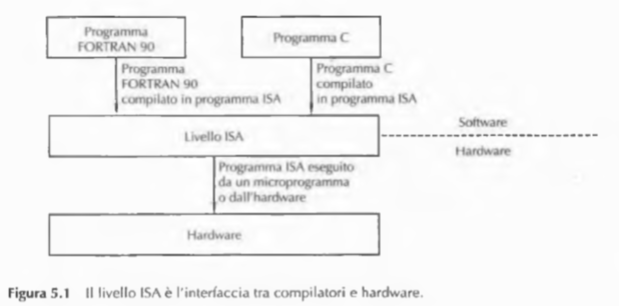
**Introduzione**

Questo capitolo tratta in dettaglio il livello di architettura dell’insieme di istruzioni (ISA, Instruction Set Architecture). Tale livello si trova tra quello della microarchitettura e il sistema operativo.

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 di 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 di 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, 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 accedere 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.

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?". 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 almeno renderlo 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 garantiscano 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.

Che cosa rende un ISA un “buon ISA”? Ci sono due fattori principali.

Innanzitutto un buon ISA dovrebbe definire un insieme di 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 sola generazione di 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 che 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 (se facile da implementare efficientemente), sia i progettisti software (se favorevole alla produzione di codice di qualità).

**Overview del livello ISA**

**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 programma più in linguaggio macchina, ridefiniamo il concetto dicendo che il codice di livello ISA è l’output di un compilatore. 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 di 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 super-scalare 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 otterrebbe prestazioni visibilmente migliori che non in caso contrario. Così i dettagli dell’operazione superscalare 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. 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.

Tali documenti di definizione contengono sezioni normative, che impongono alcuni requisiti, e sezioni informative, concepite per aiutare il lettore nella comprensione.

Le sezioni normative usano frequentemente locuzioni quali deve, non deve e dovrebbe rispettivamente per richiedere, proibire e suggerire aspetti dell’architettura.

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 architetturali è corredata da risultati di test sperimentali che verificano la reale conformità di un’implementazione alla specifica corrispondente.

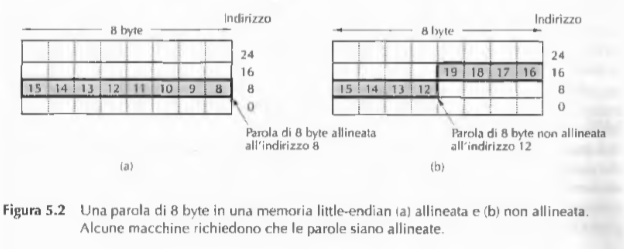
Un’altra proprietà importante del livello ISA è che la maggior parte dei processori è dotata di almeno due modalità d’esecuzione. La modalità kernel serve a eseguire il sistema operativo e permette l’esecuzione di tutte le istruzioni. La modalità utente ha lo scopo di eseguire i programmi applicativi e non consente l’esecuzione di certe istruzioni “delicate” (come quelle che manipolano direttamente la cache).

**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. Una cella di 8 bit si chiama byte. 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 riempiono esattamente un byte. Se in futuro lo standard UNICODE prenderà il sopravvento sul mercato, i computer potranno forse basarsi su unità consecutive di 16 bit.

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.

L’allineamento è richiesto perché in tal modo la memoria riesce a funzionare in modo più efficiente, d’altra parte il requisito dell’allineamento può anche causare difficoltà.



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 imporrebbero 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: “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 232 o 264 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 232 byte di programma e 232 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.

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 di 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.

I progettisti di sistema possono scegliere tra molti approcci risolutivi del problema. A un estremo c’è la possibilità di serializzare tutte le richieste di 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, ma non di altri. Nonostante l’esposizione della microarchitettura a livello ISA generi tutte queste anomalie abbastanza fastidiose (quanto meno per i programmatori che scrivono il compilatore e il linguaggio assemblativo), l’abitudine è molto diffusa. Questa tendenza è dovuta alle implementazioni sottostanti, quali il riordinamento delle microistruzioni, le pipeline profonde, i livelli multipli di cache, e così via.

**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 grossomodo 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).

I registri d’uso generale di alcune macchine sono del tutto simmetrici e intercambiabili. 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 Pentium 4 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.

II 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);

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).

**Istruzioni**

La caratteristica principale del livello ISA è l’insieme di 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.

**Tipi di dati**

Tutti i computer hanno bisogno di dati che, all’interno del computer, devono essere rappresentati in una forma specifica.

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. L’hardware si aspetta un certo formato di dati e non funziona correttamente se gli si passa altro.

Un altro problema è relativo alla necessità di un’aritmetica a più di 32 bit, perché i numeri coinvolti sono molto più grandi di 232. 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.

I tipi di dati possono essere divisi in due categorie: numerici e non.

**Tipi di dati numerici**

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. 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.

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 232-1, estremi inclusi. D’altra parte un intero con segno di 32 bit rappresentato in complemento a due può gestire numeri minori o uguali a 231-1 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. Questi sono lunghi 32, 64 o a volte 128 bit. Quasi tutti i computer dispongono di 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 mettono a disposizione un tipo di dati per i numeri decimali. Alcune macchine 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.

**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 e-mail, 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 importante 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.

Non è raro che il livello ISA comprenda istruzioni speciali per la gestione di stringhe 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.

L’ultimo tipo di dati che consideriamo è il tipo puntatore, nient’altro che un indirizzo di memoria. 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.

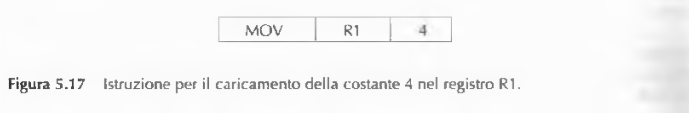
**Indirizzamento**

Molte istruzioni contengono operandi e si pone il problema di come specificarne la posizione.

Esistono diverse modalità di indirizzamento.

**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 5.17 mostra una possibile istruzione immediata 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.



**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.

**Indirizzamento a registro**

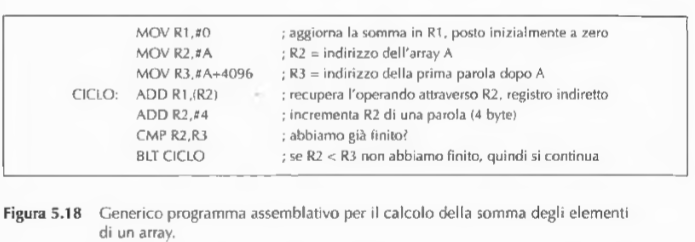
L’indirizzamento a registro è concettualmente analogo all’indirizzamento diretto, ma specifica un registro invece di una locazione di memoria. Si tratta della modalità di indirizzamento di gran lunga più utilizzata nella quasi totalità dei computer, dato che i registri sono veloci in accesso e hanno 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.

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.

**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. Quando un indirizzo è 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 di 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 5.18.



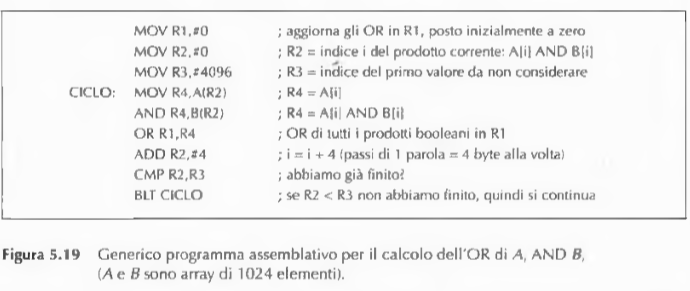
In questo semplice programma ci avvaliamo di parecchie modalità di indirizzamento. Le prime tre istruzioni usano la modalità a registro per il primo operando (la destinazione) e la modalità immediata per il secondo operando (una costante denotata dal simbolo #). La seconda istruzione copia l’indirizzo 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.

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 RI, A+4*

e così via fino alla fine del ciclo. Un programma che modifica se stesso si dice un programma auto-modificante. L’idea poteva essere sensata sui primi computer, che non disponevano di indirizzamento a registro indiretto. Oggigiorno i programmi auto-modificanti sono considerati di pessimo stile e difficili da capire. Inoltre non sono condivisibili da processi diversi.

**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 AND B, 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 5.18. Benchè sia una soluzione corretta, possiamo fare di meglio e usare uno schema più generale, come illustrato nella Figura 5.19.



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.

Dopo l’inizializzazione dei registri comincia il ciclo di sei istruzioni. La prima, etichettata come CICLO, effettua il fetch di A, 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 5.20.



Alla prima iterazione del ciclo, R2 vale 0 (è stato così inizializzato), perciò la parola di memoria indicizzata è A0, all’indirizzo 124300, e questa viene salvata in R4. All’iterazione successiva R2 vale 4, perciò la parola di memoria indicizzata è A, 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.

**Indirizzamento indicizzato esteso**

Alcune macchine dispongono della cosiddetta modalità di 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.

Poter disporre di una modalità di 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.

**Indirizzamento a stack**

Abbiamo già sottolineato che è consigliabile rendere le istruzioni macchina quanto più corte possibile. Il limite alla riduzione della lunghezza degli indirizzi equivale a non averne per nulla. Le istruzioni senza indirizzi, come l’istruzione IADD, sono possibili in associazione con uno stack.

**Notazione polacca inversa**

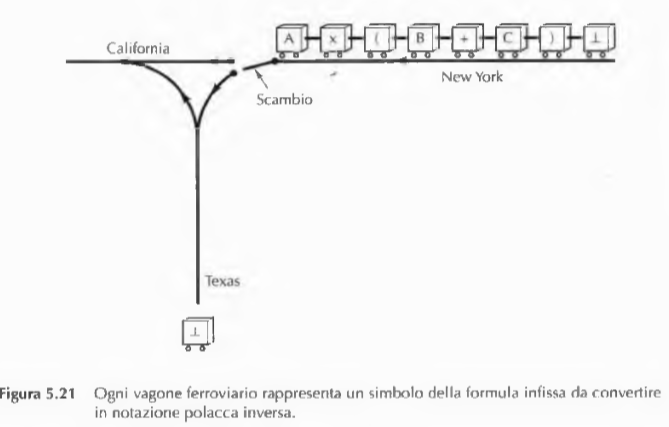
È tradizione in matematica indicare l’operatore in mezzo agli operandi, come in x + y, invece che dopo gli operandi, come in xy +. La forma con l’operatore “in” mezzo è detta notazione infissa, mentre la forma con l’operatore dopo gli operandi si chiama postfissa o anche notazione polacca inversa.

La notazione polacca inversa presenta un certo numero di vantaggi rispetto a quella 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 x b + c* vuol dire *(a x b) + c* e non *a x (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 simboli seguenti: le variabili, gli operatori binari (a due operandi) + - x / e le parentesi destra e sinistra. Il simbolo 1 delimita la formula: è anteposto al suo primo simbolo e segue l’ultimo.

La Figura 5.21 mostra un tracciato ferroviario che collega New York alla California, con al centro una diramazione che si diparte in direzione del Texas. 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.



La tabella nella Figura 5.22 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 J. va sempre in Texas.



I numeri della figura rimandano ai seguenti eventi:

1. il vagone allo scambio procede verso il Texas;

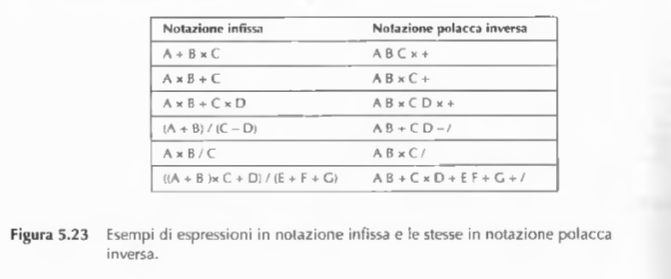
2. il vagone che per ultimo ha intrapreso la linea per il Texas inverte il percorso e prosegue per la California;

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.

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.

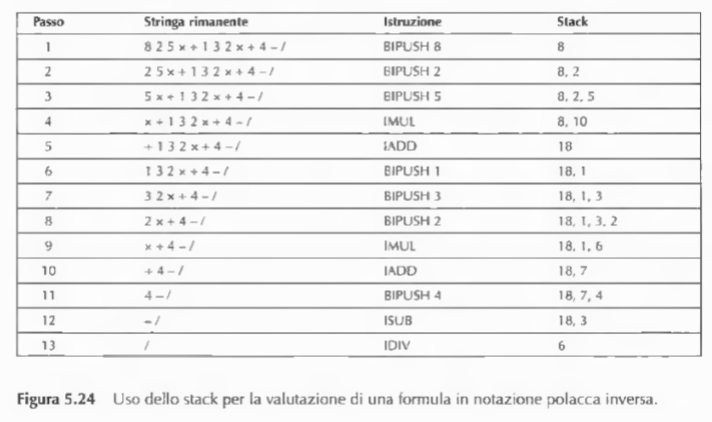


**Valutazione delle formule in notazione polacca inversa**

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 valutazione 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 5.24 illustra la valutazione della formula *(8 + 2 x 5) / (1 + 3 x 2 - 4)* nell’IJVM.

La stessa formula espressa in notazione polacca inversa è *825x+132x+4-/ .*



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 divisione e 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.

**Riassunto modalità di indirizzamento**

**Indirizzamento immediato:** operando stesso come campo.

**Indirizzamento diretto:** indirizzo completo come campo.

**Indirizzamento a registro:** registro come campo.

**Indirizzamento a registro indiretto:** registro con indirizzo come campo.

**Indirizzamento indicizzato registro:** più spiazzamento costante.

**Indirizzamento indicizzato esteso:** indirizzo come somma valori registri.

**Modalità di indirizzamento per istruzioni di salto**

Anche le istruzioni di salto (e le chiamate di procedura) necessitano di modalità di 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 di indirizzamento sensate.

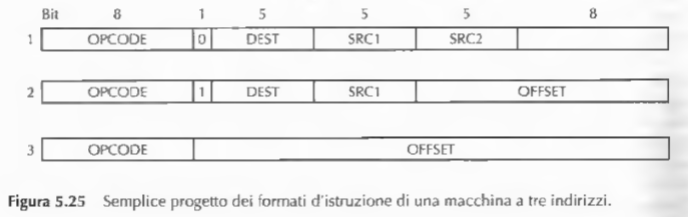
L’indirizzamento a registro indiretto consente al programma di calcolare l’indirizzo di destinazione, scriverlo in un registro e quindi effettuare il salto. È la modalità più flessibile perché l’indirizzo di destinazione può essere calcolato durante l’esecuzione, ma è anche la più incline alla generazione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

Un’altra modalità ragionevole è la modalità indicizzata, che specifica un certo offset rispetto all’indirizzo contenuto in un registro.

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.

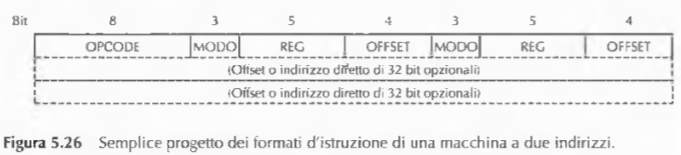
**Ortogonalità dei codici operativi e delle modalità di indirizzamento**

Dal punto di vista software, le istruzioni e l’indirizzamento dovrebbero manifestare una struttura regolare, con un numero minimo di formati di istruzioni. Tale struttura agevolerebbe molto il compilatore a produrre codice di qualità. Gli opcode dovrebbero consentire tutte le modalità di 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 di istruzioni di 32 bit della Figura 5.25 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. Il campo finale di 8 bit è inutilizzato e può essere destinato per differenziare ulteriormente le istruzioni.

A mo’ d’esempio, un solo opcode potrebbe specificare l’insieme di operazioni in virgola mobile, e il campo supplementare potrebbe distinguerle l’una dall’altra. 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 di istruzioni non del tutto generali, ma il loro uso sarebbe alquanto limitato. Si consideri ora il progetto di una macchina a due indirizzi, mostrata nella Figura 5.26, 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.

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. Potremmo anche usare una delle due modalità di 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 operandi). 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.

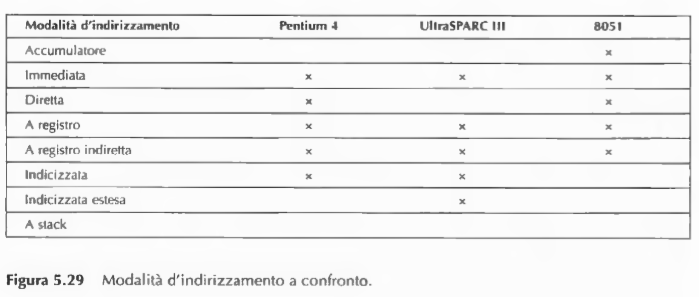
Le alternative alla Figura 5.26 sono molte. Questo progetto rende possibile l’esecuzione dell’istruzione *i = j;* con una sola istruzione di 32 bit, ammesso che *i* e *j* si trovino entrambe tra le prime 16 variabili locali. Per variabili oltre la sedicesima bisogna optare però per gli offset di 32 bit. 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.

**Analisi delle modalità di indirizzamento**

Non tutte le istruzioni possono avvalersi della gamma completa delle modalità di indirizzamento. Nella pratica, un’ISA, per essere efficace, non ha bisogno di molte modalità di indirizzamento, perciò gli aspetti più importanti di quest’ultimi in 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.

Le architetture più eleganti dispongono in genere di un numero estremamente limitato di modalità di 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.

Modalità di indirizzamento più complicate possono sì ridurre il numero di istruzioni, ma al costo dell’introduzione di sequenze di operazioni non facilmente eseguibili in modo parallelo ad altre operazioni sequenziali.



**Tipi di istruzioni**

Le istruzioni del livello ISA possono essere suddivise in gruppi facilmente rintracciabili in tutte le macchine. Ciascun computer dispone di una manciata di istruzioni aggiunte per motivi di compatibilità con modelli precedenti o a seguito di un’idea brillante di un progettista.

**Istruzioni di trasferimento dati**

Poter copiare dati da una locazione all’altra è fondamentale per tutte le operazioni. L’uso tecnico della parola “trasferimento” è diverso da quello del linguaggio comune.

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.

Ci sono due ragioni per copiare i dati da una locazione a un’altra:

- 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;

- 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 di 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 marcatore di fine dati.

**Operazioni binarie**

Le operazioni binarie sono quelle che producono un risultato dalla combinazione di due operandi.

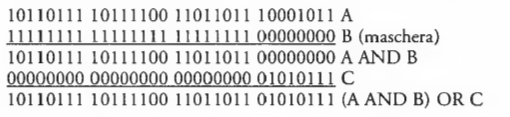
Tutti gli ISA hanno istruzioni per l’addizione e la sottrazione di interi.

Un altro insieme di operazioni binarie comprende le istruzioni booleane. Benché esistano 16 funzioni booleane in due variabili, ben poche macchine (forse nessuna) dispongono di 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.



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.



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.

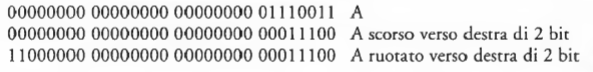
Molti dei computer odierni supportano anche un insieme di istruzioni in virgola mobile, più o meno corrispondenti alle operazioni aritmetiche sugli interi. Gran parte delle macchine mette a disposizione almeno due formati di numeri in virgola mobile, il più corto per esigenza di velocità, l’altro per 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.

**Operazioni unarie**

Le operazioni unarie prendono in ingresso un operando e restituiscono un risultato.

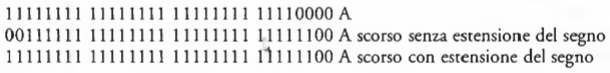
Queste istruzioni sono in genere più corte di quelle binarie, visto che contengono un indirizzo in meno, sebbene spesso richiedono la specifica di altre informazioni.

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.



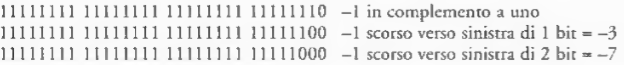
Scorrimenti e rotazioni sono utili in entrambe le direzioni. Se una parola di *n* bit è ruotata verso sinistra di *k* bit, si ottiene lo stesso risultato che ruotandola verso destra di *n - k* bit.

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.



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.

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:



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:



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 spacchettamento.

Se si vuole esaminare una parola bit per bit, la si può ruotare di 1 bit alla volta 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 di 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 di 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. Esiste, però, una 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.

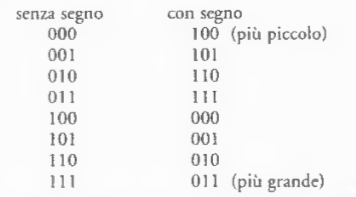
**Confronti e salti condizionali**

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 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. 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 una 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.

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 di istruzioni a tre indirizzi non incontrano difficoltà in tal senso. 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 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:



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 distruzioni per gestire entrambi gli ordinamenti.

**Invocazione di procedura**

Una procedura è un insieme di istruzioni che svolge un certo compito e che può essere invocata (chiamata) da diversi punti di un programma.

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. 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”).

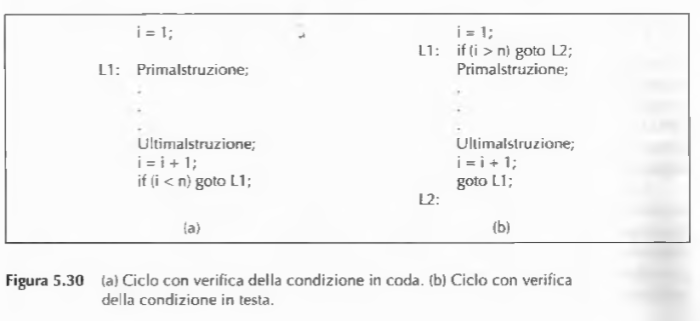
Un miglioramento sostanziale si ottiene se l’istruzione di chiamata di procedura pone l’indirizzo di ritorno in un registro, affidando alla procedura la responsabilità di salvarlo in un posto sicuro. Se la procedura è ricorsiva dovrà preoccuparsi di salvare l’indirizzo di ritorno in un posto differente 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 automaticamente senza sovrascrivere indirizzi di ritorno precedenti. In queste condizioni la ricorsione funziona perfettamente.

**Istruzioni di ciclo**

Poiché capita spesso di dover eseguire un gruppo di istruzioni un numero prefissato di volte, molte macchine dispongono di istruzioni per facilitare questo compito. Tutti gli schemi prevedono un contatore che viene incrementato o decrementato di una 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 cominciarne 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 5.30(a) ne diamo un esempio in C.



II ciclo con valutazione in coda ha la proprietà di essere eseguito sempre almeno una volta, anche se n fosse minore o uguale a 0.

La Figura 5.30(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 di informazioni su *n*, allora deve usare l’approccio della Figura 5.30(b) per poter gestire correttamente anche i casi in cui *n ≤ 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 5.30(a).

**Input/Output**

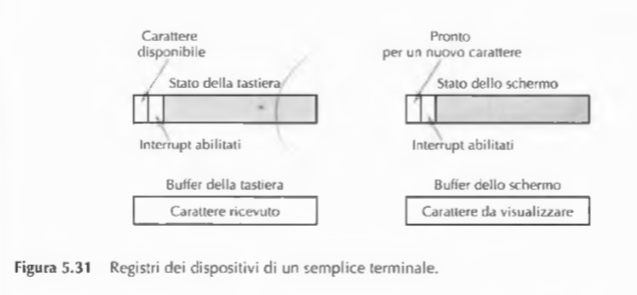
Nessun raggruppamento di istruzioni manifesta la stessa variabilità tra macchine diverse come le istruzioni di I/O. Attualmente i personal computer usano tre schemi diversi di I/O:

1) I/O programmato con attesa attiva;

2) I/O interrupt driven (cioè innescato dagli interrupt);

3) I/O con DMA.

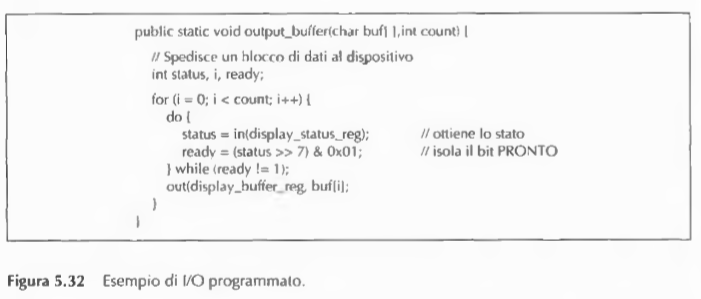
Il metodo di I/O più semplice possibile è quello programmato, impiegato di solito nei microprocessori di fascia bassa come, per esempio, i sistemi integrati o i sistemi che devono rispondere rapidamente agli stimoli esterni (sistemi in tempo reale). Queste CPU hanno in genere una sola istruzione di input 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 di 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 5.31.



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 dei loro stati 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. 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 5.32.



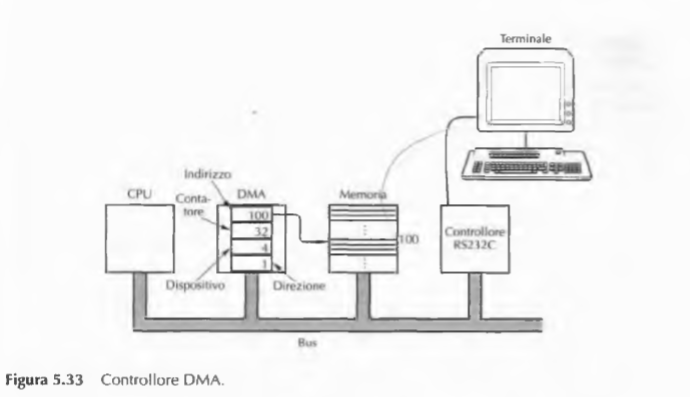
Questa procedura presenta due parametri: un array di caratteri per riporre il risultato e il numero di caratteri (count) presenti nell’array (al massimo 1024). Il corpo della procedura è costituito da un ciclo che restituisce un carattere alla volta. Per ogni carattere la CPU deve attendere innanzitutto 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). La divisione per 128 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 null’altro da eseguire. 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.

Un modo per evitare l’attesa attiva è far sì che la CPU faccia partire il dispositivo di I/O e gli impartisca l’ordine di generare un interrupt quando ha finito. Possiamo capire come funziona guardando la Figura 5.31.

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. Vale 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 Figura 5.33 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 di 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 5.33. 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 byte, il controllore DMA effettuerebbe 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 di 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. Infine, quando il contatore si azzera, 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 controllori 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”).

**Controllo del flusso**

Esso riguarda la sequenza con cui le istruzioni vengono eseguite durante l’esecuzione del programma.

In mancanza di istruzioni di salto o di chiamate di procedura, esse 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.

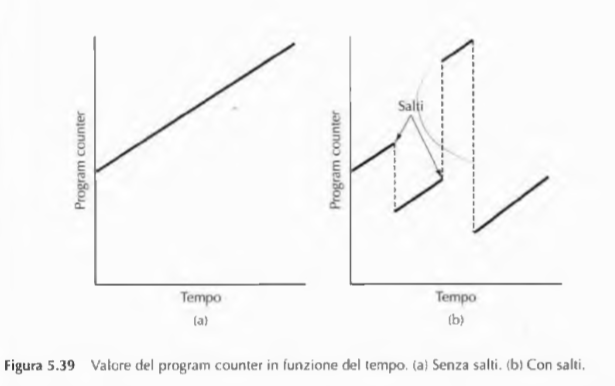
Le coroutine sono legate alle procedure e causano un’alterazione simile.

Anche le trap e gli interrupt provocano un’alterazione del flusso esecutivo quando si verificano certe condizioni speciali.

**Flusso sequenziale e diramazioni**

Il Program Counter è una funzione che viene incrementata dopo l’esecuzione di un’istruzione e punta a quella subito successiva. È inoltre lineare nella variabile tempo e aumenta di una quantità pari alla lunghezza media delle istruzioni diviso il loro tempo medio di esecuzione, come mostrato nella Figura 5.39(a).

Se invece un programma contiene salti, questa semplice relazione tra ordinamento delle istruzioni in memoria e ordine di esecuzione non vale più. Il Program Counter non è più una funzione monotona crescente nel tempo, come si evince dalla Figura 5.39(b). In ragione di ciò, diventa difficile visualizzare la sequenza di esecuzione delle istruzioni a partire dal listato.



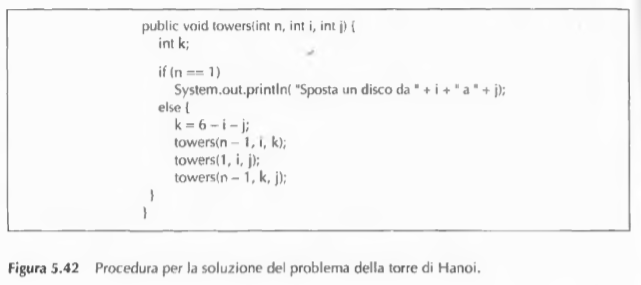
**Procedure**

Una procedura (la tecnica più importante per la strutturazione di programmi) altera il flusso esecutivo tanto quanto un salto, ma alla terminazione del suo compito ripassa il controllo al comando o all’istruzione che segue la sua chiamata.

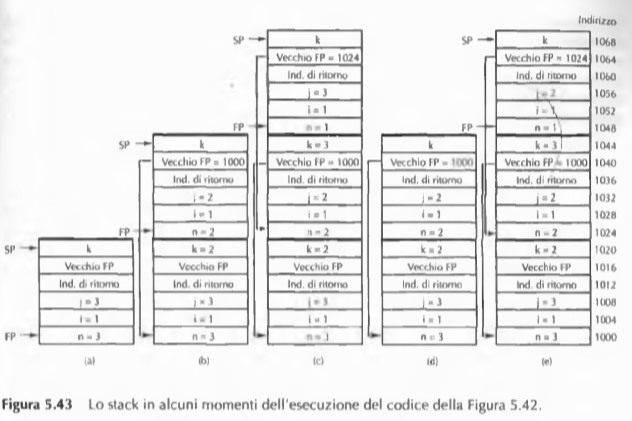
Il corpo di una procedura può essere visto come la definizione di una nuova istruzione di alto livello. La chiamata è concepibile come un’istruzione singola, senza riguardo alla sua effettiva complessità. Per la comprensione di una porzione di codice contenente una chiamata di procedura è sufficiente 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.

Per gestire le procedure ricorsive abbiamo bisogno di uno stack per memorizzare i parametri e le variabili locali a ogni invocazione. Ogni volta che una procedura viene chiamata, in cima allo stack viene allocato un record di attivazione per la procedura stessa. Il record di 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 di 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 di attivazione, FP(0), che punta a una locazione nota del record di attivazione.



La Figura 5.43 mostra il record di 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 5.43(a) mostra la situazione dello stack al termine di queste operazioni.

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 di attivazione.

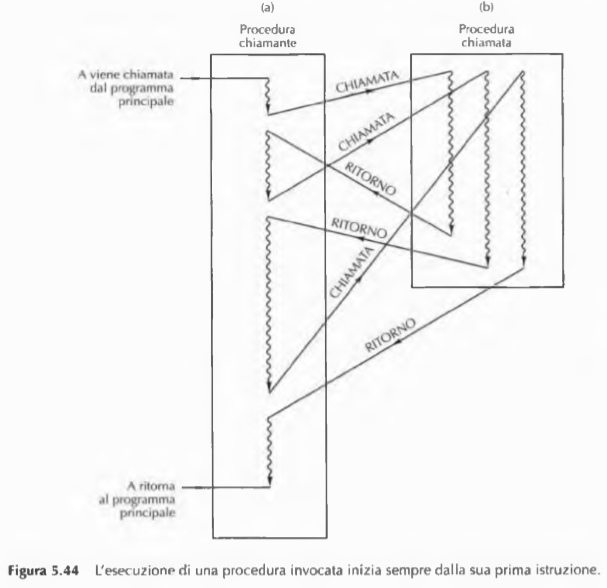
Il codice che si preoccupa della gestione del FP si chiama prologo della procedura (*procedure prolog*). All’uscita dalla procedura si rende invece necessario ripulire lo stack, che rappresenta la fase di epilogo della procedura.

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 programmatori devoti all’efficienza impareranno a evitare di scrivere molte procedure brevi, cui preferiranno programmi lunghi, monolitici e poco strutturati.

**Coroutine**

Nell’usuale sequenza di chiamata c’è una chiara distinzione tra procedura chiamante e procedura chiamata.

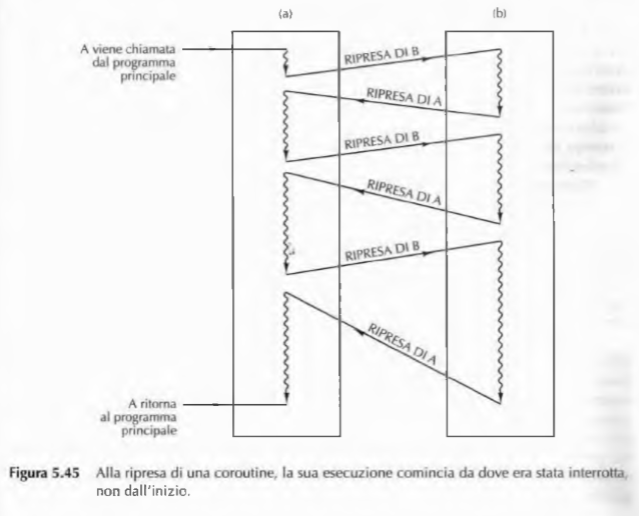
Si consideri la Figura 5.44, dove la procedura A chiama la procedura B. La procedura B svolge dei calcoli, e restituisce il controllo all’altra. A prima vista sembrerebbe una situazione simmetrica, perché né A né B costituiscono il programma principale, ma due procedure. Il controllo passa da A a B, durante la chiamata, mentre durante il rientro è trasferito da B ad A.



L’asimmetria nasce dal fatto che l’esecuzione della procedura B inizia dal suo inizio mentra la procedura A comincia dall’istruzione successiva alla chiamata. Se A richiama B quest’ultima riparte dall’inizio, mentre A mai da capo.

Questa differenza si riflette nel modo in cui il controllo è passato. 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 5.45.



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.

Vengono comunemente utilizzate per la simulazione dell’elaborazione parallela su singola CPU. Ogni coroutine gira in maniera pseuda-parallela, 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.

**Trap**

Una trap (“trappola”) è una chiamate 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 aritmetico 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 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.

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 e controllarlo manualmente. Il programmatore deve quindi 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) e richiamato ad ogni ciclo principale del programma, risparmiando così memoria. Questo tipo di test è molto conveniente in termini di costi e tempo rispetto al test svolto dal programmatore, perchè può essere sovrapposto a qualche altra operazione.

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.

**Interrupt**

Gli interrupt (“interruzioni”) sono dei cambiamenti del flusso esecutivo generati da problemi esterni al programma. Come le trap, questi interrompono il programma in esecuzione e trasferiscono il controllo ad un gestore. Al loro compimento il gestore restituisce il controllo al programma interrotto esattamente da dove era stato fermato, con il ripristino dei registri interni al momento prima dell’interruzione.

La differenza tra trap e interrupt è che le prime sono sincrone mentre le seconde no. Infatti le trap saranno richiamate sempre allo stesso punto nell’esecuzione del programma, mentre gli interrupt vengono chiamati a seconda del momento in cui l’utente preme un programma, cosa che li rende irriproducibili.

Consideriamo un esempio pratico: un calcolatore deve scrivere su schermo dei caratteri. Per prima cosa il software raccoglie i dati in un buffer e inizializza due variabili globali: *ptr* per puntare alla prima cella e *count* per memorizzare il numero di caratteri. Successivamente verifica che il terminale sia pronto e invia il primo carattere (Figura 5.31). Successivamente la CPU è libera di di eseguire un altro programma o di svolgere lavoro di altro genere.

Ecco i passi descritti compiuti sopra.

**Azioni hardware**

1) Il controllore del dispositivo attiva una linea di interrupt sul bus di sistema per dar via alla sequenza di interrupt;

2) Non appena pronta a gestire l’interrupt, la CPU attiva sul bus un segnale di conferma (acknowledgment) dell’interrupt;

3) 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;

4) La CPU preleva il vettore di interrupt dal bus e lo salva temporaneamente;

5) La CPU impila il program counter e il registro PSW sullo stack;

6) 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**

7) La routine di servizio dell’interrupt comincia con il salvare tutti i registri che utilizza per poterli ripristinare successivamente;

8) 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;

9) A questo punto è possibile leggere ogni altra informazione sull’interrupt, come il codice di stato;

10) viene gestito l’interrupt;

11) Se richiesto, viene inviato un codice speciale per specificare al dispositivo o al controllore dell’interrupt che l’interrupt è stato trattato;

12) Ripristino di tutti i registri salvati;

13) 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.

**Trasparenza degli interrupt**

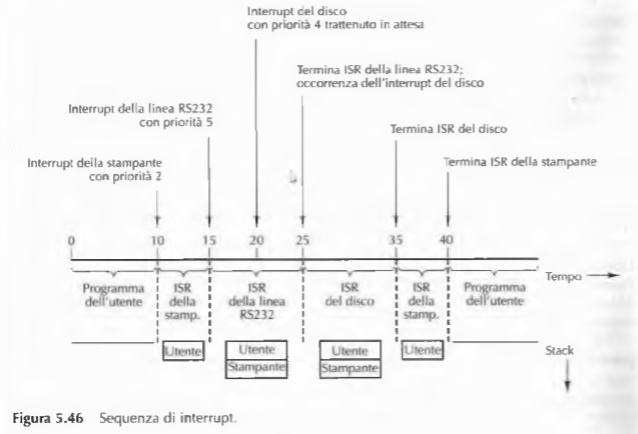
Un concetto chiave legato agli interrupt è la trasparenza, cioè che dopo la chiamata di un interrupt il sistema deve tornare allo stato precedente e questo ne semplifica la comprensione.

In presenza di un solo dispositivo di I/O, gli interrupt funzionano sempre nel modo corretto, ma nel caso ce ne siano più di 1 collegati al PC è possibile che due insieme generino contemporaneamente un interrupt.

In questi casi l’interruzione può essere gestita in due modi: il primo è far sì che un interrupt disabiliti eventuali interrupt successivi ancor prima di aggiornare i registri, con la possibilità di generare problemi ai dispositivi che non tollerano attese prolungate (per esempio se il primo carattere non è stato ancora elaborato si potrebbero perdere dei dati con i caratteri successivi); il secondo consiste nell’assegnare una priorità a ogni dispositivo di I/O (alta per i dispositivi molto critici e 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*. Quindi ogni tentativo di interruzione di dispositivi di priorità inferiore viene ignorato. 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 5.46.



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.

A partire dall’8088, tutte le CPU Intel sono state dotate di due livelli (priorità) di interrupt: mascherabile (principalmente utilizzato) e non mascherabile (usato solo da eventi catastrofici).

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 la CPU non può far sì che un dispositivo di priorità maggiore blocchi uno di priorità media ed uno di priorità bassa. Quindi i processori Intel sono usati in congiunzione con un controllore esterno di interrupt. Quando sopraggiunge il primo interrupt, 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.

**Architettura Intel IA-32 e IA-64**

Intorno all’anno 2000, la società stava arrivando al punto di aver spremuto del tutto la linea di processori IA-32. L’unica soluzione realistica per aumentare la velocità di calcolo era abbandonare IA-32 come linea principale di sviluppo e spostarsi verso un ISA completamente nuovo.

Ci sono due nuove linee:

1) EMT-64: è un rifacimento ampliato del tradizionale ISA, 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 predecessori.

2)IA-64: sviluppata congiuntamente da Intel e Hewlett Packard, si tratta di una macchina completamente a 64 bit, non di un’estensione di una macchina a 32 bit. La prima implementazione dell’architettura IA-64 è la serie Itanium.

**I problemi di IA-32**

1) È una ISA CISC con istruzioni di lunghezza variabile e una miriade di formati differenti difficili da decodificare velocemente.

2) IA-32 è un ISA orientato alla memoria ed è a due indirizzi. La maggior parte delle istruzioni referenzia la memoria e la maggior parte dei programmatori e dei compilatori non si cura di far riferimento continuo alla memoria.

3) IA-32 dispone di un insieme di registri piccolo e irregolare. Ciò implica che:

* i risultati intermedi siano riversati continuamente in memoria, causando molteplici dipendenze;
* le istruzioni devono essere eseguite fuori sequenza ed occorre un hardware complesso per riordinarle;
* per svolgere queste operazioni velocemente occorre una pipeline con molti stadi e quindi molti passaggi prima di completare l’esecuzione di una istruzione. Questo implica che la predizione dei salti deve essere precisa, ma ciò è davvero difficile. Una predizione errata richiede lo svuotamento della pipeline.

4) Per evitare il problema delle predizioni sbagliate il processore deve effettuare l’esecuzione speculativa (esegue il codice prima che venga richiesto), con tutti i problemi che comporta, specie quando i riferimenti alla memoria causano un’eccezione.

5) Disponendo di 32 bit i programmi sono limitati a 4GB di memoria.

**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 della fase di esecuzione alla fase di compilazione.

Il modello IA-64 presenta numerose funzionalità per incrementare le prestazioni:

- riduzione degli accessi in memoria: disporre una grande cache di primo livello sul chip e di una cache di secondo livello ancora più grande, vicino al chip.

- scheduling delle istruzioni: dividere il programma in una sequenza di gruppi di istruzioni, in modo che entro un certo limite le istruzioni di un gruppo non sono mai in conflitto tra di loro.

- riduzione dei salti incondizionati: utilizzo della tecnica attribuzione dei predicati.

- caricamenti speculativi: permette al compilatore di anticipare le istruzioni LOAD in posizioni antecedenti al loro effettivo bisogno, se prima iniziano è probabile che il loro risultato sia disponibile al momento opportuno.