

45. Nella Figura 3.6 I(b) si immagini di aggiungere una terza linea di input alla porta NAND che seleziona il chip PIO. Che cosa si otterrebbe se questa linea venisse connessa ad A13?
46. Si scriva un programma per simulare il comportamento di un array $m \times n$ di porte NAND a due input. Questo circuito, contenuto in un solo chip, ha j pin di input e k pin di output. I valori di j , k , m e n sono parametri della simulazione determinati a tempo di compilazione. Il programma dovrebbe iniziare leggendo una "lista dei collegamenti", in cui per ciascun collegamento sono specificati l'input e l'output. Un input può essere uno dei j pin di input oppure l'output di una porta NAND. Un output è uno dei k pin di output oppure un input di una porta NAND. Gli input non utilizzati hanno valore logico 1. Il programma, dopo aver letto la "lista dei collegamenti", dovrebbe stampare l'output di ciascuno dei 2^j possibili input. Chip come questo, composti da un array di porte, sono usati in modo diffuso per definire circuiti personalizzati su un chip; il motivo è che la maggior parte del lavoro (il collocamento dell'array di porte sul chip) è indipendente dal circuito da implementare. Da progetto a progetto cambiano soltanto i collegamenti.
47. Si scriva un programma nel linguaggio di programmazione preferito per leggere due espressioni booleane arbitrarie e vedere se rappresentano la stessa funzione. Il linguaggio di input comprende singole lettere per le variabili booleane, gli operandi AND, OR e NOT e le parentesi. Ciascuna espressione deve essere fornita su una sola linea di input. Il programma deve calcolare le tabelle di verità di entrambe le funzioni e confrontarle.



Livello di microarchitettura

Al di sopra del livello logico digitale si trova il livello di microarchitettura. Com'è illustrato nella Figura 1.2 il suo compito, consiste nell'implementare il livello ISA (*Instruction Set Architecture*, "architettura dell'insieme d'istruzioni"). Il modo in cui viene progettato il livello di microarchitettura non dipende solamente dall'ISA che si intende implementare, ma anche dagli obiettivi di costi e prestazioni del calcolatore. Molti ISA moderni, in particolare nelle architetture RISC, sono costituiti da istruzioni semplici che generalmente è possibile eseguire in un unico ciclo di clock. Nel caso di ISA più complessi, come l'insieme di istruzioni del Core i7, l'esecuzione di una singola istruzione può invece richiedere più cicli. Per eseguire un'istruzione può essere necessario localizzare gli operandi all'interno della memoria, leggerli e infine memorizzare il risultato nuovamente in memoria. Spesso l'ordinamento di queste operazioni all'interno di una singola istruzione porta a una strategia di controllo diversa rispetto a quella degli ISA più semplici.

4.1 Esempio di microarchitettura

Sarebbe preferibile introdurre gli argomenti del capitolo spiegando i principi generali su cui si basa la progettazione della microarchitettura; sfortunatamente però non esistono principi generali: ogni ISA rappresenta infatti un caso particolare. Per questo motivo analizzeremo in modo dettagliato un esempio pratico. Come avevamo promesso nel Capitolo 1 abbiamo scelto come ISA di esempio un sottoinsieme della Java Virtual Machine e, dato che questo sottoinsieme contiene solo istruzioni su interi, abbiamo deciso di chiamarlo IJVM. Nel capitolo successivo tratteremo l'intera JVM.

Inizieremo descrivendo la microarchitettura sopra la quale implementeremo IJVM. Come abbiamo già visto nel corso del Capitolo 1, molte delle architetture che, come IJVM, contengono istruzioni complesse sono implementate mediante la microprogrammazione. IJVM, pur essendo un insieme di piccole dimensioni, rappresenta tuttavia un buon punto di partenza per descrivere il controllo e l'ordinamento delle istruzioni.

La nostra microarchitettura conterrà un microprogramma (registrato in una ROM) il cui compito sarà quello di prelevare, decodificare ed eseguire le istruzioni IJVM. Dato che abbiamo bisogno di un piccolo microprogramma che guidi in modo efficiente le singole porte logiche non possiamo utilizzare l'interprete Oracle JVM; questo interprete è stato infatti scritto in C per essere portabile e non può controllare l'hardware. Dato che l'hardware realmente utilizzato consiste solamente nei componenti elementari descritti nel Capitolo 3, in teoria il lettore, dopo aver compreso interamente il capitolo, potrebbe uscire di casa, farsi una bella scorta di transistor ed essere in grado di costruire da solo questo sottoinsieme della macchina JVM. Agli studenti che riusciranno a portare a termine il compito verrà assegnato un credito in più (e verrà offerta loro anche una seduta psichiatrica gratuita!).

Un modello convenzionale per progettare una microarchitettura consiste nel concepirla come un problema di programmazione, in cui ogni istruzione del livello ISA è una funzione che deve essere richiamata dal programma principale. In questo modello il programma principale è un semplice ciclo senza fine che determina la funzione da invocare, la richiama e poi ricomincia la propria esecuzione, come suggerito dalla Figura 2.3.

Il microprogramma ha delle variabili che costituiscono lo stato del calcolatore. Ogni funzione cambia il valore di almeno una delle variabili, modificando di conseguenza lo stato del calcolatore. Il *Program Counter* (PC, “contatore d’istruzioni”) è una delle variabili che fanno parte dello stato e indica la locazione di memoria contenente la successiva funzione (cioè la successiva istruzione ISA) da eseguire. Durante l’esecuzione di un’istruzione il PC viene fatto avanzare in modo da farlo puntare all’istruzione successiva.

Le istruzioni IJVM sono corte e dall’aspetto semplice. Ogni istruzione è composta da alcuni campi, di solito uno o due, con uno scopo predefinito. Il primo campo di ogni istruzione è il *codice operativo* (*opcode*), che identifica il tipo d’istruzione, indicando se è di tipo ADD, di tipo BRANCH o altro. In molte istruzioni è presente un campo aggiuntivo che specifica l’operando: per esempio le istruzioni che accedono a una variabile locale devono indicare a *quale* variabile si riferiscono.

Questo modello di esecuzione, chiamato talvolta *fetch-decodifica-esecuzione*, è utile a livello astratto e può anche costituire la base per l’implementazione di ISA complessi come IJVM. In seguito descriveremo come funziona questo modello, che aspetto ha la sua microarchitettura e com’è controllato dalle microistruzioni. L’insieme delle microistruzioni compone il microprogramma e ciascuna di loro ha il controllo del percorso dati durante un ciclo. Nel corso del capitolo presenteremo e tratteremo in modo dettagliato il microprogramma.

4.1.1 Percorso dati

Il *percorso dati* è quella parte della CPU che contiene la ALU, i suoi input e i suoi output. Il percorso dati del nostro esempio è mostrato nella Figura 4.1. Anche se è stato attentamente ottimizzato per interpretare i programmi IJVM esso è tuttavia molto simile ai percorsi dati della maggior parte delle macchine. Il nostro percorso dati contiene dei registri a 32 bit, a cui abbiamo assegnato nomi come PC, SP e MDR. Alcuni di questi nomi sono già familiari, ma è importante capire che è possibile accedere a questi registri

solamente a livello di microarchitettura (cioè dal microprogramma). Il motivo per cui sono stati assegnati tali nomi è che generalmente questi registri memorizzano il valore di una variabile omonima appartenente all’architettura del livello ISA. La maggior parte dei registri può inviare il proprio contenuto sul bus B, collegato in input alla ALU. L’output della ALU guida invece lo shifter, che a sua volta invia il proprio risultato sul bus C; i valori di quest’ultimo possono essere scritti allo stesso tempo in uno o più registri. Per il momento non è presente il bus A (che verrà aggiunto in una fase successiva).

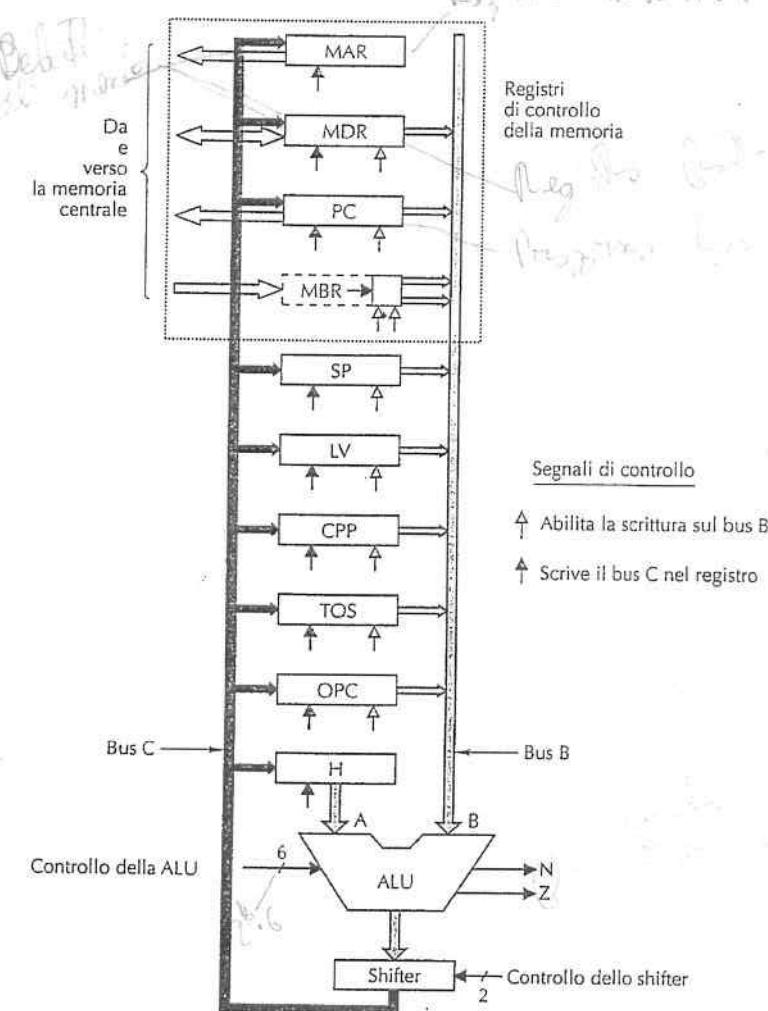


Figura 4.1 Percorso dati della microarchitettura utilizzata per l’esempio di questo capitolo.

La ALU è identica a quella vista nelle Figure 3.18 e 3.19, e la sua funzione è determinata da sei linee di controllo. Nella Figura 4.1 il trattino diagonale con a fianco il numero “6” indica che ci sono sei linee per il controllo della ALU. Fra queste F_0 e F_1 determinano l’operazione della ALU, ENA e ENB abilitano individualmente i due input, INVA inverte l’input di sinistra e INC forza la presenza di un riporto nel bit meno significativo, sommando quindi 1 al risultato. Non tutte le 64 combinazioni delle linee della ALU hanno un ruolo significativo.

La Figura 4.2 mostra alcune delle combinazioni più interessanti. Non tutte queste funzioni sono necessarie per IJVM, ma molte torneranno utili quando tratteremo l’intero insieme JVM. In molti casi esistono inoltre molteplici possibilità per raggiungere lo stesso risultato. Nello schema + e – indicano la somma e la sottrazione aritmetica; $-A$ indica il complemento a due di A .

F_0	F_1	ENA	ENB	INVA	INC	Funzione
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	A
1	0	1	1	0	0	B
1	1	1	1	0	0	$A + B$
1	1	1	1	0	1	$A + B + 1$
1	1	1	0	0	1	$A + 1$
1	1	0	1	0	1	$B + 1$
1	1	1	1	1	1	$B - A$
1	1	0	1	1	0	$B - 1$
1	1	1	0	1	1	$-A$
0	0	1	1	0	0	$A \text{ AND } B$
0	1	1	1	0	0	$A \text{ OR } B$
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1

Figura 4.2 Segnali di controllo in ingresso alla ALU e corrispondenti funzioni.

La ALU della Figura 4.1 richiede due dati in ingresso: un input sinistro (A) e uno destro (B). A quello di sinistra è collegato un registro H di mantenimento (*holding*), mentre al registro di destra è collegato il bus B . Quest’ultimo può essere caricato con i valori di una qualsiasi delle nove sorgenti indicate dalle nove frecce grigie la cui punta tocca il bus. Nel corso del capitolo tratteremo un’architettura di tipo differente, basata su due bus, che ci costringerà a effettuare nuove scelte e nuovi bilanciamenti.

È possibile caricare un valore in H scegliendo una funzione della ALU il cui compito sia semplicemente quello di far passare al suo interno l’input di destra (proveniente dal bus B) per poi porlo in output senza alcuna modifica. Una simile funzione può esse-

re ottenuta attraverso la somma di due input della ALU negando però il segnale ENA, di modo che l’input di sinistra sia forzato al valore zero. Sommando zero all’input proveniente dal bus B si ottiene come risultato lo stesso valore presente sul bus B . Per poter memorizzare questo risultato in H lo si può far passare attraverso lo shifter senza fargli subire alcuna modifica.

Oltre alle funzioni appena citate, per gestire l’output della ALU è possibile utilizzare altre due linee di controllo. SLL8 (*Shift Left Logical*, “scorrimento logico a sinistra”) trasla il valore a sinistra di un byte, impostando gli 8 bit meno significativi a 0. SRA1 (*Shift Right Arithmetic*, “scorrimento aritmetico a destra”) trasla invece il valore di 1 bit a destra, lasciando inalterato il bit più significativo.

È possibile leggere o scrivere esplicitamente lo stesso registro durante un unico ciclo. Per incrementare SP di 1 è possibile portare il valore di SP sul bus B , disabilitare l’input sinistro della ALU, abilitare il segnale INC e memorizzare il risultato nuovamente all’interno di SP. Com’è possibile leggere e scrivere un registro nello stesso ciclo senza generare dei dati incoerenti? La soluzione risiede nel fatto che la lettura e la scrittura sono in realtà eseguite in momenti diversi all’interno del ciclo. Quando si seleziona un registro come input destro della ALU i suoi valori vengono inseriti nel bus B in una fase iniziale del ciclo e vengono poi continuamente mantenuti sul bus per tutta la durata del ciclo. La ALU compie quindi le proprie operazioni generando un risultato che giunge al bus C passando attraverso lo shifter. Verso la fine del ciclo, quando gli output della ALU e dello shifter sono sicuramente stabili, un segnale di clock dà inizio alla memorizzazione del contenuto del bus C all’interno di uno o più registri. Uno di questi registri potrebbe tranquillamente essere lo stesso dal quale proveniva l’input del bus B . Seguendo la modalità appena descritta, che sfrutta la precisa temporizzazione del percorso dati, è possibile leggere e scrivere lo stesso registro durante un solo ciclo.

Temporizzazione del percorso dati

La Figura 4.3 mostra la temporizzazione degli eventi sul percorso dati. All’inizio di ogni ciclo di clock viene generato un breve impulso. Come mostrato nella Figura 3.20(c), questo impulso può essere determinato dal clock principale. In corrispondenza del fronte di discesa dell’impulso vengono impostati i bit che piloteranno tutte le porte logiche. Quest’operazione richiede un intervallo di tempo finito e conosciuto a priori: Δw . Il registro richiesto viene quindi selezionato e il suo contenuto viene portato sul bus B ; prima che il suo valore diventi stabile occorre attendere un tempo Δx . A questo punto la ALU e lo shifter, i cui circuiti combinatori hanno funzionato continuamente, hanno finalmente dati validi su cui operare. I loro output diventano stabili dopo un altro intervallo temporale, Δy . Passato un ulteriore tempo Δz i risultati vengono propagati lungo il bus C fino ai registri in cui possono essere caricati in corrispondenza del fronte di salita dell’impulso successivo. Il caricamento all’interno dei registri dovrebbe essere pilotato dal fronte del segnale ed essere veloce; in questo modo, anche se alcuni dei registri di input vengono modificati, gli effetti di queste modifiche giungeranno sul bus C solo dopo un tempo sufficientemente lungo rispetto al momento in cui sono stati caricati i registri. Inoltre, in corrispondenza del fronte di salita dell’impulso, il registro che stava alimentando il bus B smette di farlo, in preparazione del ciclo successivo. Nella figura sono indicati MPC, MIR e la memoria; i loro ruoli saranno presentati a breve.

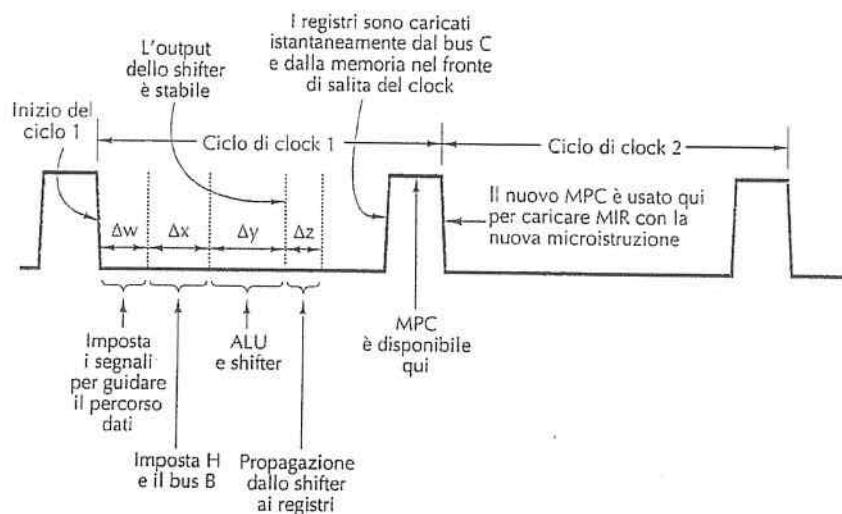


Figura 4.3 Temporizzazione di un ciclo di clock del percorso dati.

È importante rendersi conto che all'interno del percorso dati esiste un tempo di propagazione finito, anche se non sono presenti elementi di memorizzazione. Modificare il valore sul bus B non modifica il bus C se non dopo un intervallo di tempo finito (dovuto ai ritardi che vengono introdotti a ogni passo). Di conseguenza, anche se un'operazione di scrittura modifica uno dei registri di input, il valore sarà reinserito in modo sicuro al suo interno, molto tempo prima che il valore (non più corretto) che si sta inserendo sul bus B (oppure H) possa raggiungere la ALU.

Per far sì che questa architettura funzioni è necessaria una rigida temporizzazione, un lungo ciclo di clock, un ritardo di propagazione attraverso la ALU conosciuto a priori e un veloce caricamento dei registri dal bus C. Pur essendo un compito complesso, grazie a un'attenta opera di ingegneria è possibile progettare il percorso dati in modo che funzioni correttamente in ogni momento. Le macchine reali funzionano proprio in questa maniera.

Un modo diverso per vedere il ciclo del percorso dati consiste nel pensare che esso sia implicitamente diviso in più sottocicli e che l'inizio del sottociclo 1 sia guidato dal fronte di discesa del clock. Di seguito sono elencate le attività che si svolgono durante i sottocicli, accompagnate (tra parentesi) dalla durata del sottociclo corrispondente.

1. Si impostano i segnali di controllo (Δw).
2. I registri vengono caricati nel bus B (Δx).
3. La ALU e lo shifter svolgono le loro operazioni (Δy).
4. I risultati vengono propagati lungo il bus C e ritornano nei registri (Δz).

L'intervallo di tempo che segue Δz ha una certa tolleranza, perché i tempi non sono esatti. In corrispondenza del fronte di salita del ciclo successivo i risultati vengono memorizzati nei registri.

Abbiamo detto che è possibile pensare ai sottocicli come se fossero definiti *implicitamente*. Con questo vogliamo intendere che nessun impulso di clock o altro segnale esplicito comunica alla ALU quando funzionare né dice ai risultati di entrare nel bus C. In realtà la ALU e lo shifter funzionano in continuazione; tuttavia i loro input vanno considerati come inconsistenti fino al tempo $\Delta w + \Delta x$ dopo il fronte di discesa del clock. Analogamente anche i loro output sono inconsistenti finché non sia trascorso un tempo $\Delta w + \Delta x + \Delta y$ dopo il fronte di discesa del clock. Gli unici segnali esplicativi che guidano il percorso dati sono il fronte di discesa del clock, che fa partire il ciclo del percorso dati, e quello di salita, che carica i registri dal bus C. I limiti degli altri sottocicli sono determinati implicitamente dai tempi di propagazione insiti nei circuiti utilizzati. È responsabilità dell'ingegnere progettista assicurarsi che il tempo $\Delta w + \Delta x + \Delta y + \Delta z$ giunga sufficientemente in anticipo rispetto al fronte di salita del clock, in modo che il caricamento dei registri possa funzionare in ogni momento.

Operazioni della memoria

La nostra macchina ha due modi diversi per comunicare con la memoria: una porta a 32 bit con indirizzi espressi in parole e una porta a 8 bit con indirizzi espressi in byte. Come mostra la Figura 4.1, la porta a 32 bit è controllata da due registri, MAR (Memory Address Register, "registro degli indirizzi di memoria") e MDR (Memory Data Register, "registro dei dati di memoria"). La porta a 8 bit è controllata invece da un unico registro, PC, che legge 1 byte negli 8 bit meno significativi di MBR. Questa porta può soltanto leggere i dati dalla memoria.

Ciascuno dei registri (così come tutti gli altri della Figura 4.1) è comandato da uno o due segnali di controllo. Una freccia bianca sotto un registro indica un segnale di controllo che abilita l'output del registro verso il bus B. Dato che MAR non ha una connessione al bus B, esso non ha il segnale per l'abilitazione. Neanche H ne è provvisto, dato che è l'unico possibile input sinistro della ALU ed è quindi sempre abilitato.

Una freccia nera sotto un registro indica un segnale di controllo che scrive nel registro (cioè "carica") un valore proveniente dal bus C. Dato che MBR non può essere caricato dal bus C, non ha un segnale di scrittura (anche se dispone di altri due segnali, descritti più avanti). Per iniziare una lettura o una scrittura occorre caricare il registro di memoria appropriato e successivamente inviare alla memoria un segnale di scrittura (non mostrato nella Figura 4.1).

MAR contiene gli indirizzi espressi in parole, in cui i valori 0, 1, 2, e così via, si riferiscono quindi a parole consecutive. PC contiene invece gli indirizzi espressi in byte e quindi i valori 0, 1, 2, e così via, fanno riferimento a byte consecutivi. Se si inserisce in PC il valore 2 e si fa partire un'operazione di lettura, il byte 2 della memoria verrà letto e inserito negli 8 bit meno significativi di MBR. Se invece si inserisce il valore 2 in MAR e si fa partire una lettura, saranno i byte 8–11 (cioè la parola 2) della memoria a essere letti e inseriti in MDR.

Queste due diverse modalità di accesso sono necessarie poiché MAR e PC saranno utilizzati per far riferimento a due parti diverse della memoria. In seguito risulterà più chiaro il motivo di questa distinzione, ma per il momento basta dire che la combinazione MAR/MDR è utilizzata per leggere e scrivere parole di dati del livello ISA, mentre la combinazione PC/MBR è utilizzata per leggere il programma eseguibile del livello ISA, che consiste in un flusso di byte. Tutti gli altri registri contenenti indirizzi, come MAR, utilizzano indirizzi espressi in parole.

Nelle reali implementazioni è presente un'unica memoria, orientata al byte. Attraverso un semplice espediente è possibile consentire a MAR di contare il numero di parole (cosa necessaria per il modo in cui JVM è definita) anche se gli indirizzi della memoria fisica sono espressi in byte. Quando MAR viene portato sul bus degli indirizzi i suoi 32 bit non vengono mappati direttamente sulle 32 linee, da 0 a 31. Al contrario il bit 0 di MAR viene collegato alla linea 2 del bus degli indirizzi, il bit 1 di MAR viene collegato alla linea 3 del bus degli indirizzi e così via. I 2 bit più alti di MAR vengono scartati, dato che sono necessari soltanto per indirizzi superiori a 2^{32} , nessuno dei quali è significativo per la nostra macchina che ha un limite d'indirizzamento di 4 GB. In tal modo, quando MAR vale 1, viene posto sul bus l'indirizzo 4; quando MAR vale 2, viene posto l'indirizzo 8, e così via. Questo espediente è illustrato nella Figura 4.4.



Figura 4.4 Corrispondenza tra i bit di MAR e i bit del bus dell'indirizzo.

Com'è già stato menzionato i dati letti dalla memoria attraverso la porta a 8 bit sono restituiti all'interno di MBR, un registro a 8 bit. MBR può essere copiato nel bus B in due modi distinti: con o senza segno. Quando si richiede un valore senza segno la parola a 32 bit inserita nel bus B contiene il valore di MBR negli 8 bit meno significativi, mentre i restanti 24 bit sono impostati a 0. I valori senza segno sono utili come indici di tabelle oppure quando occorre assemblare un intero a 16 bit a partire da 2 byte consecutivi (e senza segno) del flusso dati dell'istruzione.

L'altra possibilità per convertire il registro MBR a 8 bit in una parola a 32 bit consiste nel trattarlo come un valore con segno compreso tra -128 e +127 e utilizzare questo numero per generare una parola a 32 bit che abbia lo stesso valore numerico. Questa conversione viene effettuata mediante un procedimento chiamato estensione del segno

che consiste nel duplicare il bit del segno (quello più a sinistra) di MBR nei 24 bit più alti del bus B. Quando si sceglie questa tecnica i 24 bit più alti saranno tutti 0 oppure tutti 1, a seconda che il bit più a sinistra di MBR valga 0 oppure 1.

La scelta tra convertire gli 8 bit di MBR in un valore a 32 bit con o senza segno prima di copiarlo sul bus B è determinata dal segnale di controllo (indicato nella Figura 4.1 dalle frecce bianche sotto MBR) che è asserito. La necessità di distinguere tra queste due opzioni giustifica la presenza delle due frecce. Il rettangolo tratteggiato che nella figura si trova alla sinistra di MBR indica che è possibile far sì che il registro MBR a 8 bit si comporti come una sorgente a 32 bit del bus B.

4.1.2 Microistruzioni

Per controllare il percorso dati della Figura 4.1 abbiamo bisogno di 29 segnali suddivisibili in cinque gruppi funzionali:

- 9 segnali per controllare la scrittura dei dati dal bus C all'interno dei registri;
- 9 segnali per controllare l'abilitazione dei registri sul bus B per l'input della ALU;
- 8 segnali per controllare le funzioni della ALU e dello shifter;
- 2 segnali (non mostrati) per indicare alla memoria di leggere (scrivere) attraverso MAR (MDR);
- 1 segnale (non mostrato) per indicare il prelievo dalla memoria attraverso PC o MBR.

I valori di questi 29 segnali specificano le operazioni da eseguire durante un ciclo del percorso dati. Un ciclo consiste nel portare i valori dei registri sul bus B, propagare i segnali attraverso la ALU e lo shifter, guidarli sul bus C e infine scrivere i risultati nel registro o nei registri appropriati. Inoltre, nel caso in cui sia asserito il segnale per una lettura dalla memoria, l'operazione viene fatta iniziare alla fine del ciclo del percorso dati, dopo che MAR è stato caricato. Alla fine del ciclo *seguente* i dati della memoria sono disponibili in MBR oppure in MDR e sono utilizzabili nel ciclo *ancora successivo*. In altre parole una lettura della memoria (su una delle due porte) iniziata alla fine del ciclo k trasmette dati che non possono essere utilizzati nel ciclo $k+1$, ma soltanto a partire dal ciclo $k+2$.

Questo comportamento apparentemente contraddittorio è spiegato nella Figura 4.3. Durante il ciclo 1 i segnali di controllo della memoria vengono generati soltanto verso la fine del ciclo, subito dopo il momento in cui MAR e PC sono caricati in corrispondenza del fronte di salita del clock. Assumeremo che la memoria inserisca i propri risultati nei bus di memoria entro un ciclo, di modo che MBR e/o MDR possano essere caricati, insieme a tutti gli altri registri, nel successivo fronte di salita del clock.

Detto in altre parole, carichiamo MAR alla fine del ciclo del percorso dati e poco dopo facciamo partire l'operazione di memoria. Di conseguenza non possiamo aspettarci che i risultati di un'operazione di lettura siano già disponibili in MDR all'inizio del ciclo successivo, soprattutto se la larghezza dell'impulso di clock è breve. Se la memoria richiede un ciclo di clock non c'è tempo a sufficienza; deve per forza passare un ciclo del percorso dati tra l'inizio di una lettura della memoria e l'utilizzo del risultato. Ovvia-

mente durante questo ciclo è possibile eseguire altre operazioni, a patto che queste non necessitino di parole dalla memoria.

L'ipotesi che la memoria richieda un ciclo per eseguire la propria operazione è equivalente ad assumere che la frequenza di successi della cache di primo livello sia pari al 100%. Questa assunzione non è mai vera, ma, per gli scopi che ci siamo posti, sarebbe troppo complesso considerare un tempo di ciclo della memoria di durata variabile.

Dato che MBR e MDR sono caricati insieme a tutti gli altri registri in corrispondenza del fronte di salita del clock, essi possono essere letti nei cicli in cui si sta svolgendo una nuova lettura della memoria. Essi restituiscano valori vecchi in quanto la lettura della memoria non ha ancora avuto il tempo di sovrascriverli e aggiornarli. Tuttavia questa situazione non presenta ambiguità; finché i nuovi valori non siano caricati in MBR e MDR nel fronte di salita del clock, i valori precedenti sono ancora presenti e utilizzabili. Dato che una lettura richiede solamente un ciclo, è possibile eseguire letture in sequenza durante due cicli consecutivi. È possibile inoltre utilizzare nello stesso momento entrambe le porte di memoria, anche se tentare di leggere e scrivere simultaneamente lo stesso byte genera risultati indefiniti.

In alcuni casi può essere utile scrivere l'output presente nel bus C in più di un registro, mentre in nessun caso ha senso abilitare nello stesso momento più di un registro sul bus B (in alcuni casi reali ciò potrebbe addirittura essere dannoso). Con pochi circuiti aggiuntivi è possibile ridurre il numero di bit necessari per la selezione delle sorgenti che alimentino il bus B. Ci sono soltanto nove possibili registri di input che possono guidare il bus B (considerando separatamente le versioni con e senza segno di MBR). Possiamo quindi codificare in 4 bit l'informazione del bus B e utilizzare un decodificatore per generare 16 segnali di controllo, 7 dei quali non vengono utilizzati. Se si trattasse di un progetto commerciale gli ingegneri subirebbero forti pressioni dai loro capi per costringerli a sbarazzarsi di uno dei registri, in modo da rendere sufficienti soltanto 3 bit. In quanto accademici noi possiamo permetterci l'enorme lusso di sprecare 1 bit semplicemente per ottenere un'architettura più semplice e ordinata.

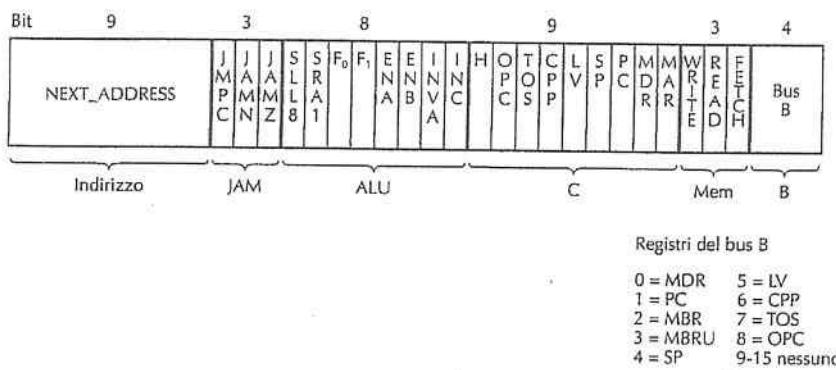


Figura 4.5 Formato delle microistruzioni di Mic-1 (da descrivere brevemente).

A questo punto possiamo controllare il percorso dati con $9 + 4 + 8 + 2 + 1 = 24$ segnali, quindi con 24 bit. Tuttavia questi bit controllano il percorso dati soltanto per un ciclo. La seconda parte del controllo consiste invece nel determinare che cosa deve essere effettuato durante il ciclo successivo. Per includere questo aspetto nel progetto del controllore definiremo un formato che ci permetterà di descrivere le operazioni da eseguire utilizzando i 24 bit di controllo più due campi aggiuntivi: NEXT_ADDRESS e JAM. Il loro contenuto sarà descritto a breve. La Figura 4.5 mostra un possibile formato, diviso in sei gruppi (elencati sotto l'istruzione) e contenente i 36 segnali seguenti.

- Addr – Contiene l'indirizzo di una potenziale successiva microistruzione.
- JAM – Determina come viene selezionata la successiva microistruzione.
- ALU – Seleziona le funzioni della ALU e dello shifter.
- C – Seleziona quali registri sono scritti dal bus C.
- Mem – Seleziona la funzione della memoria.
- B – Seleziona la sorgente del bus B; la codifica è mostrata nella figura.

In teoria l'ordinamento dei gruppi è arbitrario, anche se in realtà quello della Figura 4.6 è stato scelto attentamente in modo da minimizzare l'incrocio fra le linee. Nei diagrammi schematici simili alla Figura 4.6 l'incrocio fra linee spesso corrisponde a collegamenti che si incrociano sui chip. Dato che causano difficoltà nei progetti è buona norma cercare di minimizzarli.

4.1.3 Unità di controllo microprogrammata: Mic-1

Finora abbiamo descritto come viene controllato il microprogramma, ma non abbiamo ancora spiegato come si decide quale dei segnali di controllo abilitare durante ciascun ciclo. Ciò è determinato da un sequenzializzatore che ha la responsabilità di far avanzare passo passo la sequenza di operazioni necessarie per eseguire una singola istruzione ISA.

Durante ogni ciclo il sequenzializzatore deve produrre due tipi d'informazione:

1. lo stato di ogni segnale di controllo del sistema;
2. l'indirizzo della microistruzione da eseguire subito dopo.

La Figura 4.6 è un diagramma a blocchi dettagliato dell'intera microarchitettura della nostra macchina di esempio, che chiameremo Mic-1. A prima vista potrebbe fare un po' paura, ma vale la pena studiarlo attentamente. Quando avremo compreso il significato di tutti i rettangoli e di tutte le linee della figura, saremo sulla giusta strada per una piena comprensione del livello di microarchitettura. Il diagramma a blocchi è composto da due parti principali: il percorso dati, sulla sinistra, già approfonditamente analizzato e la sezione di controllo, sulla destra, che considereremo a partire da questo momento.

L'elemento più grande e più importante della sezione di controllo è una memoria chiamata memoria di controllo. Anche se a volte viene implementata come un insieme di porte logiche, può essere utile pensarla come una memoria che contiene l'intero microprogramma. In generale ci riferiremo a essa con il nome di memoria di controllo per non confonderla con la memoria principale, a cui si accede mediante i registri MBR e

MDR. In ogni caso, dal punto di vista funzionale, la memoria di controllo è semplicemente un circuito che memorizza le microistruzioni invece che le istruzioni ISA. Nel nostro caso contiene 512 parole, consistenti in una microistruzione a 36 bit del tipo mostrato nella Figura 4.5. In realtà non sono necessarie tutte queste parole, ma (per ragioni che spiegheremo tra poco) abbiamo bisogno di indirizzi per 512 parole distinte.

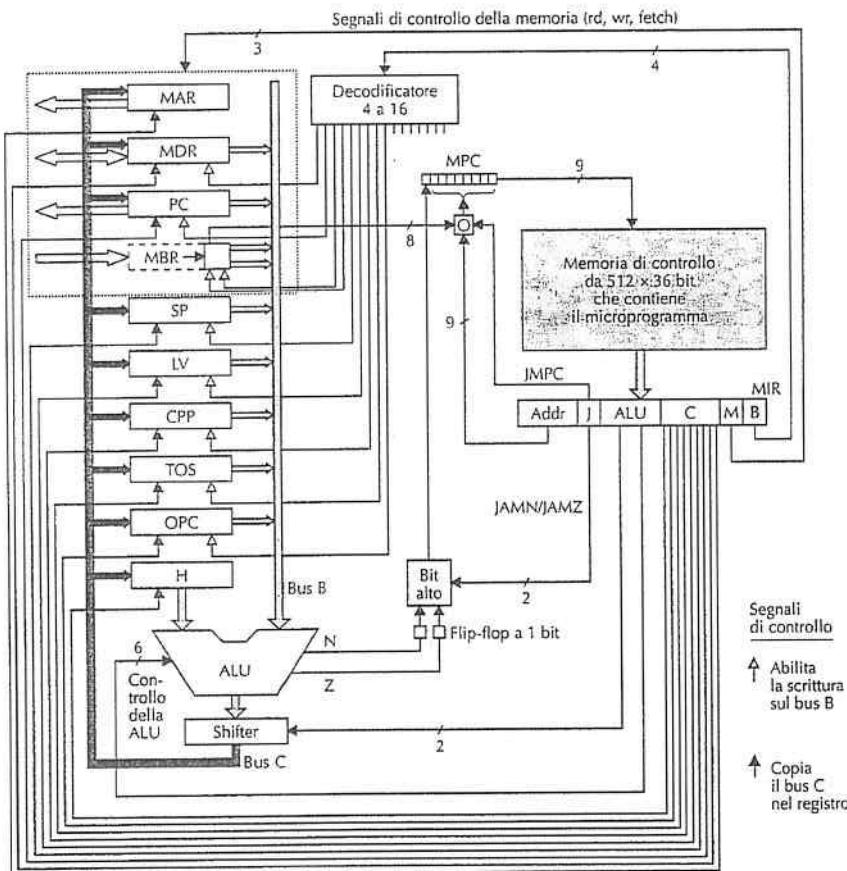


Figura 4.6 Diagramma a blocchi del nostro esempio di microarchitettura di Mic-1.

Vai a 302

La memoria di controllo differisce dalla memoria centrale per un aspetto importante: le istruzioni della memoria centrale sono sempre eseguite nell'ordine determinato dagli indirizzi (tranne nel caso delle diramazioni), mentre così non è nel caso delle microistruzioni. Nella Figura 2.3 incrementare il contatore di programma significa che la suc-

siva istruzione da eseguire corrisponde a quella che segue l'istruzione corrente all'interno della memoria.

I microprogrammi richiedono maggior flessibilità (dato che le sequenze delle microistruzioni tendono a essere brevi) e quindi, solitamente, ogni microistruzione specifica in modo esplicito il proprio successore.

Anche la memoria di controllo necessita di un proprio registro di indirizzo e di un proprio registro dei dati (ma non dei segnali di lettura e scrittura dato che la memoria viene letta in continuazione). Chiameremo MPC (*MicroProgram Counter*) il registro degli indirizzi della memoria di controllo. Questo nome è incongruo, dato che le locazioni non seguono un ordine preciso; esse sono specificate in modo esplicito e quindi non c'è bisogno di effettuare alcun conteggio (ma come si fa a mettere in discussione la tradizione?). Il registro dei dati della memoria viene invece chiamato MIR (*MicroInstruction Register*, “registro della microistruzione corrente”). Il suo ruolo consiste nel memorizzare la microistruzione in corso di esecuzione, i cui bit determinano i segnali di controllo che guidano il percorso dati.

Nella Figura 4.6 il registro MIR contiene gli stessi sei gruppi mostrati nella Figura 4.5. I gruppi Addr e J (iniziale di JAM) controllano la selezione della microistruzione successiva e verranno illustrati a breve. Il gruppo della ALU contiene 8 bit che selezionano la funzione della ALU e guidano lo shifter. I bit C indicano in quali registri verrà caricato l'output della ALU dal bus C. I bit M controllano le operazioni della memoria.

Infine gli ultimi 4 bit guidano il decodificatore, che determina quale registro deve essere portato sul bus B. Nel nostro esempio abbiamo scelto di utilizzare un decodificatore standard da 4 a 16, anche se servono solamente nove possibilità. In un progetto ottimizzato anche nei minori dettagli si sarebbe utilizzato un decodificatore da 4 a 9. In questo caso si è deciso di utilizzare un circuito standard già disponibile invece di progettare appositamente uno nuovo. L'utilizzo di circuiti standard è più semplice e meno soggetto a errori.

Il funzionamento (Figura 4.6) è il seguente. All'inizio di un ciclo di clock (il fronte di discesa del clock nella Figura 4.3) la parola contenuta nella memoria di controllo e puntata da MPC viene trasferita in MIR. Il tempo necessario per effettuare questa operazione è indicato nella figura con Δw . In termini di sottocicli, si può pensare che MIR venga caricato durante il primo sottociclo.

Subito dopo, i vari segnali si propagano all'interno del percorso dati. Grazie a questi segnali il contenuto di un registro viene inserito nel bus B e la ALU sa quale operazione deve eseguire. Seguono poi varie azioni, fino ad arrivare al secondo sottociclo. Dopo un intervallo di $\Delta w + \Delta x$ dall'inizio del ciclo, gli input della ALU diventano stabili.

Dopo un altro Δy , tutto nel circuito si è stabilizzato, compresi gli output della ALU, di N, di Z e dello shifter. I valori di N e Z vengono quindi salvati in una coppia di flip-flop. Questi bit, come tutti i registri caricati dal bus C, vengono salvati in corrispondenza del fronte di salita del clock, verso la fine del ciclo del percorso dati. L'output della ALU non viene memorizzato, ma semplicemente inserito nello shifter. Le attività della ALU e dello shifter si svolgono nel corso del sottociclo 3.

Dopo un ulteriore intervallo di tempo Δz , l'output dello shifter raggiunge i registri attraverso il bus C. I registri possono successivamente essere caricati verso la fine del

ciclo (nel fronte di salita dell'impulso del clock della Figura 4.3). Nel sottociclo 4 vengono caricati i registri e i flip-flop N e Z. Il sottociclo termina leggermente dopo il fronte di salita del clock, quando tutti i risultati sono stati salvati, i risultati delle precedenti operazioni di memoria sono disponibili e MPC è stato caricato. Questo processo continua ripetutamente finché qualcuno non si stufi e spenga la macchina.

Il microprogramma, oltre a guidare il percorso dati, deve parallelamente determinare quale sarà la microistruzione successiva, dato che non è necessario che esse vengano eseguite nello stesso ordine in cui appaiono nella memoria di controllo. Il calcolo dell'indirizzo della microistruzione successiva comincia dopo che MIR è stato caricato ed è diventato stabile. Inizialmente i 9 bit del campo NEXT_ADDRESS vengono copiati all'interno di MPC. Mentre si svolge questa copia viene ispezionato il campo JAM; se il suo valore è 000, allora non viene eseguita alcuna azione aggiuntiva. Quando termina la copia di NEXT_ADDRESS, MPC punta alla microistruzione successiva.

Se invece uno o più bit di JAM valgono 1, allora è necessario compiere delle azioni. Se JAMN è abilitato, se ne calcola l'OR logico con il flip-flop N (1 bit) e si memorizza il risultato nel bit più significativo di MPC. Analogamente, se è abilitato JAMZ, si calcola l'OR tra questo segnale e il flip-flop Z. Se sono abilitati entrambi, si calcola invece l'OR rispetto a tutti e due. La ragione per cui sono necessari i flip-flop N e Z è che, dopo il fronte di salita del clock (mentre il clock è ancora alto), il bus B non viene più alimentato e quindi gli output della ALU non possono più essere considerati corretti. Salvando in N e Z i flag di stato della ALU è possibile rendere stabili questi valori, per poterli utilizzare per calcolare correttamente MPC, indipendentemente dalle operazioni che la ALU compie nel frattempo.

Nella Figura 4.6 i componenti logici che effettuano questa elaborazione sono etichettati con la scritta "Bit alto". La funzione booleana che calcola questo bit è la seguente:

$$F = (\text{JAMZ AND } Z) \text{ OR } (\text{JAMN AND } N) \text{ OR } \text{NEXT_ADDRESS}[8]$$

Si noti che in tutti i casi possibili MPC può assumere soltanto uno di questi due valori:

1. il valore di NEXT_ADDRESS
2. il valore di NEXT_ADDRESS, in cui il bit più significativo è calcolato in OR con 1.

Non esistono altre possibilità. Se il bit più significativo di NEXT_ADDRESS valesse già 1, non avrebbe alcun senso utilizzare JAMN o JAMZ.

Si noti che quando tutti i bit di JAM valgono zero, l'indirizzo della successiva microistruzione da eseguire è semplicemente il numero a 9 bit contenuto nel campo NEXT_ADDRESS. In caso contrario, esistono due potenziali indirizzi per la microistruzione successiva: NEXT_ADDRESS e NEXT_ADDRESS calcolato in OR con 0x100 (assumendo che NEXT_ADDRESS ≤ 0xFF). (Si noti che 0x indica che il numero che lo segue è espresso in forma esadecimale.) Questo punto è illustrato nella Figura 4.7. La microistruzione corrente, che si trova nella locazione 0x75, ha il campo NEXT_ADDRESS = 0x92 e JAMZ impostato a 1. Di conseguenza l'indirizzo della microistruzione successiva dipende dal bit Z memorizzato durante la precedente operazione della ALU. Se il bit Z vale 0, la microistruzione successiva comincerà a partire dall'indirizzo 0x92; altrimenti, l'indirizzo sarà 0x192.

Il terzo bit del campo JAM è JMPC. Se è impostato a 1, gli 8 bit di MBR sono collegati in OR con gli 8 bit meno significativi del campo NEXT_ADDRESS della microistruzione precedente. Il risultato viene inviato a MPC. Il quadratino etichettato con "O" nella Figura 4.6 esegue l'OR di MBR e NEXT_ADDRESS quando JMPC vale 1, mentre lascia semplicemente passare NEXT_ADDRESS in MPC quando JMPC vale 0. In genere quando JMPC vale 1, gli 8 bit meno significativi di NEXT_ADDRESS valgono 0. Il bit più significativo può essere 0 oppure 1 e quindi il valore di NEXT_ADDRESS utilizzato con JMPC è generalmente 0x000 oppure 0x100. Il motivo per cui a volte si usa 0x000 mentre altre volte si usa 0x100 sarà spiegato in seguito.

Indirizzi	Indirizzo	JAM	Bit di controllo del percorso dati	Campo JAMZ impostato a 1
0x75	0x92	001		
			:	
0x92				
			:	
0x192			*	

Una di queste seguirà la microistruzione 0x75, a seconda del valore di Z

Figura 4.7 Le microistruzioni con JAMZ = 1 hanno due possibili successori.

La possibilità di calcolare l'OR di MBR e NEXT_ADDRESS e di memorizzare il risultato in MPC permette di implementare in modo efficiente una diramazione. Si noti che è possibile specificare uno qualsiasi dei 256 indirizzi unicamente sulla base dei bit presenti in MBR.

In un utilizzo tipico MBR contiene un codice operativo; in questo modo l'uso di JMPC fornisce, per ogni possibile codice operativo, un unico indirizzo in cui trovare la successiva microistruzione da eseguire. Questo metodo è utile per effettuare velocemente un salto verso la funzione associata al codice operativo appena prelevato.

Dato che la comprensione della temporizzazione della macchina è di fondamentale importanza per gli argomenti che seguiranno, vale la pena ripetere nuovamente il concetto. Questa volta lo faremo in termini di sottocicli, per permetterne una più facile visualizzazione; non dimentichiamoci però che gli unici reali eventi del clock sono il fronte di discesa, che fa partire il ciclo, e il fronte di salita, che carica i registri e i flip-flop N e Z. Si guardi per favore, ancora una volta, la Figura 4.3.

Durante il sottociclo 1, iniziato in corrispondenza del fronte di discesa del clock, la microistruzione all'indirizzo contenuto in quel momento in MPC viene caricato all'interno di MIR. Durante il sottociclo 2 i segnali si propagano da MIR verso l'esterno e il registro selezionato viene caricato sul bus B. Durante il sottociclo 3 la ALU e lo shifter svolgono le proprie operazioni e generano un risultato stabile. Durante il sottociclo 4 il bus C, i bus di memoria e i valori della ALU diventano stabili. In corrispondenza del

fronte di salita si verificano vari eventi: il contenuto del bus C viene caricato nei registri, i flip-flop N e Z vengono caricati, e MBR e MDR ottengono i loro risultati dall'operazione di memoria iniziata alla fine del precedente ciclo del percorso dati (se presente). Non appena MBR diventa stabile, si carica MPC in previsione della successiva microistruzione. MPC ottiene quindi il proprio valore verso la metà dell'intervallo, quando il clock è alto, ma dopo che MBR/MDR sono diventati disponibili. Il momento esatto in cui caricare MPC potrebbe essere determinato dal livello del segnale (invece che dal fronte) oppure da un ritardo temporale fisso rispetto al fronte di salita del clock. Ciò che veramente conta è che MPC non venga caricato finché non siano pronti i registri da cui dipende (MBR, N e Z). Non appena il clock scende MPC può indirizzare la memoria di controllo dando così inizio a un nuovo ciclo.

Si noti che ogni ciclo è autocontenuto; esso specifica che cosa andrà sul bus B, quali operazioni la ALU e lo shifter dovranno effettuare, dove verrà memorizzato il bus C e, infine, quale dovrà essere il successivo valore di MPC.

Vale la pena fare un'ultima osservazione riguardo alla Figura 4.6. Finora abbiamo considerato MPC come un normale registro, con una capacità di 9 bit, che viene caricato mentre il clock è alto. In realtà non c'è bisogno di utilizzare un registro: tutti i suoi input possono alimentare direttamente la memoria di controllo. È sufficiente che siano inviati alla memoria di controllo in corrispondenza del fronte di discesa del clock, quando MIR viene selezionato e letto; non c'è alcun bisogno di memorizzare effettivamente questi segnali all'interno di MPC. Per questa ragione MPC può essere implementato come un **registro virtuale**, cioè il luogo in cui raccogliere dei segnali e che assomiglia più a un pannello elettronico che a un vero e proprio registro. Il fatto di rendere MPC un registro virtuale semplifica la temporizzazione: in questo caso gli eventi si verificano solamente in corrispondenza dei fronti di salita e discesa del clock e in nessun altro momento. Se qualcuno ritiene che sia più facile interpretare MPC come un vero e proprio registro, può continuare a farlo, in quanto ciò non è altro che un diverso, ma comunque valido, punto di vista.

4.2 Esempio di ISA: IJVM

Continuiamo il nostro esempio introducendo il livello ISA della macchina che sarà interpretato dal microprogramma eseguito nella microarchitettura della Figura 4.6 (IJVM). Per praticità a volte ci riferiremo all'architettura dell'insieme d'istruzioni (ISA) con il termine di **macroarchitettura**, in contrapposizione con la microarchitettura. Prima di descrivere IJVM faremo una breve digressione che ci permetterà di mettere in luce le motivazioni.

4.2.1 Stack

Praticamente tutti i linguaggi di programmazione supportano il concetto di procedure (metodi), dotate di un insieme di variabili locali. È possibile accedere a queste variabili dall'interno della procedura, ma ciò diventa impossibile una volta che la procedura termina. La domanda che sorge è: "In quale parte della memoria bisogna memorizzare queste variabili?".

La soluzione più semplice, che consiste nell'assegnare a ciascuna variabile un indirizzo assoluto della memoria, non funziona. Il problema nasce dal fatto che una procedura potrebbe richiamare se stessa; nel corso del Capitolo 5 studieremo le procedure che si comportano in questo modo e che sono dette ricorsive. Per il momento è sufficiente dire che se una procedura è attiva (cioè è stata invocata) due volte, è impossibile memorizzare le sue variabili in locazioni assolute della memoria in quanto la seconda invocazione interferirebbe con la prima.

Si usa quindi un'altra strategia. Per memorizzare le variabili viene riservata un'area della memoria, chiamata **stack** ("pila"), al cui interno però non si stabiliscono indirizzi assoluti per le singole variabili. Si imposta invece un registro, chiamiamolo LV, in modo che punti alla base delle variabili locali per la procedura corrente. Nella Figura 4.8(a) è stata richiamata una procedura A, che possiede tre variabili locali, a1, a2 e a3; per memorizzarle è stato quindi riservato uno spazio di memoria a partire dalla locazione puntata da LV. Un altro registro, SP, punta alla parola che si trova nella locazione più alta all'interno dello stack delle variabili locali di A. Se LV è 100 e le parole sono di 4 byte, il valore di SP sarà 108. Per far riferimento alle variabili locali si fornisce il loro spiazzamento (*offset*) rispetto a LV. La struttura dati compresa tra LV e SP (considerando anche le parole puntate dai due registri) è chiamata **blocco delle variabili locali** di A.

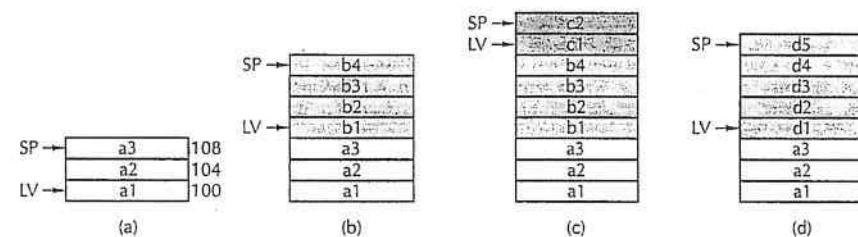


Figura 4.8 Utilizzo dello stack per memorizzare le variabili locali. (a) Mentre A è attiva. (b) Dopo che A ha richiamato B. (c) Dopo che B ha richiamato C. (d) Dopo che C e B hanno restituito il controllo e A ha richiamato D.

Consideriamo ora che cosa succede se A richiama un'altra procedura, diciamo B. Dove dovrebbero essere memorizzate le quattro variabili locali (b1, b2, b3, b4) di B? Risposta: nello stack sopra il blocco di A, come mostra la Figura 4.8(b). Si noti che LV è stato modificato dalla chiamata alla procedura in modo da puntare alle variabili locali di B invece che a quelle di A. È possibile far riferimento alle variabili locali di B fornendo il loro spiazzamento rispetto a LV. Analogamente, se B richiama C, LV e SP vengono nuovamente modificati in modo da allocare lo spazio necessario per memorizzare le due variabili di C; questa situazione è mostrata nella Figura 4.8(c).

Quando C termina, B diventa nuovamente attiva e lo stack viene modificato per tornare allo stato mostrato nella Figura 4.8(b), in modo che LV torni a puntare alle variabili locali di B. Analogamente, quando B restituisce il controllo, si torna nuovamente

alla situazione mostrata nella Figura 4.8(a). In tutte le situazioni LV punta alla base del blocco dello stack corrispondente alla procedura correntemente attiva, mentre SP punta alla cima dello stesso blocco.

Supponendo ora che *A* richiami *D*, che ha cinque variabili locali, si ottiene la situazione mostrata nella Figura 4.8(d), che illustra come le variabili locali di *D* usino la stessa area di memoria utilizzata precedentemente da *B*, oltre a una parte di quella che era stata utilizzata da *C*. Organizzando la memoria in questo modo, è possibile allocare solamente la memoria necessaria alle procedure attive. Quando una procedura restituisce il controllo viene rilasciata la memoria utilizzata dalle sue variabili locali.

Oltre a memorizzare le variabili locali gli stack hanno anche un altro utilizzo. Possono essere utilizzati per memorizzare gli operandi durante il calcolo di un'espressione aritmetica. Quando uno stack è utilizzato in questo modo ci si riferisce a esso con il termine di **stack degli operandi**. Supponiamo per esempio che prima di richiamare *B*, *A* debba eseguire il calcolo

$$a1 = a2 + a3;$$

Un modo per effettuare questa somma consiste nel porre *a2* in cima allo stack, come mostra la Figura 4.9(a). Qui SP è stato incrementato del numero di byte che formano una parola, diciamo 4, in modo da puntare all'indirizzo in cui è stato memorizzato il primo operando. In seguito *a3* viene posto in cima allo stack, come mostra la Figura 4.9(b). Aprendo una parentesi riguardo alla notazione utilizzata nel testo, specifichiamo che tutti i frammenti di programmi saranno scritti utilizzando il carattere Courier, com'è già stato fatto in precedenza. Utilizzeremo questo carattere anche per i codici operativi del linguaggio assemblativo e per i registri della macchina, mentre nel testo scriveremo in *corsivo* i nomi delle variabili e delle procedure del programma. La differenza risiede nel fatto che le variabili e i nomi delle procedure sono scelti dall'utente, mentre i codici operativi e i nomi dei registri sono predefiniti.

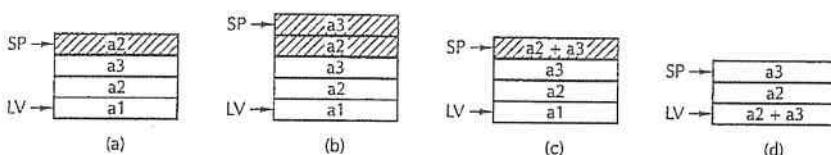


Figura 4.9 Utilizzo dello stack degli operandi per il calcolo di un'espressione aritmetica.

Il calcolo effettivo può a questo punto essere realizzato eseguendo un'istruzione che preleva due parole dallo stack, le somma e inserisce il risultato nuovamente nello stack, come mostra la Figura 4.9(c). Infine, la parola che si trova in cima allo stack può essere rimossa e memorizzata nella variabile locale *a1*, com'è illustrato nella Figura 4.9(d).

Il blocco delle variabili locali e lo stack degli operandi possono essere mischiati tra loro. Per esempio quando si calcola un'espressione come $x^2 + f(x)$, una parte dell'e-

spressione (per esempio, x^2) può trovarsi nello stack degli operandi nel momento in cui viene invocata la funzione *f*. Il risultato della funzione viene lasciato nello stack, al di sopra di x^2 , in modo che l'istruzione successiva li possa sommare.

Vale la pena precisare che mentre tutte le macchine utilizzano uno stack per memorizzare le variabili locali, non tutte usano uno stack degli operandi per eseguire le operazioni aritmetiche. In realtà la maggior parte non lo utilizza; JVM e IJVM funzionano però in questo modo e per questo motivo abbiamo deciso di introdurlo.

4.2.2 Modello della memoria di IJVM

Siamo ora pronti ad analizzare l'architettura di IJVM. Fondamentalmente essa consiste in una memoria concepibile in due modi distinti: come un array di 4.294.967.296 byte (4 GB) oppure come un array di 1.073.741.824 parole da 4 byte. Diversamente dalla maggior parte degli ISA, la Java Virtual Machine non rende direttamente visibili a livello ISA gli indirizzi di memoria assoluti, ma utilizza degli indirizzi impliciti che forniscono la base per l'uso di puntatori. L'unico modo che le istruzioni IJVM hanno per accedere alla memoria è quello di indicizzarla utilizzando questi puntatori. In ogni momento sono definite le seguenti aree di memoria.

- Porzione costante di memoria.* I programmi IJVM non possono scrivere in quest'area che contiene costanti, stringhe e puntatori ad altre aree di memoria cui è possibile far riferimento. È caricata quando il programma è portato in memoria e in seguito non viene modificata. Esiste un registro隐式的, CPP, contenente l'indirizzo della prima parola della porzione costante di memoria.
- Blocco delle variabili locali.* Per ogni invocazione di un metodo viene allocata un'area in cui memorizzare le variabili locali durante l'intero ciclo di vita dell'invocazione. Quest'area è chiamata **blocco delle variabili locali**. Nella parte iniziale di questo blocco sono memorizzati i parametri (chiamati anche argomenti) con cui è stato invocato il metodo. Il blocco delle variabili locali non comprende lo stack degli operandi, che è separato. Tuttavia, per motivi di efficienza, qui abbiamo scelto di implementare lo stack degli operandi immediatamente sopra il blocco delle variabili locali. È presente un registro隐式的, LV, contenente l'indirizzo della prima localizzazione all'interno del blocco delle variabili locali. I parametri passati durante l'invocazione del metodo sono memorizzati all'inizio del blocco delle variabili locali.
- Stack degli operandi.* Il blocco dello stack non può superare una certa dimensione, stabilita in anticipo dal compilatore Java. Lo spazio per lo stack degli operandi è allocato direttamente sopra il blocco delle variabili locali (Figura 4.10). Nella nostra implementazione conviene pensare che lo stack degli operandi faccia parte del blocco delle variabili locali. In ogni caso c'è un registro隐式的, CPP e LV, contenente l'indirizzo della parola in cima allo stack. Si noti che, diversamente da CPP e LV, questo puntatore, SP, cambia durante l'esecuzione del metodo a seconda che gli operandi siano inseriti nello stack o rimossi da quest'ultimo.
- Area dei metodi.* Infine c'è una regione di memoria in cui risiede il programma, simile all'area "testo" di un processo UNIX. È presente un registro隐式的 che contiene l'indirizzo della successiva istruzione da prelevare. Questo puntatore è

chiamato contatore di programma, oppure PC. Diversamente dalle altre regioni di memoria l'area dei metodi è trattata come un array di byte.

Per quanto riguarda i puntatori occorre precisare un aspetto. I registri CPP, LV e SP sono tutti puntatori a *parole*, e non a *byte*; anche i loro spiazzamenti sono espressi come numero di parole. Per il sottoinsieme d'istruzioni su interi che abbiamo scelto, tutti i riferimenti a elementi che si trovano nella porzione costante di memoria, nel blocco delle variabili locali e nello stack sono definiti come parole; allo stesso modo tutti gli spiazzamenti utilizzati all'interno di questi blocchi sono definiti in termini di parole. Per esempio LV, LV + 1 e LV + 2 fanno riferimento alle prime tre parole del blocco delle variabili locali. LV, LV + 4 e LV + 8 fanno invece riferimento a parole che si trovano a intervalli di quattro parole (16 byte) l'una dall'altra.

Al contrario, PC contiene un indirizzo espresso in byte; un'addizione o una sottrazione effettuata su PC modifica quindi l'indirizzo in base a un certo numero di byte, e non di parole. L'indirizzamento di PC è diverso da quello degli altri registri; la speciale porta di memoria fornita per PC su Mic-1 ne è una dimostrazione. Come abbiamo già visto questa porta è larga soltanto 1 byte. Incrementare PC di uno e dare inizio a una lettura corrisponde a prelevare il *byte* successivo; incrementare SP di uno e dare inizio a una lettura corrisponde invece a prelevare la *parola* successiva.

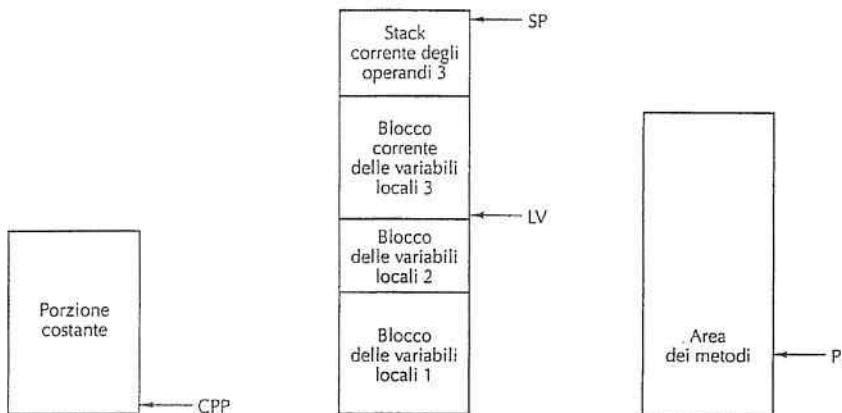


Figura 4.10 Parti della memoria di IJVM.

4.2.3 Insieme d'istruzioni IJVM

La Figura 4.11 mostra l'insieme d'istruzioni IJVM. Ogni istruzione è composta da un codice operativo e in alcuni casi da un operando, che può essere uno spiazzamento o una costante. La prima colonna mostra la codifica esadecimale dell'istruzione, la seconda il nome mnemonico in linguaggio assemblativo e la terza una breve descrizione del significato dell'istruzione.

Esa	Nome mnemonico	Significato
0x10	BIPUSH byte	Scrive un byte in cima allo stack
0x59	DUP	Legge la parola in cima allo stack e la inserisce in cima allo stack
0xA7	GOTO offset	Diramazione incondizionata
0x60	IADD	Sostituisce le due parole in cima allo stack con la loro somma
0x7E	IAND	Sostituisce le due parole in cima allo stack con il loro AND
0x99	IFEQ offset	Estrae una parola dalla cima dello stack ed effettua una diramazione se ha valore zero
0x9B	IFLT offset	Estrae una parola dalla cima dello stack ed effettua una diramazione se ha valore negativo
0x9F	IF_ICMPEQ offset	Estrae le due parole in cima allo stack ed effettua una diramazione se sono uguali
0x84	IINC varnum const	Aggiunge una costante a una variabile locale
0x15	ILOAD varnum	Scrive una variabile locale in cima allo stack
0xB6	INVOKEVIRTUAL disp	Invoca un metodo
0x80	IOR	Sostituisce le due parole in cima allo stack con il loro OR
0xAC	IRETURN	Termina un metodo restituendo un valore intero
0x36	ISTORE varnum	Preleva una parola dalla cima dello stack e la memorizza in una variabile locale
0x64	ISUB	Sostituisce le due parole in cima allo stack con la loro differenza
0x13	LDC_W index	Scrive in cima allo stack una costante proveniente dalla porzione costante di memoria
0x00	NOP	Non esegue nulla
0x57	POP	Rimuove la cima allo stack
0x5F	SWAP	Scambia le due parole in cima allo stack
0xC4	WIDE	Istruzione prefisso: l'istruzione successiva ha un indice a 16 bit

Figura 4.11 Insieme d'istruzioni di IJVM. Gli operandi byte, const e varnum sono lunghi 1 byte. Gli operandi disp, index e offset sono lunghi 2 byte.

Alcune istruzioni permettono di inserire nello stack una parola proveniente da varie fonti, come per esempio la porzione costante di memoria (LDC_W), il blocco delle variabili locali (ILOAD) e l'istruzione stessa (BIPUSH). Una variabile può anche essere estratta dallo stack e memorizzata nel blocco delle variabili locali (ISTORE). È possibile eseguire due operazioni aritmetiche (IADD e ISUB) e due operazioni logiche, cioè booleane, (IAND e IOR) utilizzando come operandi le due parole che si trovano in cima allo stack. In tutte le operazioni logiche e aritmetiche vengono estratte due parole dallo stack e il risultato viene inserito sopra di esso. Sonò fornite quattro istruzioni per i salti, una non

condizionale (GOTO) e tre condizionali (IFEQ, IFLT e IF_ICMPEQ). Tutte queste istruzioni, se accettate, modificano il valore di PC in base alla grandezza del loro spiazzamento (16 bit con segno), che si trova nell'istruzione, subito dopo il codice operativo. Questo spiazzamento viene aggiunto all'indirizzo dell'istruzione. Ci sono anche istruzioni IJVM che permettono di scambiare le due parole in cima allo stack (SWAP), di duplicare la parola che si trova in cima (DUP) e di rimuoverla (POP).

Alcune istruzioni hanno più formati, per permettere di utilizzare una forma più breve per le versioni utilizzate più frequentemente. In IJVM abbiamo incluso due dei vari meccanismi che JVM mette a disposizione a questo scopo. In un caso abbiamo omesso la versione corta in favore di quella più generale. In un altro caso mostriamo come si può utilizzare l'istruzione prefisso WIDE per modificare l'istruzione successiva.

Infine c'è un'istruzione (INVOKEVIRTUAL) per invocare un altro metodo e un'istruzione (IRETURN) per terminare il metodo e restituire il controllo a quello che l'aveva invocato. A causa della complessità del meccanismo abbiamo semplificato leggermente la definizione, rendendo possibile un semplice metodo per invocare una chiamata e restituire il controllo. Questa restrizione, a differenza di quanto avviene realmente in Java, ci permette solamente di invocare un metodo che esiste all'interno dello stesso oggetto del metodo chiamante. Tale restrizione invalida severamente l'orientamento a oggetti, ma permette di non dover localizzare dinamicamente il metodo. (Se non si ha familiarità con la programmazione orientata agli oggetti si può ignorare senza alcun problema quest'ultima osservazione. Quello che abbiamo fatto è stato trasformare Java in un linguaggio non orientato agli oggetti, come C o Pascal.) Su tutti i calcolatori, fatta eccezione per la JVM, l'indirizzo della procedura da richiamare è determinato direttamente dall'istruzione CALL; il nostro approccio è quindi il caso normale e non rappresenta un'eccezione.

Il meccanismo per l'invocazione dei metodi funziona nel modo seguente. Prima di tutto il chiamante inserisce sullo stack un riferimento (puntatore) all'oggetto da chiamare. (Questo riferimento non è necessario in IJVM dato che non può essere specificato alcun altro oggetto, ma lo abbiamo mantenuto per coerenza con JVM.) Nella Figura 4.12(a) questo riferimento è indicato con OBJREF. Successivamente il chiamante inserisce sullo stack i parametri del metodo che, in questo esempio, sono *Parametro 1*, *Parametro 2* e *Parametro 3*. Infine viene eseguita l'istruzione INVOKEVIRTUAL.

Quest'istruzione include uno spiazzamento che indica una posizione all'interno della porzione costante di memoria; questa locazione contiene l'indirizzo dell'area dei metodi in cui inizia il metodo che si sta invocando. Tuttavia, mentre il codice del metodo risiede nella locazione puntata da questo puntatore, i primi 4 byte nell'area dei metodi contengono dati speciali. I primi 2 byte sono interpretati come un intero a 16 bit che indica il numero di parametri del metodo (i parametri stessi sono stati precedentemente inseriti in cima allo stack). In questo conteggio OBJREF viene considerato come un parametro: il parametro 0. Questo intero a 16 bit fornisce, insieme al valore di SP, la locazione di OBJREF. Si noti che LV punta a OBJREF invece che al primo, vero, parametro; questa scelta è sostanzialmente arbitraria.

Anche i successivi 2 byte dell'area dei metodi sono interpretati come un intero a 16 bit, che però indica la dimensione del blocco delle variabili locali per il metodo invocato.

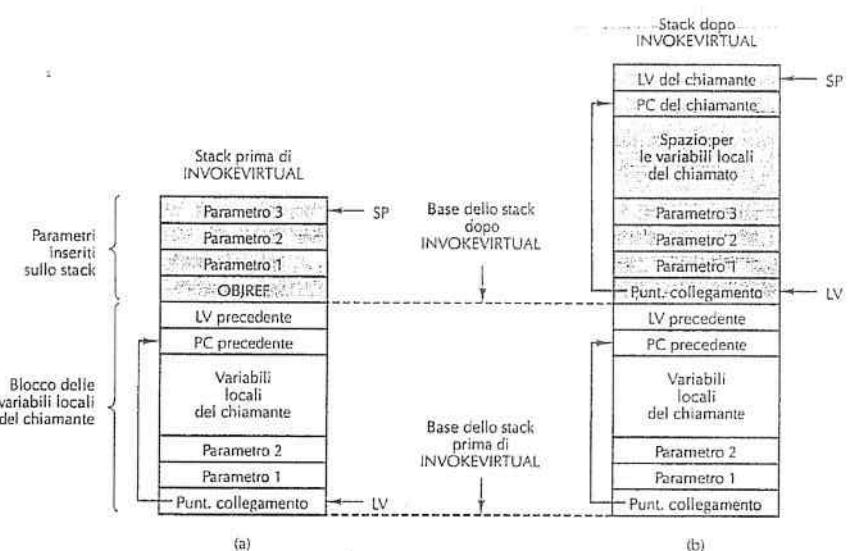


Figura 4.12 (a) Memoria prima di eseguire l'istruzione INVOKEVIRTUAL. (b) Dopo averla eseguita.

Questo valore è necessario, dato che verrà stabilito un nuovo stack per il metodo, immediatamente sopra il blocco delle variabili locali. Infine il quinto byte dell'area dei metodi contiene il primo codice operativo da eseguire.

Descriviamo ora l'effettiva sequenza di operazioni (Figura 4.12) che si verificano quando viene eseguita l'istruzione INVOKEVIRTUAL. I due byte senza segno che seguono il codice operativo sono utilizzati per costruire un indice relativo alla porzione costante di memoria (il primo byte è quello più significativo). L'istruzione calcola l'indirizzo base del nuovo blocco delle variabili locali sottraendo il numero di parametri dal puntatore allo stack e impostando LV in modo che punti a OBJREF. In questa locazione, sovrascrivendo OBJREF, viene memorizzato l'indirizzo della locazione in cui si trova il vecchio PC. Questo indirizzo è calcolato sommando la dimensione del blocco delle variabili locali (parametri + variabili locali) all'indirizzo contenuto in LV. Immediatamente sopra l'indirizzo in cui memorizzare il vecchio PC c'è l'indirizzo in cui deve essere memorizzato il vecchio LV. Al di sopra di questo indirizzo inizia lo stack per la procedura che è stata appena richiamata. SP viene impostato in modo da puntare al vecchio LV, che è l'indirizzo immediatamente sotto la prima locazione vuota dello stack. Ricordiamoci che SP punta sempre alla parola in cima allo stack. Se lo stack è vuoto, allora punta alla prima locazione sotto la fine dello stack, dato che i nostri stack crescono verso l'alto, verso gli indirizzi più grandi. Anche nelle nostre figure gli stack crescono verso l'alto, con gli indirizzi che aumentano in direzione del bordo superiore della pagina.

L'ultima operazione richiesta per terminare l'esecuzione di INVOKEVIRTUAL è impostare PC in modo che punti al quinto byte nell'area del codice del metodo.

L'istruzione IRETURN inverte la sequenza delle operazioni compiute da INVOKEVIRTUAL, come mostra la Figura 4.13. Essa dealloca lo spazio utilizzato dal metodo che sta restituendo il controllo e riporta lo stack nel suo precedente stato, tranne per il fatto che (1) la parola OBJREF (ora sovrascritta) e tutti i parametri sono stati estratti dallo stack e che (2) il valore restituito dal metodo è stato inserito in cima allo stack, nella locazione occupata precedentemente da OBJREF. Per memorizzare il vecchio stato l'istruzione IRETURN deve essere in grado di riportare i puntatori PC e LV ai loro precedenti valori. Per far ciò accede al puntatore di collegamento (*link pointer*), cioè alla parola identificata dal puntatore LV corrente. Non dimentichiamo che in questa locazione, in cui era precedentemente memorizzato OBJREF, l'istruzione INVOKEVIRTUAL ha memorizzato l'indirizzo contenente il vecchio PC. Questa parola, e quella che si trova al di sopra, vengono recuperate e utilizzate per restituire a PC e a LV i loro precedenti valori. Il valore fornito dal metodo, memorizzato in cima allo stack, viene copiato nella locazione in cui era originariamente memorizzato OBJREF, e SP viene reimpostato in modo da puntare a questa locazione. Il controllo viene quindi restituito all'istruzione immediatamente successiva rispetto all'istruzione INVOKEVIRTUAL.

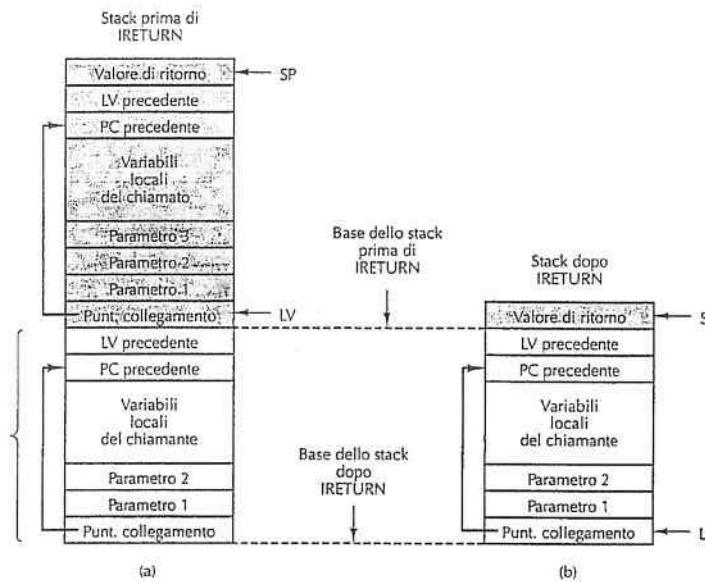


Figura 4.13 (a) Memoria prima di eseguire l'istruzione IRETURN. (b) Dopo averla eseguita.

Finora nella nostra macchina non abbiamo introdotto istruzioni di input/output, e non lo faremo neanche in seguito. Queste istruzioni non sono necessarie, a meno che non le richieda la Java Virtual Machine; tuttavia le specifiche ufficiali di JVM non fanno mai

cenno alle istruzioni di I/O. La teoria è che una macchina che non esegue alcun input o output è "sicura". (In JVM lettura e scrittura sono eseguite per mezzo di chiamate a speciali metodi di I/O.)

4.2.4 Compilazione da Java a IJVM

Vediamo ora le relazioni tra Java e IJVM. Un compilatore Java che riceve in ingresso il semplice frammento di codice Java della Figura 4.14(a) dovrebbe generare qualcosa di simile al codice assemblativo della Figura 4.14(b). I commenti (preceduti dai caratteri //) e i numeri di linea alla sinistra del codice assemblativo non fanno parte dell'output del compilatore. L'assemblatore Java dovrebbe in seguito tradurre il programma assemblativo nel codice binario mostrato nella Figura 4.14(c). (In realtà il compilatore Java crea il proprio programma assemblativo e genera il codice binario direttamente.) In questo esempio abbiamo assunto che *i* sia la variabile locale 1, *j* la variabile locale 2 e *k* la variabile locale 3.

<pre> i = j + k; if (i == 3) k = 0; else j = j - 1; </pre>	<pre> 1 ILOAD i // i = j + k 0x15 0x02 2 ILOAD k 0x15 0x03 3 IADD 0x60 4 ISTORE i 0x36 0x01 5 ILOAD i 0x15 0x01 6 BIPUSH 3 0x10 0x03 7 IF_ICMPEQ L1 0x9F 0x00 0x0D 8 ILOAD j 0x15 0x02 9 BIPUSH 1 0x10 0x01 10 ISUB 0x64 11 ISTORE j 0x36 0x02 12 GOTO L2 0xA7 0x00 0x07 13 L1: BIPUSH 0 0x10 0x00 14 ISTORE k 0x36 0x03 15 L2: </pre>	<pre> 0x15 0x02 0x15 0x03 0x60 0x36 0x01 0x15 0x01 0x10 0x03 0x9F 0x00 0x0D 0x15 0x02 0x10 0x01 0x64 0x36 0x02 0xA7 0x00 0x07 0x10 0x00 0x36 0x03 </pre>
(a)	(b)	(c)

Figura 4.14 (a) Frammento di programma Java. (b) Il frammento in linguaggio assemblativo Java. (c) Programma IJVM in esadecimale.

Il codice compilato è semplice. Inizialmente *j* e *k* vengono inseriti in cima allo stack, vengono sommati, e il risultato viene memorizzato in *i*. Successivamente *i* e la costante 3 vengono messi nello stack e confrontati. Se sono uguali, si effettua una diramazione che porta in *L1*, dove *k* è impostato al valore 0. Se invece sono diversi il test dà esito negativo e viene eseguito il codice che segue l'istruzione IF_ICMPEQ. Dopo alcune istruzioni si effettua una diramazione che porta in *L2*, cioè nel punto in cui i rami then e else si riuniscono.

La Figura 4.15 mostra come varia lo stack degli operandi durante l'esecuzione del programma IJVM della Figura 4.14(b). Prima che cominci l'esecuzione del codice lo stack è vuoto, com'è indicato dalla linea orizzontale sopra il numero 0. Dopo la prima istruzione ILOAD, *j* si trova sullo stack, com'è indicato dal rettangolo *j* sopra il numero

1 (questo valore significa che è stata eseguita un'istruzione). Dopo la seconda istruzione ILOAD, nello stack ci sono due parole, come mostrano i due rettangoli sopra il numero 2. Dopo l'istruzione IADD, nello stack c'è soltanto una parola, che contiene la somma $j + k$. Quando la parola in cima allo stack viene estratta e memorizzata in i lo stack è vuoto, come mostra la linea sopra il numero 4.

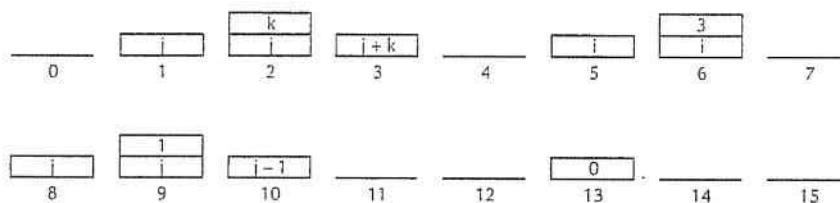


Figura 4.15 Stack dopo ogni istruzione della Figura 4.14(b).

L'istruzione 5 (ILOAD) inizia il costrutto if inserendo i nello stack (in 5). Successivamente (in 6) arriva la costante 3. Dopo il confronto lo stack è nuovamente vuoto (7). L'istruzione 8 rappresenta l'inizio del ramo else del frammento di programma Java. La parte else continua fino all'istruzione 12, punto in cui c'è la diramazione che porta a L2 e che permette di saltare il codice relativo al ramo then.

4.3 Implementazione di esempio

Dopo aver specificato nel dettaglio la microarchitettura e la macroarchitettura, rimane il problema dell'implementazione. In altre parole, che aspetto ha e come funziona un programma che viene eseguito sulla prima e che interpreta la seconda? Prima di poter rispondere a queste domande dobbiamo considerare in modo preciso la notazione che utilizzeremo per descrivere l'implementazione.

4.3.1 Microistruzioni e notazione

In teoria potremmo descrivere la memoria di controllo in binario, 36 bit per parola. Ma, come avviene normalmente nei linguaggi di programmazione, è molto vantaggioso introdurre una notazione che esprime l'essenza dei problemi da trattare mascherando allo stesso tempo i dettagli che possono essere ignorati o che possono essere meglio gestiti in maniera automatica. È importante capire che il linguaggio da noi scelto ha lo scopo di illustrare i concetti e non di facilitare la progettazione di un'architettura efficiente. Se questo fosse il nostro obiettivo dovremmo utilizzare una differente notazione, in grado di fornire al progettista la massima flessibilità possibile. Un aspetto in cui questo problema è rilevante riguarda la scelta degli indirizzi.

Dato che la memoria non è ordinata dal punto di vista logico, non esiste alcuna naturale "istruzione successiva" da poter impiegare quando specifichiamo una sequenza di

operazioni. Molta della potenza dell'organizzazione del controllo dipende dalla capacità del progettista (o dell'assemblatore) di specificare gli indirizzi in modo efficiente. Cominciamo dunque con l'introdurre un semplice linguaggio simbolico che descrive in modo completo le operazioni senza spiegare in modo esaustivo come sono stati determinati tutti gli indirizzi.

La nostra notazione specifica su una singola linea tutte le attività che si svolgono durante un unico ciclo di clock. In teoria per descrivere le operazioni potremmo utilizzare un linguaggio ad alto livello. Tuttavia un controllo ciclo-dopo-ciclo è molto importante, in quanto dà la possibilità di effettuare più operazioni concorrenti; inoltre è necessario per poter analizzare i cicli in modo da comprenderne lo svolgimento e verificare le operazioni. Se l'obiettivo è quello di definire un'implementazione veloce ed efficiente (a parità di altri fattori, veloce ed efficiente è sempre meglio che lento e inefficiente) ogni ciclo ha la sua importanza. Le implementazioni reali celano molte sottili astuzie, come l'uso di sequenze o operazioni difficilmente comprensibili per cercare di risparmiare anche un solo ciclo. Risparmiare cicli ha un importante impatto sulle prestazioni: se un'istruzione a quattro cicli può essere ridotta a due cicli, allora verrà eseguita al doppio della velocità.

Un possibile approccio per definire la notazione si limita all'elenco dei segnali che dovrebbero essere attivati durante ciascun ciclo di clock. Supponiamo che in un ciclo vogliamo incrementare il valore di SP, oltre a dare inizio a un'operazione di lettura; desideriamo inoltre che l'istruzione successiva sia quella che risiede nella locazione 122 della memoria di controllo. Potremmo scrivere

ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122

dove WSP significa "scrivere il registro SP". Questa notazione è completa, ma di difficile lettura. Al suo posto preferiamo combinare le operazioni in un modo naturale e intuitivo in modo da cogliere la sostanza di quello che sta avvenendo.

SP = SP + 1; rd

Chiamiamo il nostro linguaggio ad alto livello Micro Assembly Language, "MAL" (iniziale di "malato", quello che uno diventa se è obbligato a scrivere molte linee di codice in questo linguaggio). MAL è definito appositamente per rispecchiare le caratteristiche della microarchitettura. Durante ogni ciclo è possibile scrivere qualsiasi registro, anche se in genere ne viene scritto soltanto uno. Soltanto un registro può essere collegato al lato B della ALU, mentre sul lato A le scelte possibili sono +1, 0, -1 e il registro H. Possiamo inoltre usare un semplice costrutto di assegnamento, come in Java, per indicare l'operazione da eseguire; per copiare un dato da SP a MDR possiamo per esempio dire

MDR = SP

Per indicare che si intende utilizzare funzioni della ALU diverse dal semplice passaggio verso il bus B possiamo scrivere

MDR = H + SP

che somma il contenuto del registro H a SP e scrive il risultato in MDR. L'operatore + è commutativo (il che significa che l'ordine degli operandi è ininfluente); quindi l'istruzione può essere riscritta come

$$MDR = SP + H$$

e genera la stessa microistruzione a 36 bit; per essere precisi H deve però essere l'operando sinistro della ALU.

Dobbiamo fare attenzione a usare soltanto le operazioni lecite. Quelle più importanti sono mostrate nella Figura 4.16, in cui SOURCE può essere MDR, PC, MBR, MBRU, SP, LV, CPP, TOS oppure OPC (MBRU indica la versione senza segno di MBR). Tutti questi registri possono fungere da sorgenti per la ALU sul bus B. In modo analogo DEST può essere uno qualsiasi fra MAR, MDR, PC, SP, LV, CPP, TOS, OPC e H; tutti questi registri possono essere la destinazione dell'output della ALU che viene inviato sul bus C. Questo formato è insidioso, in quanto molti costrutti apparentemente corretti sono invece illegali. Per esempio, l'assegnamento

$$MDR = SP + MDR$$

sembra perfettamente ragionevole, ma non può in alcun modo essere eseguito in un unico ciclo sul percorso dati della Figura 4.6. Questa restrizione è dovuta al fatto che in un'addizione (fatta eccezione per un incremento o un decremento) uno degli operandi deve essere il registro H.

DEST = H
DEST = SOURCE
DEST = H
DEST = SOURCE
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE - H
DEST = SOURCE - 1
DEST = -H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = -1

Figura 4.16 Operazioni consentite. Possono tutte essere estese aggiungendo "<< 8" per traslare il risultato a sinistra di 1 byte. Un'operazione comune è H = MBR << 8.

Analogamente, l'assegnamento

$$H = H - MDR$$

potrebbe essere utile, ma è anch'esso ineseguibile dato che l'unica possibile sorgente per il sottraendo è il registro H. È compito dell'assemblatore rifiutare quelle istruzioni che sembrano valide, ma in realtà non lo sono.

Estendiamo ora la notazione per permettere assegnamenti multipli mediante la ripetizione del simbolo di uguaglianza. Per esempio, per sommare 1 a SP, memorizzare il risultato nuovamente in SP e scriverlo anche all'interno di MDR, si può utilizzare la seguente notazione

$$SP = MDR = SP + 1$$

Per indicare letture e scritture di parole di dati da 4 byte aggiungeremo semplicemente rd e wr nella microistruzione. Il prelievo di un byte attraverso la porta a 1 byte si indica con fetch. Gli assegnamenti e le operazioni della memoria possono svolgersi durante lo stesso ciclo, il che è indicato scrivendoli sulla stessa linea.

Per evitare possibili confusioni ripetiamo che Mic-1 ha due modi per accedere alla memoria. Le letture e le scritture di parole a 4 byte usano MAR/MDR e nelle microistruzioni sono indicate rispettivamente da rd e wr. Le letture dei codici operativi a 1 byte dal flusso dell'istruzione utilizzano PC/MBR e sono indicate nelle microistruzioni mediante la parola fetch. Le due operazioni possono procedere in modo simultaneo.

Tuttavia lo stesso registro non può ricevere un valore dalla memoria e dal percorso dati durante lo stesso ciclo. Consideriamo il codice

$$\begin{aligned} MAR &= SP; \text{rd} \\ MDR &= H \end{aligned}$$

La prima microistruzione assegna alla fine della seconda microistruzione un valore della memoria a MDR. Tuttavia anche la seconda microistruzione assegna nello stesso momento un valore a MDR. Questi due assegnamenti sono in conflitto e non sono permessi, poiché in tal caso il risultato sarebbe indefinito.

Ricordiamoci che ogni microistruzione deve fornire in modo esplicito l'indirizzo della successiva microistruzione da eseguire. Tuttavia spesso succede che una microistruzione sia invocata solamente da un'altra microistruzione, in particolare da quella che si trova nella linea immediatamente sopra di essa. Per facilitare il lavoro del programmatore normalmente il microassemblatore assegna un indirizzo a ciascuna microistruzione (non necessariamente consecutiva all'interno della memoria di controllo) e completa il campo NEXT_ADDRESS in modo che le microistruzioni scritte su linee consecutive vengano eseguite in sequenza.

Tuttavia in alcuni casi il micropogrammatore desidera inserire una diramazione, condizionale oppure no. La notazione per i salti incondizionati è semplice:

goto label

e può essere inclusa in ogni microistruzione per indicare in modo esplicito il proprio successore. Per esempio, la maggior parte delle sequenze di microistruzioni termina con

un ritorno alla prima istruzione del ciclo principale. Dunque l'ultima istruzione di queste sequenze include generalmente

```
goto Main1
```

Si noti che anche quando una microistruzione contiene `goto` il percorso dati è disponibile per le normali operazioni. Dopo tutto ogni microistruzione contiene un campo `NEXT_ADDRESS` e quello che fa `goto` non è altro che comunicare al microassemblatore di inserire in questo campo un particolare valore, al posto dell'indirizzo in cui esso aveva deciso di mettere la microistruzione della linea successiva. In linea di principio ogni linea dovrebbe avere un'istruzione `goto` che però, per comodità, viene omessa quando l'indirizzo di destinazione è la linea successiva.

Per i salti condizionali abbiamo invece bisogno di una notazione differente. Ricordiamoci che `JAMN` e `JAMZ` usano i bit `N` e `Z`, che vengono impostati in base all'output della ALU. A volte è necessario testare un registro per vedere se il suo valore è 0. Un modo per farlo consiste nel far passare il valore del registro attraverso la ALU e nel memorizzare il risultato nuovamente nel registro stesso. Scrivere

```
TOS = TOS
```

può sembrare strano, ma svolge correttamente il compito (impostando il flip-flop `Z` in base a `TOS`). Tuttavia per migliorare l'aspetto dei microprogrammi estendiamo `MAL`, aggiungendo due nuovi registri immaginari, `N` e `Z`, a cui è possibile effettuare degli assegnamenti. Per esempio,

```
Z = TOS
```

fa passare `TOS` attraverso la ALU, impostando quindi i flip-flop `Z` (e `N`), senza però effettuare alcuna memorizzazione nei registri. Utilizzare `Z` oppure `N` come destinazione non fa altro che indicare al microassemblatore di impostare a 0 tutti i bit del campo `C` della Figura 4.5. Il percorso dati esegue un ciclo in cui sono consentite tutte le normali operazioni, ma in cui non viene scritto alcun registro. Si noti che è ininfluente il fatto che la destinazione sia `N` oppure `Z`, dato che la microistruzione generata dal microassemblatore è identica. I programmati che scelgono intenzionalmente quella "sbagliata", per punizione dovrebbero essere obbligati a lavorare per una settimana intera con il PC IBM originale a 4,77 MHz!

La sintassi per comunicare al microassemblatore di impostare il bit `JAMZ` è

```
if (Z) goto L1; else goto L2
```

Dato che l'hardware richiede che gli 8 bit meno significativi di questi indirizzi siano identici, è compito del microassemblatore assegnare gli indirizzi correttamente. D'altra parte, dato che `L2` può trovarsi in un qualsiasi punto delle prime 256 parole della memoria di controllo, il microassemblatore ha molta libertà nel cercare un altro indirizzo che possa essere accoppiato al primo.

In genere queste ultime due istruzioni vengono combinate; per esempio

```
Z = TOS; if (Z) goto L1; else goto L2
```

Come risultato `MAL` genera una microistruzione in cui `TOS` viene fatto passare attraverso la ALU (ma senza essere memorizzato in alcun registro) in modo che il suo valore imposta il bit `Z`. Poco dopo aver caricato in `Z` il bit di condizione della ALU, viene calcolato in OR all'interno del bit più significativo di `MPC`, obbligando a prelevare l'indirizzo della successiva microistruzione o da `L2` oppure da `L1` (che deve essere 256 più di `L2`). `MPC` sarà stabile e pronto per essere utilizzato nella successiva microistruzione.

Infine abbiamo bisogno di una notazione per utilizzare il bit `JMPC`. Quella che adotteremo è

```
goto (MBR OR valore)
```

Questa sintassi indica al microassemblatore di utilizzare `valore` per `NEXT_ADDRESS` e di impostare il bit `JMPC` in modo che in `MPC` venga calcolato l'OR logico tra `MBR` e `NEXT_ADDRESS`. Se `valore` è 0, allora la situazione è quella normale ed è sufficiente scrivere semplicemente

```
goto (MBR)
```

Si noti che in questo caso non si verifica il problema dell'estensione del segno (cioè la differenza tra `MBR` e `MBRU`), dato che soltanto gli 8 bit meno significativi di `MBR` sono collegati a `MPC` (Figura 4.6). Si osservi inoltre che quello che si utilizza è il valore di `MBR` disponibile alla fine del ciclo corrente. Un prelievo che inizia in questa microistruzione avviene troppo in ritardo per poter modificare la scelta della microistruzione successiva.

4.3.2 Implementazione di IJVM con Mic-1

Siamo finalmente arrivati al momento in cui possiamo mettere insieme tutti i pezzi. La Figura 4.17 è il microprogramma eseguito su `Mic-1` che interpreta `IJVM`. È un programma sorprendentemente corto, con un totale di sole 112 microistruzioni. Per ciascuna microistruzione ci sono tre colonne: l'etichetta simbolica, il microcodice effettivo e un commento. Come abbiamo già sottolineato più volte, si può notare che microistruzioni consecutive non sono necessariamente localizzate in indirizzi consecutivi all'interno della memoria di controllo.

Ormai le scelte dei nomi per la maggior parte dei registri della Figura 4.1 dovrebbero essere ovvie: `CPP`, `LV` e `SP` sono utilizzati per memorizzare, rispettivamente, i puntatori alla porzione costante di memoria, al blocco delle variabili locali e alla cima dello stack, mentre `PC` mantiene l'indirizzo del successivo byte da prelevare dal flusso dell'istruzione. `MBR` è un registro a 1 byte che mantiene in modo sequenziale i byte dello stream dell'istruzione provenienti dalla memoria per poterli interpretare. `TOS` e `OPC` sono registri aggiuntivi il cui utilizzo viene descritto in seguito.

In qualsiasi momento ognuno di questi registri contiene un particolare valore; se necessario è tuttavia possibile utilizzare ciascuno di loro come un registro temporaneo. All'inizio e alla fine di ogni istruzione, `TOS` contiene il valore puntato da `SP`, la parola che si trova in cima allo stack. Questo valore è ridondante, in quanto la parola può essere letta in qualsiasi momento dalla memoria; ciononostante avendola a disposizione in un registro è spesso possibile risparmiare un riferimento alla memoria. Per poche istruzioni il mantenimento di `TOS` implica un numero maggiore di operazioni di memoria. Per

esempio, l'istruzione POP deve leggere dalla memoria la nuova parola che si trova in cima allo stack per poterla copiare all'interno di TOS.

Etichetta	Operazioni	Commenti
Main1	PC = PC + 1; fetch; goto (MBR)	MBR contiene il codice operativo; prelievo del byte successivo; diramazione
nopl	goto Main1	Non esegue nulla
iadd1	MAR = SP = SP - 1; rd	Legge la seconda parola in cima allo stack
iadd2	H = TOS	H = cima dello stack
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Somma le due parole in cima allo stack; scrive in cima allo stack
isub1	MAR = SP = SP - 1; rd	Legge la seconda parola in cima allo stack
isub2	H = TOS	H = cima dello stack
isub3	MDR = TOS = MDR - H; wr; goto Main1	Esegue la sottrazione; scrive in cima allo stack
iand1	MAR = SP = SP - 1; rd	Legge la seconda parola in cima allo stack
iand2	H = TOS	H = cima dello stack
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Esegue l'AND; scrive nella nuova cima dello stack
ior1	MAR = SP = SP - 1; rd	Legge la seconda parola in cima allo stack
ior2	H = TOS	H = cima dello stack
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Esegue l'OR; scrive nella nuova cima dello stack
dup1	MAR = SP = SP + 1	Incrementa SP e lo copia in MAR
dup2	MDR = TOS; wr; goto Main1	Scrive la nuova parola dello stack
pop1	MAR = SP = SP - 1; rd	Legge la seconda parola in cima allo stack
pop2		Attende che il nuovo TOS sia letto dalla memoria
pop3	TOS = MDR; goto Main1	Copia la nuova parola in TOS
swap1	MAR = SP - 1; rd	Imposta MAR a SP - 1; legge la seconda parola dallo stack
swap2	MAR = SP	Imposta MAR con la parola in cima allo stack
swap3	H = MDR; wr	Salva TOS in H; scrive la seconda parola in cima allo stack
swap4	MDR = TOS	Copia il vecchio TOS in MDR
swap5	MAR = SP - 1; wr	Imposta MAR a SP - 1; scrive la seconda parola nello stack
swap6	TOS = H; goto Main1	Aggiorna TOS
bipush1	SP = MAR = SP + 1	MBR = byte da inserire nello stack
bipush2	PC = PC + 1; fetch	Incrementa PC, preleva il successivo codice operativo
bipush3	MDR = TOS = MBR; wr; goto Main1	Estende il segno della costante e la inserisce nello stack
iload1	H = LV	MBR contiene l'indice; copia LV in H
iload2	MAR = MBRU + H; rd	MAR = indirizzo della variabile locale da inserire nello stack
iload3	MAR = SP = SP + 1	SP punta alla nuova cima dello stack; prepara la scrittura
iload4	PC = PC + 1; fetch; wr	Incrementa PC; ottiene il nuovo codice operativo; scrive la cima dello stack
iload5	TOS = MDR; goto Main1	Aggiorna TOS
istore1	H = LV	MBR contiene l'indice; copia LV in H
istore2	MAR = MBRU + H	MAR = indirizzo della variabile locale da memorizzare
istore3	MDR = TOS; wr	Copia TOS in MDR; scrive la parola
istore4	SP = MAR = SP - 1; rd	Legge la nuova parola in cima allo stack
istore5	PC = PC + 1; fetch	Incrementa PC; preleva il successivo codice operativo
istore6	TOS = MDR; goto Main1	Aggiorna TOS

Figura 4.17 Micropogramma per Mic-I.

widel	PC = PC + 1; fetch; goto (MBR OR 0x100)	Preleva il byte dell'operando o il successivo codice operativo Diramazione a più destinazioni con il bit alto impostato a 1
wide_iload1	PC = PC + 1; fetch H = MBRU << 8	MBR contiene il primo byte dell'indice; preleva il secondo H = primo byte dell'indice traslato a sinistra di 8 bit
wide_iload2	H = MBRU OR H	H = indice a 16 bit della variabile locale
wide_iload3	MAR = LV + H;	MAR = indirizzo della variabile locale da inserire nello stack
wide_iload4	rd; goto iload3	
wide_istore1	PC = PC + 1; fetch H = MBRU << 8	MBR contiene il primo byte dell'indice; preleva il secondo H = primo byte dell'indice traslato a sinistra di 8 bit
wide_istore2	H = MBRU OR H	H = indice a 16 bit della variabile locale
wide_istore3	MAR = LV + H;	MAR = indirizzo della variabile locale da memorizzare
wide_istore4	goto istore3	
ldc_w1	PC = PC + 1; fetch H = MBRU << 8	MBR contiene il primo byte dell'indice; preleva il secondo H = primo byte dell'indice << 8
ldc_w2	H = MBRU OR H	H = indice a 16 bit relativo alla porzione costante di memoria
ldc_w3	MAR = H + CPP;	MAR = indirizzo della costante nella porzione costante
ldc_w4	rd; goto iload3	
iinc1	H = LV	MBR contiene l'indice; copia LV in H
iinc2	MAR = MBRU + H; rd	Copia LV + indice in MAR; legge la variabile
iinc3	PC = PC + 1; fetch	Preleva la costante
iinc4	H = MDR	Copia la variabile in H
iinc5	PC = PC + 1; fetch	Preleva il successivo codice operativo
iinc6	MDR = MBR + H;	Inserisce in MDR la somma; aggiorna la variabile
wr; goto Main1		
goto1	OPC = PC - 1	Salva l'indirizzo del codice operativo
goto2	PC = PC + 1; fetch	MBR contiene il primo byte dell'indice; preleva il secondo
goto3	H = MBR << 8	Trasla e salva in H il primo byte con segno
goto4	H = MBRU OR H	H = spiazzamento a 16 bit della diramazione
goto5	PC = OPC + H;	Aggiunge lo spiazzamento a OPC
goto6	fetch goto Main1	Attende il prelievo del successivo codice operativo
iflt1	MAR = SP = SP - 1; rd	Legge la seconda parola in cima allo stack
iflt2	OPC = TOS	Salva temporaneamente TOS in OPC
iflt3	TOS = MDR	Inserisce in TOS la nuova cima dello stack
iflt4	N = OPC; if (N) goto T; else goto F	Diramazione in base al bit N
ifeq1	MAR = SP = SP - 1; rd	Legge la seconda parola in cima allo stack
ifeq2	OPC = TOS	Salva temporaneamente TOS in OPC
ifeq3	TOS = MDR	Inserisce in TOS la nuova cima dello stack
ifeq4	Z = OPC; if (Z) goto T; else goto F	Diramazione in base al bit Z
if_icmpeq1	MAR = SP = SP - 1; rd	Legge la seconda parola in cima allo stack
if_icmpeq2	MAR = SP = SP - 1	Imposta MAR per leggere la nuova cima dello stack
if_icmpeq3	H = MDR; rd	Copia in H la seconda parola dello stack
if_icmpeq4	OPC = TOS	Salva temporaneamente TOS in OPC
if_icmpeq5	TOS = MDR	Inserisce in TOS la nuova cima dello stack
if_icmpeq6	Z = OPC - H; if (Z) goto T; else goto F	Se le due parole in cima allo stack sono uguali, goto T, altrimenti, goto F
T	OPC = PC - 1; goto goto2	Uguale a goto1; necessario per l'indirizzo destinazione
F	PC = PC + 1	Salta il primo byte dello spiazzamento
F2	PC = PC + 1; fetch	PC punta ora al nuovo codice operativo
F3	goto Main1	Attende il prelievo del codice operativo

Figura 4.17 Micropogramma per Mic-I (continua).

invokevirtual1	$PC = PC + 1; \text{fetch}$	MBR = byte 1 dell'indice; incrementa PC; ottiene il secondo byte
invokevirtual2	$H = MBRU << 8$	Trasla e salva in H il primo byte
invokevirtual3	$H = MBRU \text{ OR } H$	$H = \text{spiazzamento del puntatore al metodo rispetto a CPP}$
invokevirtual4	$MAR = CPP + H; \text{rd}$	Ottiene dall'area CPP un puntatore al metodo
invokevirtual5	$OPC = PC + 1$	Salva temporaneamente il PC di ritorno in OPC
invokevirtual6	$PC = MDR; \text{fetch}$	PC punta ora al nuovo metodo; ottiene il numero di parametri
invokevirtual7	$PC = PC + 1; \text{fetch}$	Preleva il secondo byte del numero di parametri
invokevirtual8	$H = MBRU << 8$	Trasla e salva in H il primo byte
invokevirtual9	$H = MBRU \text{ OR } H$	$H = \text{numero di parametri}$
invokevirtual10	$PC = PC + 1; \text{fetch}$	Preleva il primo byte del numero di variabili locali
invokevirtual11	$TOS = SP - H$	$TOS = \text{indirizzo di OBJREF} - 1$
invokevirtual12	$TOS = MAR = TOS + 1$	$TOS = \text{indirizzo di OBJREF (nuovo LV)}$
invokevirtual13	$PC = PC + 1; \text{fetch}$	Preleva il secondo byte del numero di variabili locali
invokevirtual14	$H = MBRU << 8$	Trasla e salva in H il primo byte
invokevirtual15	$H = MBRU \text{ OR } H$	$H = \text{numero di variabili locali}$
invokevirtual16	$MDR = SP + H + 1; \text{wr}$	Sovrascrive OBJREF con il puntatore di collegamento
invokevirtual17	$MAR = SP = MDR;$	Imposta SP e MAR con la locazione in cui memorizzare il vecchio PC
invokevirtual18	$MDR = OPC; \text{wr S}$	Salva il vecchio PC sopra le variabili locali
invokevirtual19	$MAR = SP = SP + 1$	SP punta alla locazione in cui salvare il vecchio LV
invokevirtual20	$MDR = LV; \text{wr}$	Salva il vecchio LV sopra il PC salvato
invokevirtual21	$PC = PC + 1; \text{fetch}$	Preleva il primo codice operativo del nuovo metodo
invokevirtual22	$LV = TOS; \text{goto Main1}$	Imposta LV per puntare al blocco LV
ireturn1	$MAR = SP = LV; \text{rd}$	Reimposta SP e MAR per ricevere il puntatore di collegamento
ireturn2	$LV = MAR = MDR; \text{rd}$	Attende che si completi la lettura
ireturn3	$MAR = LV + 1$	Imposta LV con il puntatore di collegamento; ottiene il vecchio PC
ireturn4	$PC = MDR; \text{rd}; \text{fetch}$	Imposta MAR per leggere il vecchio LV
ireturn5	$MAR = SP$	Ripristina PC; preleva il successivo codice operativo
ireturn6	$LV = MDR$	Imposta MAR per scrivere TOS
ireturn7	$MDR = TOS;$	Ripristina LV
ireturn8	$\text{wr}; \text{goto Main1}$	Salva il valore di ritorno sulla cima originale dello stack

Figura 4.17 Microprogramma per Mic-1 (continua).

Il registro OPC è un registro temporaneo (cioè di “appunti”) e non ha un utilizzo predefinito. È usato per esempio per salvare l’indirizzo del codice operativo di un’istruzione di diramazione, mentre PC viene incrementato per accedere ai parametri. Viene utilizzato come registro temporaneo anche nelle istruzioni di diramazione condizionale di IJVM.

Come tutti gli interpreti, il microprogramma della Figura 4.17 ha un ciclo principale che preleva, decodifica ed esegue le istruzioni del programma interpretato, in questo caso le istruzioni IJVM. Il suo ciclo principale inizia nella linea etichettata con Main1. All’inizio del ciclo deve valere la condizione che PC sia già stato caricato con un indirizzo di una locazione di memoria contenente un codice operativo; inoltre questo codice operativo deve già essere stato memorizzato all’interno di MBR. Ciò implica che prima di iniziare ogni iterazione dobbiamo assicurarci che PC sia stato aggiornato in modo da puntare al successivo codice operativo da interpretare e che il byte del codice operativo stesso sia già stato portato all’interno di MBR.

All’inizio di ogni istruzione viene eseguita la stessa sequenza di operazioni; è importante che la lunghezza di questa sequenza sia la più breve possibile. Attraverso un’attenta progettazione dell’hardware Mic-1 e del software siamo riusciti a ridurre il ciclo principale a una sola istruzione. A partire dal momento in cui viene avviata la macchina, ogni volta che viene eseguita questa microistruzione il codice operativo IJVM da eseguire si trova già all’interno in MBR. Quello che fa la microistruzione è effettuare un salto nel microcodice per eseguire questa istruzione IJVM, oltre a iniziare il prelievo del byte che segue il codice operativo; questo byte potrebbe essere un operando oppure un nuovo codice operativo.

A questo punto possiamo svelare il vero motivo per cui le istruzioni non vengono eseguite sequenzialmente, ma ciascuna di loro è obbligata a indicare esplicitamente il proprio successore. Tutti gli indirizzi della memoria di controllo corrispondenti ai codici operativi devono essere riservati per la prima parola della corrispondente istruzione dell’interprete. Dalla Figura 4.11 vediamo quindi che il codice che interpreta POP inizia in 0x57, mentre il codice che interpreta DUP inizia in 0x59. (Come fa MAL a sapere che POP deve iniziare in 0x57 è uno dei misteri dell’universo, probabilmente esiste da qualche parte un documento segreto che ne spiega il motivo.)

Sfortunatamente il codice corrispondente a POP è lungo tre microistruzioni e quindi, se fosse memorizzato in parole consecutive, interferirebbe con l’inizio di DUP. All’interno di ciascuna sequenza le microistruzioni che seguono quella iniziale devono essere memorizzate nelle locazioni di memoria ancora libere, non riservate per i codici operativi. Per questo motivo occorre effettuare molti salti da una locazione di memoria all’altra; se regolarmente, dopo poche microistruzioni, occorresse utilizzare microdiramazioni esplicite (cioè microistruzioni che effettuano salti) per saltare da un buco a un altro della memoria, il tempo sprecato sarebbe significativo.

Per vedere come funziona l’interprete, assumiamo che MBR contenga il valore 0x60, cioè il codice operativo corrispondente a IADD (Figura 4.11). Nel ciclo principale composto da una sola microistruzione compiamo tre azioni.

1. Incrementiamo PC, in modo che contenga l’indirizzo del primo byte dopo il codice operativo.
2. Iniziamo a prelevare il byte successivo da portare all’interno di MBR. Questo byte, prima o poi, si rivelerà necessario, o come operando per l’istruzione IJVM corrente oppure come codice operativo successivo (come nel caso dell’istruzione IADD, che non ha il byte per l’operando).
3. All’inizio di Main1 effettuiamo una diramazione verso l’indirizzo contenuto in MBR. Questo indirizzo, memorizzato nel registro dalla precedente microistruzione, equivale al valore numerico del codice operativo che in quel momento è in esecuzione. Occorre prestare particolare attenzione al fatto che il valore che si sta iniziando a estrarre in questa microistruzione non gioca alcun ruolo nella diramazione.

Il prelievo del byte successivo comincia in questo punto in modo che sia disponibile all’inizio della terza microistruzione. In seguito potrebbe risultare necessario oppure no, ma conviene comunque far partire il prelievo, dato che la cosa non produce assolutamente alcun danno.

Se il byte contenuto in MBR è composto da soli 0, allora identifica il codice operativo di un'istruzione NOP. In questo caso la successiva microistruzione viene prelevata dalla locazione 0 ed è quella etichettata con `nop1`. Dato che questa istruzione non esegue alcunché essa effettua semplicemente un salto all'inizio del ciclo principale, dove viene ripetuta la sequenza utilizzando però il nuovo codice operativo memorizzato in MBR.

Sottolineiamo ancora una volta il fatto che nella Figura 4.17 le microistruzioni non sono memorizzate consecutivamente all'interno della memoria e che `Main1` non si trova all'indirizzo 0 della memoria di controllo (dato che all'indirizzo 0 deve essere memorizzata `nop1`). È compito del microassemblatore posizionare le microistruzioni negli indirizzi adatti e collegarle in sequenze di breve lunghezza usando il campo `NEXT_ADDRESS`. Ogni sequenza inizia all'indirizzo corrispondente al valore numerico del codice operativo IJVM che interpreta (POP comincia per esempio in 0x57), ma il resto della sequenza può trovarsi in qualsiasi punto della memoria di controllo, e non necessariamente negli indirizzi immediatamente successivi.

Consideriamo ora l'istruzione `IADD` di IJVM. Dopo la diramazione presente nel ciclo principale si raggiunge la microistruzione con l'etichetta `iadd1`. Questa istruzione dà inizio alle operazioni specifiche di `IADD`.

1. TOS è già presente, mentre occorre prelevare dalla memoria la seconda parola a partire dalla cima dello stack.
2. TOS deve essere sommata alla parola che è appena stata prelevata dalla memoria.
3. Il risultato, che deve essere inserito in cima allo stack, va memorizzato sia in memoria sia nel registro TOS.

Per poter prelevare l'operando dalla memoria è necessario decrementare il puntatore allo stack e scriverlo all'interno di MAR. Si noti che, per praticità, questo indirizzo è anche l'indirizzo che sarà utilizzato nella successiva scrittura. Inoltre, dato che questa locazione sarà la nuova cima dello stack, occorre assegnare questo valore anche a SP. Una singola operazione può quindi determinare il nuovo valore di SP e MAR, decrementare SP e scriverlo in entrambi i registri.

Queste azioni vengono svolte durante il primo ciclo, `iadd1`, mentre inizia l'operazione di lettura. Inoltre MPC ottiene l'indirizzo di `iadd2` leggendo il valore del campo `NEXT_ADDRESS` di `iadd1`. Successivamente `iadd2` viene letta dalla memoria di controllo. Durante il secondo ciclo, nell'attesa che l'operando sia letto dalla memoria, la parola in cima allo stack viene copiata da TOS a H, in modo che, una volta completata la lettura, sia disponibile per il calcolo dell'addizione.

All'inizio del terzo ciclo, `iadd3`, MDR contiene l'addendo prelevato dalla memoria. In questo ciclo l'addendo viene sommato al contenuto di H, e il risultato viene memorizzato sia all'interno di MDR sia all'interno di TOS. Viene fatta partire anche un'operazione di scrittura per memorizzare all'interno della memoria la nuova parola che si trova in cima allo stack. In questo ciclo goto assegna a MPC l'indirizzo di `Main1`, facendoci tornare al punto d'inizio, in modo da poter eseguire l'istruzione seguente.

Se il successivo codice operativo IJVM contenuto in MBR è 0x64 (ISUB), la sequenza di eventi che si svolge è praticamente identica a quella appena vista. Dopo aver eseguito `Main1`, il controllo viene trasferito alla microistruzione che si trova all'indirizzo 0x64

(`isub1`). Questa microistruzione è seguita da `isub2` e `isub3`, e poi nuovamente da `Main1`. L'unica differenza tra questa sequenza e la precedente è che in `isub3` il contenuto di H viene sottratto da MDR invece di sommarlo.

L'interpretazione di `IAND` è praticamente identica a quella di `IADD` e di `ISUB`, tranne per il fatto che si calcola l'AND tra le due parole in cima allo stack invece di sommarle oppure sottrarle. Nel caso dell'istruzione `IOR` la situazione è del tutto simile.

Se il codice operativo IJVM è `DUP`, `POP` oppure `SWAP` occorre modificare lo stack. L'istruzione `DUP` replica semplicemente la parola che si trova in cima allo stack. Dato che il valore di questa parola è già contenuto in `TOS`, l'operazione non fa altro che incrementare `SP` in modo che punti alla nuova locazione e memorizzare `TOS` in quella posizione. L'istruzione `POP` è altrettanto semplice e consiste nel decrementare `SP` per scartare la parola in cima allo stack. Tuttavia, per mantenere corretto il valore memorizzato in `TOS`, occorre leggere dalla memoria la nuova parola che si trova in cima allo stack per scriverla all'interno di `TOS`. Infine l'istruzione `SWAP` effettua lo scambio dei valori contenuti nelle due locazioni che si trovano in cima allo stack. L'operazione è facilitata dal fatto che `TOS` contiene già uno di questi valori, e non è necessario leggerlo dalla memoria. In seguito tratteremo in modo più dettagliato questa istruzione.

L'istruzione `BIPUSH` è un po' più complicata poiché, come mostra la Figura 4.18, il codice operativo è seguito da un solo byte, che deve essere interpretato come un intero con segno. Questo byte, che durante `Main1` è già stato portato all'interno di MBR, deve essere esteso con segno fino a 32 bit e copiato all'interno di MDR. Infine `SP` viene incrementato e copiato in MAR, in modo da permettere la scrittura dell'operando in cima allo stack. Questo operando deve anche essere copiato nel registro `TOS`. Inoltre è importante notare che, prima di ritornare al programma principale, PC deve essere nuovamente incrementato in modo da rendere disponibile in `Main1` il successivo codice operativo.

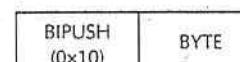
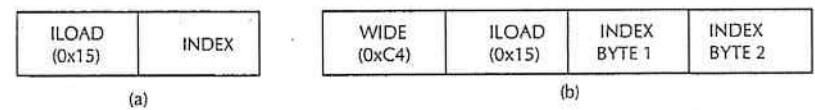


Figura 4.18 Formato dell'istruzione BIPUSH.



(a)

(b)

Figura 4.19 (a) ILOAD con un indice a 1 byte. (b) WIDE ILOAD con un indice a 2 byte.

Consideriamo ora l'istruzione `ILOAD`. Come mostra la Figura 4.19(a) anche in questa istruzione il codice operativo è seguito da un byte. In questo caso si tratta però di un indice (senza segno) usato per identificare nello spazio delle variabili locali la parola da mettere nello stack. Dato che si usa un solo byte è possibile distinguere soltanto $2^8 = 256$

parole, per la precisione le prime 256 parole dello spazio delle variabili locali. L'istruzione **ILOAD** richiede sia una lettura (per ottenere la parola) sia una scrittura (per porla in cima allo stack). Tuttavia per poter determinare l'indirizzo di lettura occorre sommare lo spiazzamento, contenuto in **MBR**, al valore di **LV**. Dato che è possibile accedere sia a **MBR** sia a **LV** solamente attraverso il bus B, inizialmente si copia **LV** in **H** (in **iLoad1**), e poi si somma **MBR**. Dopo che il risultato è stato copiato in **MAR** può aver inizio una lettura (in **iLoad2**).

Tuttavia l'utilizzo di **MBR** come indice è leggermente diverso dall'uso che ne è stato fatto in **BIPUSH**. Nel caso di un indice il suo valore è sempre positivo e quindi il byte che lo rappresenta deve essere considerato come un intero senza segno; nel caso dell'istruzione **BIPUSH** esso è stato invece interpretato come un intero a 8 bit con segno. L'interfaccia tra **MBR** e il bus è stata attentamente progettata in modo da consentire entrambi i tipi di operazione. Nel caso di **BIPUSH** (intero con segno a 8 bit) l'operazione corretta è l'estensione con segno, in cui il bit più a sinistra del primo byte di **MBR** viene copiato nei 24 bit più alti del bus B. Nel caso di **ILOAD** (intero senza segno a 8 bit) l'operazione corretta è invece un riempimento con valori 0; in questo caso tutti i 24 bit più alti del bus B vengono impostati a 0. Usando segnali separati è possibile distinguere le due operazioni e indicare quale deve essere effettuata (Figura 4.6). Nel microcodice ciò è indicato da **MBR** (estensione con segno, come in **bipush3**) oppure da **MBRU** (senza segno, come in **iLoad2**).

Mentre si attende che la memoria fornisca l'operando (in **iLoad3**), **SP** viene incrementato in modo che il suo valore indichi la locazione in cui memorizzare il risultato, ovvero la nuova cima dello stack. Questo valore viene copiato anche in **MAR** per preparare la scrittura dell'operando nello stack. **PC** deve essere nuovamente incrementato per prelevare il codice operativo successivo (in **iLoad4**). Infine **MDR** viene copiato in **TOS** in modo da riflettere correttamente la nuova cima dello stack (in **iLoad5**).

ISTORE è l'operazione inversa di **ILOAD** e consiste nel rimuovere una parola dalla cima dello stack e nel memorizzarla nella locazione specificata dalla somma tra **LV** e l'indice contenuto nell'istruzione. **ISTORE** ha lo stesso formato di **ILOAD**, mostrato nella Figura 4.19(a), tranne per il fatto che il codice operativo è 0x36 invece che 0x15. Dato che si conosce già (in **TOS**) il valore della parola in cima allo stack, si potrebbe pensare che sia sufficiente memorizzare il contenuto di **TOS** direttamente in memoria; in realtà il comportamento dell'istruzione è leggermente più complesso. Occorre infatti prelevare la nuova parola che si trova in cima allo stack e per far ciò è necessario effettuare una lettura oltre alla scrittura; tuttavia non è importante l'ordine secondo il quale vengono eseguite queste due operazioni di memoria (se possibile possono anche essere eseguite in parallelo).

Sia **ILOAD** sia **ISTORE** possono accedere solamente alle prime 256 variabili locali. Ovviamente, anche se nella maggior parte dei programmi questo limite potrebbe essere superiore allo spazio effettivamente necessario, è fondamentale avere la possibilità di accedere a una variabile ovunque essa si trovi (nello spazio delle variabili locali). Per superare questa restrizione IJVM utilizza lo stesso meccanismo impiegato in JVM: si usa uno speciale codice operativo, **WIDE**, conosciuto come **byte di prefisso**, seguito dal codice operativo di **ILOAD** oppure da quello di **ISTORE**. Quando si incontra questa sequenza si modificano le definizioni di **ILOAD** e **ISTORE**, in modo che l'indice che segue

il codice operativo sia un valore a 16 bit invece che a 8 bit. L'uso del codice operativo **WIDE** è mostrato nella Figura 4.19(b).

WIDE viene decodificato nel solito modo, effettuando una diramazione che porta all'istruzione **wide1** in cui inizia il microcodice specifico per la gestione di questo codice operativo. Anche se il codice operativo sul quale si vuole effettuare l'"allargamento" è già presente in **MBR**, **wide1** preleva il primo byte che lo segue, dato che la logica del microprogramma si aspetta sempre di trovarlo in quella posizione. In **wide2** viene effettuata una seconda diramazione in base al byte che segue **WIDE**. Tuttavia, dato che sia **WIDE ILOAD** sia **WIDE ISTORE** richiedono microcodice diverso rispetto a **ILOAD** e **ISTORE**, la seconda diramazione non può utilizzare il codice operativo come indirizzo di destinazione, come invece fa **Main1**.

In **wide2** viene invece calcolato l'OR tra il codice operativo e 0x100, e il risultato viene memorizzato in **MPC**. Come conseguenza l'interpretazione di **WIDE ILOAD** comincia in 0x115 (al posto di 0x15) e l'interpretazione di **WIDE ISTORE** comincia in 0x136 (al posto di 0x36), e così via. Così facendo ogni codice operativo di tipo **WIDE** inizia a un indirizzo che è 256 (cioè, 0x100) parole più in alto rispetto al corrispondente codice operativo di tipo normale. La Figura 4.20 mostra la sequenza iniziale di microistruzioni sia per **ILOAD** sia per **WIDE ILOAD**.

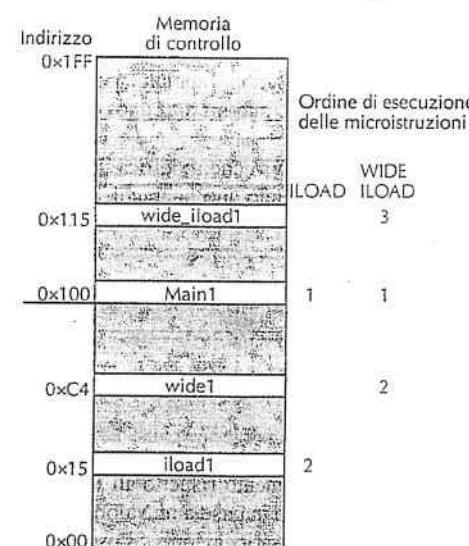


Figura 4.20 La sequenza iniziale di microistruzioni per **ILOAD** e **WIDE ILOAD**. Gli indirizzi sono puramente esemplificativi.

Raggiunto il codice che implementa **WIDE ILOAD** (0x115), esso differisce dal normale codice di **ILOAD** soltanto per il fatto che l'indice deve essere costruito concatenando 2

byte invece che estendendo con segno un unico byte. La concatenazione e la successiva addizione devono essere compiute in passi distinti, il primo dei quali consiste nel copiare in H il byte 1 dell'indice traslato a sinistra di 8 bit. Dato che l'indice è un intero senza segno, MBR viene poi esteso con valori zero utilizzando MBRU. Successivamente viene sommato il secondo byte (l'operazione di addizione è identica alla concatenazione, dato che ora il byte meno significativo di H vale zero, garantendo quindi che non ci sarà alcun riporto tra i byte) e il risultato viene memorizzato nuovamente in H. A partire da questo punto l'operazione può procedere, esattamente come se fosse il caso normale di ILOAD. Piuttosto che duplicare le ultime istruzioni di ILOAD (da iload3 a iload5), abbiamo semplicemente introdotto un salto da wide_iload4 a iload3. Si noti tuttavia che, durante l'esecuzione dell'istruzione, PC deve essere incrementato due volte affinché punti correttamente al successivo codice operativo. Sia l'istruzione ILOAD, sia la sequenza WIDE_ILOAD lo incrementano una volta.

La stessa situazione si verifica nel caso di WIDE_ISTORE: dopo aver eseguito le prime quattro microistruzioni (da wide_istore1 a wide_istore4), la sequenza è identica a quella di ISTORE a partire dalla terza istruzione. Viene quindi introdotto un salto da wide_istore4 a istore3.

La successiva istruzione d'esempio che prendiamo in considerazione è LDC_W. Questo codice operativo differisce da ILOAD per due aspetti. In primo luogo, l'istruzione ha uno spiazzamento di 16 bit senza segno (come la versione "allargata" di ILOAD). In secondo luogo, l'indice è relativo a CPP e non a LV, dato che la sua funzione consiste nel leggere dalla porzione costante di memoria e non dal blocco delle variabili locali. (In realtà esiste anche una versione corta di LDC_W, chiamata LDC; non l'abbiamo però inclusa in IJVM, dato che la versione lunga comprende già al suo interno tutti i possibili casi di quella corta, anche se richiede 3 byte al posto di soli 2.)

L'istruzione IINC è, oltre a ISTORE, l'unica istruzione IJVM che modifica una variabile locale. Come mostra la Figura 4.21, essa lo fa includendo due operandi, ciascuno dei quali è lungo 1 byte.

IINC (0x84)	INDEX	CONST
----------------	-------	-------

Figura 4.21 L'istruzione INC ha due diversi campi per gli operandi.

L'istruzione IINC utilizza INDEX per specificare lo spiazzamento rispetto all'inizio del blocco delle variabili locali, legge la variabile, la incrementa in base a un valore, CONST, contenuto nell'istruzione e memorizza il risultato nella stessa locazione. Si osservi che questa istruzione può incrementare di un valore negativo, nel senso che CONST è una costante a 8 bit con segno, il cui valore è dunque compreso tra -128 e +127. La versione completa di JVM comprende una versione "allargata" di IINC in cui ciascun operando è lungo 2 byte.

Arriviamo ora alla prima istruzione di salto di IJVM: GOTO. L'unica funzione di questa istruzione è quella di modificare il valore di PC, in modo che la successiva istruzione IJVM da eseguire sia quella che si trova nell'indirizzo calcolato sommando lo spiazzamento a 16 bit (con segno) all'indirizzo del codice operativo del salto. Una complicazione è data dal fatto che lo spiazzamento è relativo al valore che PC aveva all'inizio della decodifica dell'istruzione e non a quello prelevato dopo i due byte dello spiazzamento.

Per chiarire questo punto osserviamo nella Figura 4.22(a) la situazione all'inizio di Main1. Il codice operativo è già presente in MBR, ma PC non è ancora stato incrementato. Nella Figura 4.22(b) vediamo la situazione all'inizio di goto1. A questo punto PC è stato incrementato, ma il primo byte dello spiazzamento non è ancora stato prelevato e salvato in MBR. Una microistruzione dopo ci troviamo nel caso della Figura 4.22(c) in cui il vecchio PC, che punta al codice operativo, è stato salvato in OPC e il primo byte dello spiazzamento è stato memorizzato in MBR. Il valore salvato in OPC è necessario, in quanto lo spiazzamento dell'istruzione IJVM GOTO è relativo a esso e non al valore corrente di PC. Questo è in realtà il motivo principale per il quale abbiamo bisogno del registro OPC.

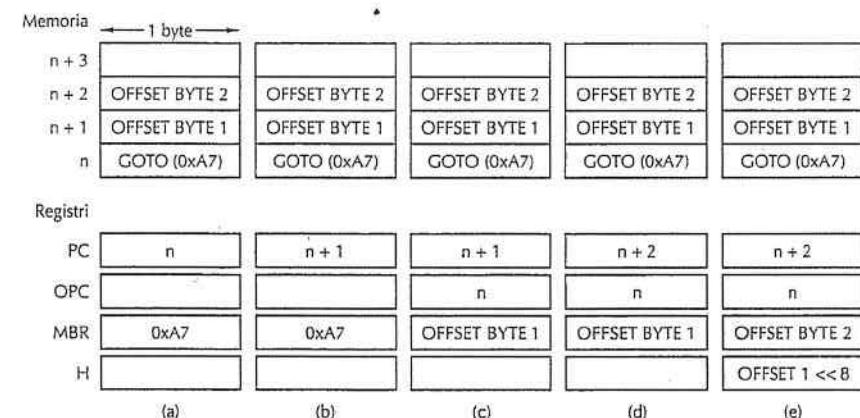


Figura 4.22 Situazione all'inizio di alcune microistruzioni. (a) Main1. (b) goto1. (c) goto2. (d) goto3. (e) goto4.

La microistruzione in goto2 inizia a prelevare il secondo byte dello spiazzamento, portando, all'inizio di goto3, alla situazione mostrata nella Figura 4.22(d). Dopo che il primo byte dello spiazzamento è stato traslato a sinistra di 8 bit e copiato in H, arriviamo a goto4 e allo stato della Figura 4.22(e). Ora il primo byte dello spiazzamento si trova in H traslato a sinistra di 8 bit, il secondo byte dello spiazzamento è in MBR e l'indirizzo base è salvato in OPC. In goto5 otteniamo il nuovo indirizzo da memorizzare in PC.

costruendo in H lo spiazzamento completo a 16 bit e sommandolo alla base. Si faccia attenzione al fatto che in `goto4` utilizziamo `MBRU` al posto di `MBR`, dato che non vogliamo estendere con segno il secondo byte. Inoltre lo spiazzamento a 16 bit viene costruito calcolando in realtà l'OR logico tra le due metà. Infine, prima di tornare all'istruzione `Main1`, dobbiamo prelevare il successivo codice operativo, dato che all'inizio di ogni iterazione del ciclo principale il microporgramma si aspetta di trovarlo all'interno di `MBR`.

Nell'istruzione IJVM `goto` gli spiazzamenti sono valori a 16 bit con segno, compresi tra un minimo di -32768 e un massimo di $+32767$. Questo significa che non è possibile effettuare salti verso etichette la cui distanza superi questi limiti. Questa proprietà può essere vista come un bug oppure come una delle caratteristiche funzionali di IJVM (e anche di JVM). Chi sostiene che sia un bug ritiene che la definizione di JVM non dovrebbe restringere lo stile di programmazione. Chi invece sostiene che sia una caratteristica di JVM crede che il lavoro di molti programmatore migliererebbe radicalmente se fossero inseguiti dall'incubo di veder apparire questo temibile messaggio da parte del compilatore.

**Il programma è troppo grande e complesso. Occorre riscriverlo.
La compilazione è annullata.**

Sfortunatamente (dal nostro punto di vista) questo messaggio appare solamente quando una clausola `then` o `else` supera 32 KB, corrispondenti a circa 50 pagine di codice Java.

Consideriamo ora le tre istruzioni di salto condizionale di IJVM: `IFLT`, `IFEQ` e `IF_ICMPEQ`. Le prime due estraggono la parola in cima allo stack ed effettuano una diramazione se e solo se è rispettivamente minore di zero oppure uguale a zero. L'istruzione `IF_ICMPEQ` estrae invece le due parole in cima allo stack e compie una diramazione se e solo se sono uguali. In tutti e tre i casi è necessario leggere la nuova parola che si trova in cima allo stack e memorizzarla in `TOS`.

Per tutte e tre queste istruzioni il controllo è simile. Inizialmente l'operando, o gli operandi, vengono salvati nei registri; successivamente viene letto il nuovo valore in cima allo stack e inserito in `TOS`; infine vengono effettuati il test e la diramazione. Come prima istruzione consideriamo `IFLT`. La parola da testare è già presente in `TOS`, ma dato che `IFLT` estrae una parola dallo stack occorre leggere la nuova parola che si trova in cima allo stack per memorizzarla in `TOS`. Questa lettura inizia in `iflt1`. Durante `iflt2` viene temporaneamente salvata in `OPC` la parola da testare, in modo che il nuovo valore possa essere memorizzato in `TOS` senza perdere il valore corrente. Infine, durante `iflt4`, la parola che si vuole testare, salvata in `OPC`, viene fatta passare nella ALU senza memorizzarla in modo da testare il bit `N`. Questa microistruzione contiene anche una diramazione che sceglie la destinazione `T` se il test ha successo, mentre la `F` in caso contrario.

Se il test ha successo il resto dell'operazione è essenzialmente uguale all'istruzione `GOTO`; la sequenza continua quindi con `goto2`, in un punto intermedio della sequenza di `GOTO`. Se invece il test fallisce occorre eseguire una sequenza di breve durata (`F`, `F2` e `F3`) per scavalcare il resto dell'istruzione (lo spiazzamento) prima di ritornare a `Main1` e continuare con l'istruzione successiva.

Il codice in `ifeq2` e `ifeq3` segue la stessa logica, con l'unica differenza che si usa il bit `Z` invece del bit `N`. In entrambi i casi è compito dell'assemblatore di `MAL` riconoscere che gli indirizzi `T` e `F` sono speciali e assicurarsi che siano posizionati nella memoria di controllo in modo che i loro indirizzi differiscano soltanto nel bit più a sinistra.

Anche la logica dell'istruzione `IF_ICMPEQ` è più o meno simile a quella di `IFEQ`, tranne per il fatto che in questo caso dobbiamo leggere anche il secondo operando. Il secondo operando viene caricato in H durante `if_icmpeq3`, quando inizia la lettura della nuova parola in cima allo stack. Anche questa volta la parola che si trova in cima allo stack viene salvata in `OPC` e quella nuova viene memorizzata in `TOS`. Infine il test `if_icmpeq6` è simile a quello in `ifeq4`.

Consideriamo ora l'implementazione di `INVOKEVIRTUAL` e `IRETURN`; com'è stato descritto nel Paragrafo 4.2.3 sono le istruzioni che permettono di richiamare una procedura e di restituire il controllo al chiamante. `INVOKEVIRTUAL` è l'istruzione più complessa implementata in IJVM ed è costituita da una sequenza di 22 microistruzioni, mostrata nella Figura 4.12. L'istruzione utilizza uno spiazzamento a 16 bit per determinare l'indirizzo del metodo da invocare. Nella nostra implementazione lo spiazzamento è semplicemente un valore di distanza all'interno della porzione costante di memoria; questa locazione punta al metodo da invocare. Non dimentichiamo però che i primi 4 byte di ciascun metodo *non* sono istruzioni; essi contengono infatti due puntatori a 16 bit. Il primo fornisce il numero di parole usate per i parametri (compreso `OBJREF`, vedi Figura 4.12). Il secondo fornisce invece la dimensione dell'area delle variabili locali espressa come numero di parole. Questi campi sono prelevati attraverso la porta a 8 bit e vengono assemblati come se fossero due spiazzamenti a 16 bit presenti all'interno di un'istruzione.

Successivamente vengono memorizzate delle informazioni di collegamento necessarie per poter riportare la macchina allo stato precedente. Queste informazioni comprendono l'indirizzo d'inizio di ciascuna area delle variabili locali oltre al precedente valore di `PC`; esse sono memorizzate immediatamente sopra l'area delle variabili locali appena create e sotto il nuovo stack. Infine, prima di ritornare a `Main1` e poter iniziare l'esecuzione della nuova istruzione, viene prelevato il codice operativo dell'istruzione successiva e `PC` viene incrementato.

`IRETURN` è un'istruzione semplice che non contiene alcun operando. Essa utilizza semplicemente l'indirizzo memorizzato nella prima parola dell'area delle variabili locali per recuperare le informazioni di collegamento. In seguito reimposta `SP`, `LV` e `PC` ai loro precedenti valori e, come mostra la Figura 4.13, copia in cima allo stack originale il valore di ritorno presente in cima allo stack corrente.

4.4 Progettazione del livello di microarchitettura

La progettazione del livello di microarchitettura, come tutto ciò che riguarda i calcolatori, è caratterizzata da compromessi e bilanciamenti. Sono molte le caratteristiche di un calcolatore che si vorrebbero ottimizzare: velocità, costi, affidabilità, facilità di utilizzo, consumo energetico e dimensioni fisiche. Fra tutti i possibili compromessi ce n'è uno in particolare che determina le principali scelte che un progettista deve compiere: l'equili-

brio tra la velocità della macchina e i suoi costi. In questo paragrafo analizzeremo il problema nel dettaglio per vedere quali elementi occorre bilanciare, come riuscire a ottenere prestazioni elevate e a quale costo, in termini di hardware e complessità, ciò sia possibile.

4.4.1 Velocità/costi

Ultimamente l'impiego di nuove tecnologie ha prodotto nei calcolatori il maggior incremento di velocità finora registrato; tuttavia questo aspetto esula dagli scopi del libro, dato che siamo maggiormente interessati agli aumenti di velocità dovuti all'architettura. I miglioramenti dovuti a modifiche dell'architettura dei calcolatori, pur non essendo stati stupefacenti quanto quelli dovuti all'impiego di circuiti più rapidi, hanno avuto comunque un impatto notevole. La velocità può essere misurata in più modi; in ogni caso, una volta scelto un ISA e stabilita la tecnologia dei circuiti, esistono principalmente tre approcci tramite i quali è possibile aumentare la velocità di esecuzione.

1. Ridurre il numero di cicli di clock necessari per eseguire un'istruzione.
2. Semplificare l'organizzazione in modo che il ciclo di clock possa essere più breve.
3. Sovrapporre l'esecuzione delle istruzioni.

Mentre i primi due punti sono ovvi, esiste un numero sorprendentemente alto di possibili progettazioni che possono influire in modo drastico sul numero di cicli di clock, sul periodo di clock o, molto spesso, su entrambi i fattori. In questo paragrafo daremo un esempio di come la codifica e la decodifica di un'istruzione siano in grado di modificare il ciclo di clock.

Il numero di cicli di clock necessario per eseguire un insieme di operazioni è conosciuto con il nome di **lunghezza del percorso**. In alcuni casi la lunghezza del percorso può essere accorciata aggiungendo dei componenti hardware specializzati. Se per esempio si aggiunge al PC un incrementatore (ovvero un sommatore in cui un addendo è sempre impostato a 1) è possibile eliminare alcuni cicli, dato che non è più necessario utilizzare la ALU per far avanzare il suo valore; in questo caso il prezzo da pagare è rappresentato dall'hardware aggiuntivo. Tuttavia questa possibilità non aiuta tanto quanto ci si potrebbe aspettare. Per la maggior parte delle istruzioni, durante gli stessi cicli spesi per incrementare PC, viene infatti svolta un'operazione di lettura; di conseguenza l'esecuzione dell'istruzione successiva non può essere anticipata, dato che dipende dal dato che deve giungere dalla memoria.

Se si intende ridurre il numero di cicli necessari per prelevare le istruzioni occorre qualcosa di più rispetto al solo circuito che incrementa il PC. La tecnica più efficace per una significativa accelerazione del prelievo delle istruzioni è la terza, ovvero la sovrapposizione dell'esecuzione di più istruzioni. Separare i componenti del circuito relativi al prelievo dell'istruzione (la porta a 8 bit della memoria e i registri MBR e PC) è più efficace se si rende quest'unità indipendente, dal punto di vista funzionale, dal percorso dati. In questo modo l'unità può prelevare autonomamente il successivo codice operativo o operando, anche in modo asincrono rispetto al resto della CPU. In tal modo è possibile prelevare in anticipo una o più istruzioni.

Nell'esecuzione di molte istruzioni una delle operazioni più-costose-in termini di tempo consiste nel prelevare uno spiazzamento a 2 byte, nell'estenderlo in modo appropriato e nel depositarlo nel registro H in preparazione di un'addizione (ciò serve, per esempio, per calcolare l'indirizzo $PC \pm n$ a cui porta una diramazione). Una possibile soluzione potrebbe essere quella di rendere la porta della memoria larga 16 bit. Purtroppo però questa scelta complicherebbe l'operazione; la larghezza della memoria è infatti di 32 bit e, dato che i 16 bit richiesti potrebbero superare i limiti della parola, anche una singola lettura di 32 bit non riuscirebbe necessariamente a prelevare entrambi i byte richiesti.

La sovrapposizione dell'esecuzione delle istruzioni è di gran lunga la soluzione più interessante e offre le maggiori opportunità per un incremento drastico della velocità. La semplice sovrapposizione del prelievo e dell'esecuzione di un'istruzione ha un'efficacia sorprendente. Esistono anche alcune tecniche più sofisticate che si spingono oltre, permettendo per esempio di sovrapporre l'esecuzione di più istruzioni. Questa idea è infatti il cuore delle architetture dei calcolatori moderni. In seguito illustreremo alcune delle tecniche più semplici per sovrapporre l'esecuzione delle istruzioni e spiegheremo quali sono le motivazioni che portano all'adozione di tecniche ancora più sofisticate.

La velocità costituisce soltanto metà del problema; l'altra è rappresentata dai costi. Anche il costo può essere misurato in vari modi, ma è difficile darne una precisa definizione. Alcune misure si basano semplicemente sul numero di componenti (il che aveva un senso quando i processori venivano costruiti a partire da componenti acquistati separatamente e poi assemblati). Oggi l'intero processore è invece realizzato su un unico chip. Resta comunque vero che chip più complessi sono più costosi rispetto a quelli più semplici e piccoli. Anche se è possibile contare i singoli componenti, come i transistor, le porte logiche e le unità funzionali, spesso questo valore non è importante quanto invece lo è l'area occupata sul circuito integrato. Maggiori sono le funzioni incluse nel processore e più grande diventa il chip; all'aumentare delle sue dimensioni i costi di produzione crescono, ma in modo molto più veloce rispetto alla sua area. Per questo motivo spesso i progettisti parlano di costi in termini di "proprietà terriera", riferendosi in questo modo all'area richiesta dal circuito (presumibilmente misurata in pico-acri).

Uno dei circuiti che nella storia è stato studiato in modo più accurato è il sommatore. Sono stati realizzati migliaia di progetti e quelli più veloci, oltre a essere molto complessi, hanno prestazioni decisamente migliori rispetto a quelli più lenti. Il progettista di sistema deve decidere se l'aumento di velocità giustifica l'aumento della "proprietà terriera".

I sommatori non sono però gli unici componenti per i quali esistono più scelte possibili. Praticamente qualsiasi componente del sistema può essere progettato per funzionare più velocemente o più lentamente, con un'influenza sul costo. La sfida che si presenta al progettista è quella di identificare i componenti che possono migliorare il sistema nel modo più marcato, aumentandone la velocità. È interessante sapere che spesso un singolo componente può essere sostituito da uno molto più veloce producendo però soltanto un effetto limitato, se non addirittura nullo, sulla velocità globale del sistema. Nei paragrafi successivi analizzeremo alcuni problemi di progettazione e i relativi compromessi nella scelta dei componenti.

Per determinare a quale velocità può arrivare il clock, uno dei fattori chiave è la quantità di lavoro da eseguire durante ogni ciclo. Ovviamente all'aumentare del lavoro da eseguire il ciclo di clock si allunga. In realtà però la cosa non è così semplice, dato che l'hardware è sufficientemente evoluto per eseguire più compiti in parallelo; ciò che determina effettivamente la lunghezza del ciclo di clock è quindi la sequenza di operazioni che devono essere eseguite *serialmente* durante ogni singolo ciclo.

Un aspetto che può essere controllato è la quantità di operazioni di decodifica da eseguire. Per esempio, nella Figura 4.6 abbiamo visto che, nonostante qualunque dei nove registri potesse essere caricato nella ALU dal bus B, all'interno della parola della microistruzione avevamo bisogno solamente di 4 bit per indicare il registro selezionato. Purtroppo questo risparmio ha i suoi contro: il ritardo introdotto dal circuito di decodifica nel percorso dati. Ciò significa che il registro selezionato per portare i propri dati sul bus B riceverà il comando leggermente in ritardo e solo in seguito i suoi dati verranno effettivamente messi sul bus. Ciò provoca un effetto a catena: la ALU riceverà i propri input, e quindi produrrà il risultato, leggermente in ritardo e, con lo stesso ritardo, il risultato verrà scritto sul bus C. Dato che spesso questo ritardo determina quanto deve essere lungo il ciclo di clock, ciò potrebbe limitare la sua frequenza, costringendo quindi l'intero calcolatore a funzionare a una velocità leggermente inferiore. C'è dunque un bilanciamento tra velocità e costi. La riduzione della memoria di controllo di 5 bit per parola avviene a spese di un rallentamento del clock. Il progettista, quando deve compiere la scelta più appropriata, deve tener ben presente gli obiettivi della progettazione. Se si vuole implementare un sistema ad alte prestazioni, l'utilizzo di un decodificatore non è probabilmente una buona idea, ma lo è per i sistemi a basso costo.

4.4.2 Riduzione della lunghezza del percorso di esecuzione

La macchina Mic-1 è stata progettata per essere allo stesso tempo moderatamente semplice e moderatamente veloce, anche se questi obiettivi sono chiaramente in conflitto l'uno con l'altro. In poche parole, le macchine semplici non sono veloci e le macchine veloci non sono semplici. La CPU di Mic-1 utilizza inoltre una quantità minima di hardware: 10 registri, la semplice ALU della Figura 3.19 replicata 32 volte, uno shifter, un decodificatore, una memoria di controllo e alcuni bit qua e là, necessari per tenere insieme i diversi componenti. L'intero sistema potrebbe essere costruito con meno di 5000 transistor, la ROM di controllo e la memoria centrale.

Dopo aver visto com'è possibile implementare IJVM mediante microcodice e con poco hardware è giunto ora il momento di analizzare delle alternative più veloci. Studieremo ora alcuni modi per ridurre il numero di microistruzioni delle istruzioni ISA (cioè, per ridurre la lunghezza del percorso di esecuzione).

Unione del ciclo d'interpretazione con il microcodice

In Mic-1 il ciclo principale consiste in una microistruzione da eseguire all'inizio di ogni istruzione IJVM. In alcuni casi è possibile sovrapporla all'istruzione precedente e in realtà in parte questo è già stato fatto. Notiamo infatti che quando Main1 viene eseguito il codice operativo da interpretare è già presente all'interno di MBR. Il codice operativo

si trova già nel registro, in quanto è già stato prelevato o dal precedente ciclo principale oppure durante l'esecuzione dell'istruzione precedente.

È possibile spingere ancora più in avanti quest'idea, che consiste nel sovrapporre l'inizio dell'istruzione. In alcuni casi è possibile addirittura ridurre il ciclo principale fino a farlo scomparire. Ciò può verificarsi nel modo seguente. Consideriamo le sequenze di microistruzioni che terminano con un salto a Main1. In ognuno di questi punti l'istruzione del ciclo principale può essere spostata alla fine della sequenza (piuttosto che all'inizio della sequenza successiva), replicando così in più punti del microcodice la diramazione a più destinazioni (mantenendo però sempre lo stesso insieme di destinazioni). In alcuni casi la microistruzione Main1 può essere unita alle microistruzioni precedenti, dato che non sempre queste vengono utilizzate completamente.

La Figura 4.23 mostra la sequenza dinamica di microistruzioni nel caso di un'istruzione POP. Il ciclo principale si verifica prima e dopo ciascuna istruzione, anche se la figura mostra solamente l'occorrenza successiva all'istruzione POP. Notiamo che l'esecuzione di questa istruzione richiede quattro cicli di clock: tre per le microistruzioni specifiche di POP e uno per il ciclo principale.

Etichetta	Operazioni	Commenti
pop1	MAR = SP = SP - 1; rd	Legge la parola sotto la cima dello stack
pop2		Attende che il nuovo TOS sia letto dalla memoria
pop3	TOS = MDR; goto Main1	Copia la nuova parola in TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR contiene il codice operativo; prelievo del byte successivo; diramazione

Figura 4.23 Sequenza della versione originale del microprogramma per l'esecuzione di POP.

Nella Figura 4.24 la sequenza è stata ridotta a tre istruzioni, unendo le istruzioni del ciclo principale alle altre; per eseguire alcune delle operazioni di Main1 è stato infatti sfruttato il ciclo di clock pop2 in cui la ALU non veniva utilizzata. È importante notare che questa sequenza termina con una diramazione che porta l'esecuzione direttamente nel punto del codice specifico per l'istruzione successiva: complessivamente tre cicli sono sufficienti. Questa piccola modifica riduce di un ciclo il tempo di esecuzione della successiva microistruzione. Ciò equivale ad accelerare il clock da 250 MHz (microistruzioni da 4 ns) a 333 MHz (microistruzioni da 3 ns), senza realmente modificare la temporizzazione.

L'istruzione POP si presta particolarmente bene a questo trattamento, dato che nel mezzo della sua esecuzione è presente un ciclo inattivo in cui la ALU non viene utilizzata.

Dato che il ciclo principale richiede l'uso della ALU, se si vuole ridurre la lunghezza di un'istruzione, è necessario trovare al suo interno un ciclo in cui la ALU è inattiva. Questi cicli "morti" non sono comuni, ma quando si verificano vale la pena unire Main1 alla fine di ciascuna sequenza di microistruzioni. Il prezzo da pagare è solamente un

piccolo incremento della memoria di controllo. È possibile formulare in questo modo la prima tecnica che ci permette di ridurre la lunghezza del percorso:

unire il ciclo d'interpretazione alla fine delle sequenze di microcodice.

Etichetta	Operazioni	Commenti
pop1	MAR = SP = SP - 1; rd	Legge la parola sotto la cima dello stack
Main1.pop	PC = PC + 1; fetch	MBR contiene il codice operativo; prelievo del byte successivo
pop3	TOS = MDR; goto (MBR)	Copia la nuova parola in TOS; diramazione a seconda del codice operativo

Figura 4.24 Sequenza della versione migliorata del microprogramma per l'esecuzione di POP.

Architettura a tre bus

Cos'altro possiamo fare per ridurre la lunghezza del percorso di esecuzione? Un'altra semplice modifica consiste nell'utilizzo di due bus di input, A e B, per la ALU, per un totale di tre bus. Tutti (almeno la maggior parte) dei registri dovrebbero avere accesso a entrambi i bus di input. Il vantaggio di avere due bus di input è che in questo modo diventa possibile sommare fra loro, in unico ciclo, due registri qualsiasi. Per comprendere il valore di questa funzionalità consideriamo l'implementazione di ILOAD per Mic-1 (Figura 4.25).

Etichetta	Operazioni	Commenti
iload1	H = LV	MBR contiene l'indice; copia LV in H
iload2	MAR = MBRU + H; rd	MAR = indirizzo della variabile locale da inserire nello stack
iload3	MAR = SP = SP + 1	SP punta alla nuova cima dello stack; prepara la scrittura
iload4	PC = PC + 1; fetch; wr	Incrementa PC; prelievo del successivo codice operativo; scrive in cima allo stack
iload5	TOS = MDR	Aggiorna TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR contiene il codice operativo; prelievo del byte successivo; diramazione

Figura 4.25 Codice di Mic-1 per l'esecuzione di ILOAD.

Possiamo osservare che durante iload1 LV è copiato all'interno di H. Ciò avviene per l'unico motivo di poterlo successivamente sommare a MBRU durante iload2. Nella nostra architettura originale a due bus non esiste alcun modo per sommare direttamente due registri arbitrari; l'unica soluzione consiste nel copiarne inizialmente uno in H. Nella nuova architettura a tre bus possiamo invece risparmiare un ciclo, come si può vedere

nella Figura 4.26. In questo caso abbiamo sommato il ciclo d'interpretazione a ILOAD, ma così facendo non abbiamo né aumentato né diminuito la lunghezza del percorso di esecuzione.

Etichetta	Operazioni	Commenti
iload1	MAR = MBRU + LV; rd	MAR = indirizzo della variabile locale da inserire nello stack
iload2	MAR = SP = SP + 1	SP punta alla nuova cima dello stack; prepara la scrittura
iload3	PC = PC + 1; fetch; wr	Incrementa PC; prelievo del successivo codice operativo; scrive in cima allo stack
iload4	TOS = MDR	Aggiorna TOS
iload5	PC = PC + 1; fetch; goto (MBR)	MBR contiene già il codice operativo; prelievo del byte dell'indice

Figura 4.26 Codice dell'architettura a tre bus per l'esecuzione di ILOAD.

Tuttavia l'aggiunta del terzo bus ha ridotto il tempo totale di esecuzione di ILOAD, portandolo da sei a cinque cicli. A questo punto abbiamo a disposizione una seconda tecnica che ci permette di ridurre la lunghezza del percorso:

passare da un'architettura a due bus a una a tre bus.

Unità di prelievo dell'istruzione

Entrambe le tecniche precedenti sono valide, ma per ottenere un netto miglioramento delle prestazioni abbiamo bisogno di qualcosa di molto più radicale. Facciamo un passo indietro e analizziamo le parti comuni a tutte le istruzioni, ovvero il prelievo e la decodifica dei campi. Per ogni istruzione si possono verificare le seguenti operazioni:

1. il valore di PC viene fatto passare attraverso la ALU per incrementarlo;
2. il valore di PC è utilizzato per prelevare il byte successivo nel flusso delle istruzioni;
3. gli operandi sono letti dalla memoria;
4. gli operandi sono scritti in memoria;
5. la ALU esegue una computazione e i risultati vengono memorizzati.

Se un'istruzione contiene campi aggiuntivi (per gli operandi), questi vanno prelevati esplicitamente, un byte alla volta, e poi assemblati prima di poterli utilizzare. Prelevare e assemblare un operando impegnava la ALU per almeno un ciclo per byte, per incrementare PC e in seguito assemblare l'indice, o lo spiazzamento, risultante. Durante praticamente ogni ciclo la ALU viene utilizzata, oltre che per il "lavoro" vero e proprio dell'istruzione, anche per una grande varietà di operazioni relative al prelievo dell'istruzione e l'assemblaggio dei campi al suo interno.

Per poter sovrapporre il ciclo principale, è necessario liberare la ALU da alcuni di questi compiti. Ciò potrebbe essere fatto introducendo una seconda ALU, anche se per

molte operazioni non è realmente necessario disporre di tutte le funzionalità che fornisce una ALU completa. Possiamo notare infatti che in molti casi la ALU viene semplicemente usata come un cammino per copiare un valore da un registro a un altro. Questi cicli potrebbero essere eliminati introducendo percorsi dati aggiuntivi che non passino attraverso la ALU. Per esempio potrebbe essere vantaggioso creare un percorso da TOS a MDR, oppure da MDR a TOS, dato che la parola in cima allo stack viene scambiata frequentemente tra questi due registri.

In Mic-1 è possibile eliminare gran parte del carico di lavoro che grava sulla ALU creando un'unità indipendente che si occupa del prelevò e dell'elaborazione delle istruzioni. Questa unità, chiamata IFU (*Instruction Fetch Unit*, “unità di fetch delle istruzioni”), può incrementare il PC in modo indipendente e può prelevare i byte prima ancora che siano richiesti. Per implementare questa unità è sufficiente un incrementatore, cioè un circuito molto più semplice di un sommatore. Portando ancora più avanti l'idea di un'unità separata per la gestione delle istruzioni è possibile realizzare una IFU in grado di assemblare gli operandi a 8 e 16 bit, in modo che siano immediatamente pronti in qualsiasi momento vengano richiesti. Per poter far ciò esistono almeno due modi distinti.

1. La IFU può interpretare realmente ciascun codice operativo, determinando quanti campi aggiuntivi devono essere prelevati, e assemblarli in un registro pronto per essere utilizzato dall'unità di esecuzione principale.
2. La IFU può trarre vantaggio dal fatto che le istruzioni sono in realtà un flusso di byte e rendere disponibili, in ogni momento, le successive porzioni di 8 e 16 bit, anche se potrebbero non essere necessarie. Spetta poi all'unità di esecuzione principale richiedere i dati veramente necessari.

La Figura 4.27 mostra i principi fondamentali del secondo schema. Al posto di un solo registro MBR a 8 bit, ora ce ne sono due: MBR1 a 8 bit e MBR2 a 16 bit. La IFU tiene traccia del byte, o dei byte, utilizzati più recentemente dall'unità di esecuzione principale. Inoltre, allo stesso modo di quanto avveniva in Mic-1, rende disponibile all'interno di MBR1 il byte successivo; la differenza è che in questo caso l'unità, non appena rileva che MBR1 è stato letto, preleva il byte successivo e lo memorizza immediatamente all'interno di MBR1. Come in Mic-1 il registro ha due interfacce con il bus B: MBR1 e MBR1U. La prima è estesa col segno a 32 bit, mentre la seconda è estesa con bit zero.

In modo analogo, MBR2 fornisce le stesse funzionalità memorizzando i successivi 2 byte. Anche questo registro ha due interfacce con il bus B, MBR2 e MBR2U, che forniscono rispettivamente l'estensione con il segno a 32 bit e l'estensione con bit zero.

La IFU è responsabile del prelevio di un flusso di byte. Essa compie il proprio lavoro utilizzando una convenzionale porta di memoria a 4 byte, prelevando in anticipo intere parole a 4 byte e caricando i byte consecutivi all'interno di un registro a scorrimento. Questo registro ha la funzione di mantenere una coda di byte provenienti dalla memoria per alimentare MBR1 e MBR2, fornendo loro uno, o due byte, alla volta nell'ordine in cui sono stati prelevati.

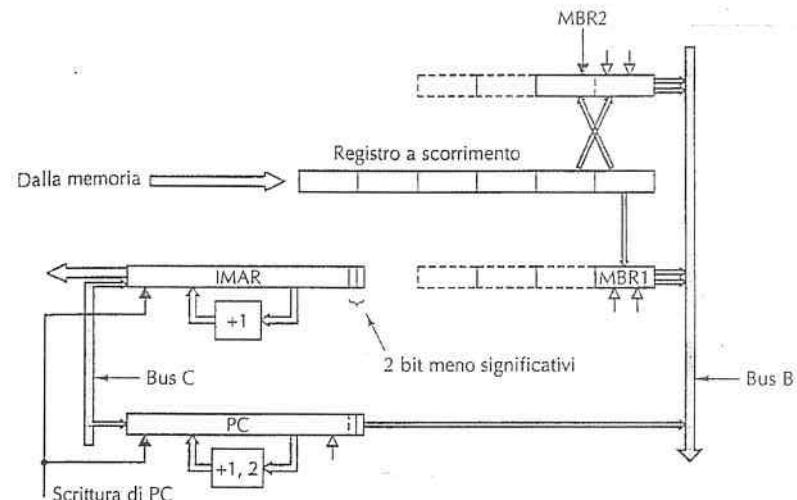


Figura 4.27 Unità di prelievo per Mic-1.

MBR1 contiene sempre il byte più vecchio del registro a scorrimento, mentre MBR2 contiene i 2 byte più vecchi (il più vecchio dei due è a sinistra), che servono a formare un intero a 16 bit, come mostrato nella Figura 4.19(b). I 2 byte di MBR2 possono anche provenire da parole diverse, dato che in memoria le istruzioni IJVM non sono allineate alle parole.

Ogni volta che viene letto MBR1, il registro a scorrimento viene traslato a destra di 1 byte, e di 2 quando viene letto MBR2. In seguito MBR1 e MBR2 vengono ricaricati rispettivamente con il byte o con la coppia di byte più vecchi. Se nella parte sinistra del registro a scorrimento c'è sufficiente spazio per contenere un'intera nuova parola, la IFU inizia un ciclo di memoria per leggerla. Assumiamo che, dopo la lettura di uno dei due registri MBR, questo venga nuovamente riempito all'inizio del ciclo successivo, in modo che sia possibile effettuare più letture durante cicli consecutivi.

Come mostra la Figura 4.28, è possibile modellare l'architettura della IFU mediante un **automa a stati finiti**. Questi automi sono costituiti da stati (rappresentati da cerchi) e da **transizioni** (rappresentate come archi che collegano uno stato con un altro). Ogni stato rappresenta una possibile situazione nella quale si può trovare l'automa. Il nostro automa ha sette stati, corrispondenti ai sette stati del registro a scorrimento della Figura 4.27. Gli stati sono identificati dagli interi da 0 a 6, che indicano il numero di byte presenti in un dato momento nel registro.

Un arco rappresenta un evento che si può verificare. Nel nostro caso ci sono tre possibili eventi. Il primo è la lettura di un byte da MBR1, e quando si verifica il registro a scorrimento viene attivato e trasla a destra per estrarre 1 byte, riducendo dunque lo stato di 1. Il secondo evento è una lettura di 2 byte da MBR2, che riduce lo stato di 2. Entram-

be queste transizioni fanno sì che MBR1 e MBR2 vengano ricaricati nuovamente. Quando l'automa si porta negli stati 0, 1 o 2, ha inizio un riferimento alla memoria per prelevare una nuova parola (assumendo che la memoria non sia già occupata nella lettura di una parola). Quando termina la lettura della parola l'automa avanza di 4 stati.

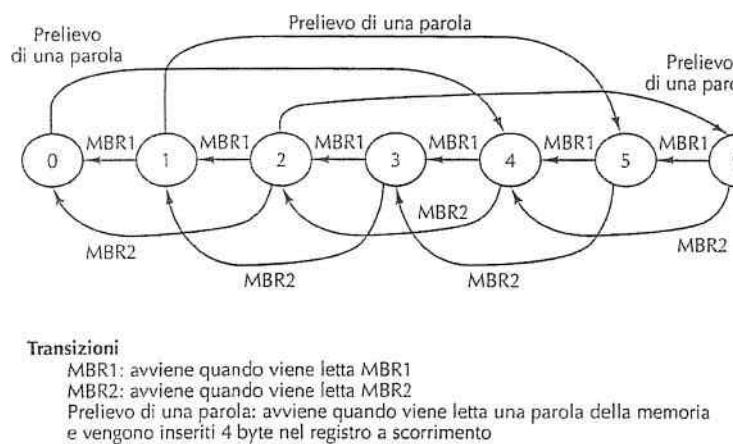


Figura 4.28 Automa a stati finiti che implementa la IFU.

Per un corretto funzionamento, l'automa deve bloccarsi quando gli viene fatta una richiesta che non è in grado di soddisfare, per esempio fornire il valore di MBR2 quando all'interno del registro c'è un solo byte e la memoria è ancora occupata nel prelevio della nuova parola. Inoltre l'automa può eseguire solamente un'azione alla volta e occorre quindi serializzare gli eventi in entrata. Infine, l'automa deve essere aggiornato ogni volta che PC viene modificato. Questi dettagli rendono l'automa più complicato di quanto abbiamo mostrato precedentemente; tuttavia molti dispositivi hardware sono progettati come automi a stati finiti.

La IFU possiede un proprio registro per l'indirizzo di memoria, chiamato IMAR, utilizzato per puntare all'indirizzo di memoria in cui si trova la parola da prelevare. Questo registro possiede un proprio incrementatore e non occorre quindi utilizzare la ALU per aggiornarlo (affinché si riferisca alla parola successiva). La IFU deve monitorare il bus C per rilevare quando il PC viene caricato e copiare di conseguenza il nuovo valore in IMAR. Dato che il nuovo valore del PC potrebbe non trovarsi sul limite di una parola, la IFU deve prelevare la parola corretta e regolare in modo appropriato il registro di scorrimento.

Utilizzando la IFU, il PC viene modificato dalla ALU solamente nel caso in cui sia necessario cambiare la natura sequenziale del flusso di byte delle istruzioni. Ciò si verifica quando viene soddisfatta la condizione di una diramazione e nel caso delle istruzioni INVOKEVIRTUAL e IRETURN.

È compito della IFU tenere aggiornato il PC, dato che il microprogramma non lo incrementa più in modo esplicito quando vengono prelevati i codici operativi. Per sapere quando occorre incrementare il PC, la IFU rileva quando viene consumato un byte del flusso dell'istruzione, cioè quando viene letto MBR1, MBR2, o le loro versioni senza segno. A seconda di quanti byte sono stati consumati un incrementatore aumenta il valore del PC di 1 oppure di 2. In questo modo il PC contiene sempre l'indirizzo del primo byte non ancora consumato. All'inizio di ogni istruzione, MBR1 contiene il codice operativo dell'istruzione stessa.

Si noti la presenza di due incrementatori, con funzioni distinte. PC conta i *byte* e viene incrementato di 1 o di 2; IMAR conta invece le *parole* e viene incrementato soltanto di 1 (per 4 nuovi byte). Analogamente a MAR, anche IMAR è collegato al bus dell'indirizzo in modo disallineato, con il bit 0 di IMAR connesso alla linea d'indirizzo 2, e così via, in modo da convertire implicitamente gli indirizzi di parole in indirizzi di byte.

Come vedremo tra poco in modo più dettagliato, il fatto di non dover incrementare PC nel ciclo principale rappresenta un grande vantaggio. Spesso infatti la microistruzione nella quale viene incrementato il PC compie poco lavoro aggiuntivo; se si eliminano queste istruzioni si riduce quindi il percorso di esecuzione. In questo caso il compromesso sta nell'aumento del numero di componenti hardware per disporre di una macchina più veloce. La nostra terza tecnica per ridurre la lunghezza del percorso prevede quindi

che le istruzioni siano prelevate dalla memoria da un'unità funzionale specializzata.

4.4.3 Architettura con prefetching: Mic-2

La IFU permette di ridurre in modo considerevole la lunghezza media del percorso di un'istruzione. In primo luogo elimina interamente il ciclo principale, dato che alla fine di ogni istruzione si effettua un semplice salto all'istruzione successiva. Inoltre risparmia l'utilizzo della ALU per l'incremento del PC, e riduce la lunghezza del percorso ogni volta che viene calcolato un indice o uno spiazzamento a 16 bit (dato che assembla il valore a 16 bit e lo fornisce direttamente alla ALU come un valore a 32 bit, evitando di doverlo assemblare all'interno di H). La Figura 4.29 mostra Mic-2, cioè una versione migliorata di Mic-1 in cui è stata aggiunta la IFU della Figura 4.27. La Figura 4.30 mostra il microcodice per la nuova macchina.

Per vedere un esempio del funzionamento di Mic-2 consideriamo IADD. La macchina preleva la seconda parola dello stack ed esegue l'addizione come prima, tranne per il fatto che, una volta terminata l'operazione, non è più necessario tornare a Main1 per incrementare PC e passare alla microistruzione successiva. Quando la IFU vede che durante iadd3 è stato effettuato un riferimento a MBR1, il suo registro a scorrimento trasla tutto il proprio contenuto a destra e ricarica sia MBR1 sia MBR2. L'unità effettua quindi una transizione verso uno stato che è inferiore di uno rispetto a quello corrente e, se il nuovo stato è 2, la IFU inizia a prelevare una nuova parola dalla memoria. Tutto ciò viene effettuato dall'hardware, senza alcun intervento del microcodice. Questo è il motivo che permette di ridurre IADD da quattro a tre microistruzioni.

Mic-2 permette di ottenere miglioramenti più marcati per alcune istruzioni piuttosto che altre. LDC_W passa da nove a tre microistruzioni; SWAP invece passa soltanto da otto

a sei microistruzioni. Per migliorare le prestazioni globali del sistema ciò che conta veramente è il miglioramento che si può ottenere con le istruzioni usate più frequentemente. Le istruzioni più comuni sono ILOAD (da 6 a 3), IADD (da 4 a 3) e IF_ICMPEQ (da 13 a 10 quando la condizione è verificata; da 10 a 8 altrimenti). Anche se per misurare il miglioramento delle prestazioni bisognerebbe eseguire dei test, è tuttavia evidente che il guadagno sia significativo.

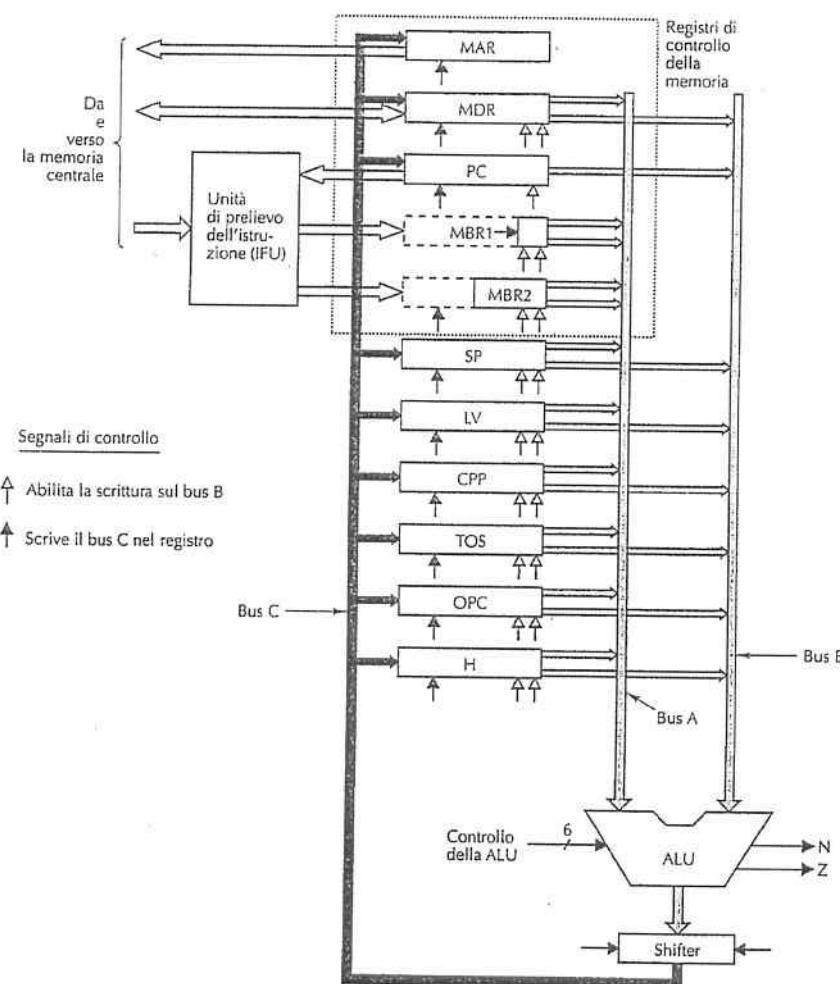


Figura 4.29 Percorso dati di Mic-2.

Etichetta	Operazioni	Commenti
nop1	goto (MBR)	Salto all'istruzione successiva
iadd1	MAR = SP = SP - 1; rd H = TOS MDR = TOS = MDR + H; wr; goto (MBR1)	Legge la parola sotto la cima dello stack; H = cima dello stack Somma le due parole in cima allo stack; scrive in cima allo stack
iadd2		
iadd3		
isub1	MAR = SP = SP - 1; rd H = TOS MDR = TOS = MDR - H; wr; goto (MBR1)	Legge la parola sotto la cima dello stack H = cima dello stack Sottrae TOS da TOS Prelevato-1
isub2		
isub3		
iand1	MAR = SP = SP - 1; rd H = TOS MDR = TOS = MDR AND H; wr; goto (MBR1)	Legge la parola sotto la cima dello stack H = cima dello stack AND tra TOS Prelevato-1 e TOS
iand2		
iand3		
ior1	MAR = SP = SP - 1; rd H = TOS MDR = TOS = MDR OR H; wr; goto (MBR1)	Legge la parola sotto la cima dello stack H = cima dello stack OR tra TOS Prelevato-1 e TOS
ior2		
ior3		
dup1	MAR = SP = SP + 1 MDR = TOS; wr; goto (MBR1)	Incrementa SP e lo copia in MAR Scrive la nuova parola sullo stack
dup2		
pop1	MAR = SP = SP - 1; rd	Legge la parola sotto la cima dello stack
pop2		Attende che si completi la lettura
pop3	TOS = MDR; goto (MBR1)	Copia la nuova parola in TOS
swap1	MAR = SP - 1; rd	Legge la parola sotto la cima dello stack; imposta MAR a SP
swap2	MAR = SP	Prepara la scrittura della nuova seconda parola
swap3	H = MDR; wr	Salva il nuovo TOS; scrive la seconda parola nello stack
swap4	MDR = TOS	Copia il vecchio TOS in MDR
swap5	MAR = SP - 1; wr	Scrive il vecchio TOS nella seconda posizione dello stack
swap6	TOS = H; goto (MBR1)	Aggiorna TOS
bipush1	SP = MAR = SP + 1	Imposta MAR per scrivere la nuova cima dello stack
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Aggiorna lo stack in TOS e nella memoria
iload1	MAR = LV + MBR1U; rd	Imposta MAR a LV + indice per la lettura
iload2	MAR = SP = SP + 1	SP punta alla nuova cima dello stack; prepara la scrittura
iload3	TOS = MDR; wr; goto (MBR1)	Aggiorna lo stack in TOS e nella memoria
istore1	MAR = LV + MBR1U	Imposta MAR a LV + indice
istore2	MDR = TOS; wr	Copia TOS per la scrittura in memoria
istore3	MAR = SP = SP - 1; rd	Decrementa SP, legge il nuovo TOS
istore4		Attende che si completi la lettura
istore5	TOS = MDR; goto (MBR1)	Aggiorna TOS
wide1	goto (MBR1 OR 0x100)	L'indirizzo successivo è il risultato dell'OR tra 0x100 e il codice operativo
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Identico a iload1, ma con un indice a 2 byte
wide_istore1	MAR = LV + MBR2U; goto istore2	Identico a istore1, ma con un indice a 2 byte
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	Uguale a wide_iload1, ma indicizza a partire da CPP

Figura 4.30 Microprogramma per Mic-2 (continua).

iinc1	$MAR = LV + MBR1U; rd$	Imposta MAR a $LV +$ indice per la lettura
iinc2	$H = MBR1$	Imposta H a una costante
iinc3	$MDR = MDR + H; wr;$ goto (MBR1)	Incrementa di una costante e aggiorna
goto1	$H = PC - 1$	Copia PC in H
goto2	$PC = H + MBR2$	Aggiunge uno spiazzamento e aggiorna PC
goto3	goto (MBR1)	Deve attendere che la IFU prelevi il nuovo codice operativo
goto4		Salto alla nuova istruzione
iflt1	$MAR = SP = SP - 1; rd$	Legge la parola sotto la cima dello stack
iflt2	$OPC = TOS$	Salva temporaneamente TOS in OPC
iflt3	$TOS = MDR$	Inserisce in TOS la nuova cima dello stack
iflt4	$N = OPC; if (N) goto T;$ else goto F	Diramazione in base al bit N
ifeq1	$MAR = SP = SP - 1; rd$	Legge la parola sotto la cima dello stack
ifeq2	$OPC = TOS$	Salva temporaneamente TOS in OPC
ifeq3	$TOS = MDR$	Inserisce in TOS la nuova cima dello stack
ifeq4	$Z = OPC; if (Z) goto T;$ else goto F	Diramazione in base al bit Z
if_icmpeq1	$MAR = SP = SP - 1; rd$	Legge la parola sotto la cima dello stack
if_icmpeq2	$MAR = SP = SP - 1$	Imposta MAR per leggere la nuova cima dello stack
if_icmpeq3	$H = MDR; rd$	Copia in H la parola sotto la cima dello stack
if_icmpeq4	$OPC = TOS$	Salva temporaneamente TOS in OPC
if_icmpeq5	$TOS = MDR$	Inserisce in TOS la nuova cima dello stack
if_icmpeq6	$Z = H - OPC; if (Z) goto T;$ else goto F	Se le due parole in cima allo stack sono uguali, goto T, altrimenti, goto F
T	$H = PC - 1; goto goto2$	Uguale goto1
F	$H = MBR2$	Aggiorna i byte in MBR2 per scartarne il contenuto
F2	goto (MBR1)	
invokevirtual1	$MAR = CPP + MBR2U; rd$	Inserisce in MAR l'indirizzo del puntatore al metodo
invokevirtual2	$OPC = PC$	Salva il PC di ritorno in OPC
invokevirtual3	$PC = MDR$	Imposta PC al primo byte del codice del metodo
invokevirtual4	$TOS = SP - MBR2U$	$TOS =$ indirizzo di OBJREF - 1
invokevirtual5	$TOS = MAR = H = TOS + 1$	$TOS =$ indirizzo di OBJREF
invokevirtual6	$MDR = SP + MBR2U + 1; wr$	Sovrascrive OBJREF con il puntatore di collegamento
invokevirtual7	$MAR = SP = MDR$	Imposta SP e MAR alla locazione in cui memorizzare il vecchio PC
invokevirtual8	$MDR = OPC; wr$	Si prepara a salvare il vecchio PC
invokevirtual9	$MAR = SP = SP + 1$	Incrementa SP per puntare alla locazione in cui memorizzare il vecchio LV
invokevirtual10	$MDR = LV; wr$	Salva il vecchio LV
invokevirtual11	$LV = TOS; goto (MBR1)$	Imposta LV per puntare al parametro Ø
ireturn1	$MAR = SP = LV; rd$	Reimposta SP e MAR per leggere il puntatore di collegamento
ireturn2	$LV = MAR = MDR; rd$	Attende il puntatore di collegamento
ireturn3	$MAR = LV + 1$	Imposta LV e MAR al puntatore di collegamento; legge il vecchio PC
ireturn4	$PC = MDR; rd$	Imposta MAR per puntare al vecchio LV; legge il vecchio LV
ireturn5	$MAR = SP$	Ripristina PC
ireturn6	$LV = MDR$	Ripristina LV
ireturn7	$MDR = TOS; wr; goto (MBR1)$	Salva il valore di ritorno sulla cima originale dello stack
ireturn8		

Figura 4.30 Micropogramma per Mic-2.

4.4.4 Architettura a pipeline: Mic-3

Mic-2 rappresenta ovviamente un miglioramento rispetto a Mic-1: è più veloce e utilizza una minor quantità di memoria di controllo, anche se in termini di "proprietà terriera" il costo della IFU è senza dubbio maggiore del guadagno ottenuto riducendo la memoria di controllo. Mic-2 è dunque una macchina considerevolmente più veloce con un prezzo leggermente maggiore.

Che cosa si può dire riguardo al tentativo di diminuire il tempo del ciclo? In buona parte il tempo del ciclo è determinato dalla tecnologia impiegata. Più piccoli sono i transistor e minore è la distanza che li separa, maggiore è la velocità alla quale può funzionare il clock. Per una data tecnologia il tempo richiesto per eseguire un'operazione sull'intero percorso dati è fisso (almeno dal nostro punto di vista). Ciononostante esistono alcuni spazi di libertà che tra poco cercheremo di sfruttare pienamente.

L'alternativa che abbiamo è quella di introdurre nella macchina un maggior parallelismo. Ora come ora Mic-2 funziona in modo altamente sequenziale; copia i registri sul bus, aspetta che la ALU e lo shifter li processino e infine scrive i risultati nuovamente all'interno dei registri. Fatta eccezione per la IFU, in questo funzionamento vi è poco parallelismo; vale la pena provare ad aumentarlo.

Com'è già stato detto, il ciclo del clock è limitato dal tempo necessario affinché i segnali si possano propagare lungo il percorso dati. La Figura 4.3 mostra schematicamente quali sono, durante ciascun ciclo, i ritardi attraverso i vari componenti. Il ciclo del percorso dati è composto da tre componenti principali:

1. il tempo necessario per portare i registri selezionati sui bus A e B;
 2. il tempo impiegato dalla ALU e dallo shifter per compiere il proprio lavoro;
 3. il tempo necessario per riportare i risultati nei registri e memorizzarli al loro interno.
- Nella Figura 4.31 si vede una nuova architettura a tre bus che, oltre alla IFU, presenta anche tre *latch* (registri) inseriti a metà di ciascun bus. Questi registri vengono scritti a ogni ciclo. In effetti i latch spezzano il percorso dati in parti distinte che possono funzionare indipendentemente l'una dall'altra. Per riferirci a questa architettura utilizzeremo il nome **Mic-3**, oppure modello a pipeline.

Come possono essere d'aiuto questi registri? Con la nuova architettura occorrono ora tre cicli di clock per utilizzare il percorso dati: uno per caricare i latch A e B, uno per far funzionare la ALU e lo shifter e per caricare il latch C, e uno per memorizzare il latch nei registri. Questo è sicuramente peggiore di quello che avevamo già. Siamo pazzi? Ovviamente no! Infatti il vantaggio dato dall'inserimento dei latch è duplice:

1. possiamo accelerare il clock dato che il ritardo massimo è ora più corto;
2. possiamo usare tutte le parti del percorso dati durante ogni ciclo.

Spezzando il percorso dati in tre parti il ritardo massimo si riduce, con il risultato che la frequenza del clock può essere maggiore. Supponendo di dividere il ciclo del percorso dati in tre intervalli di tempo, lunghi circa 1/3 di quello originario, possiamo triplicare la velocità del clock. La cosa non è proprio realistica, dato che abbiamo anche aggiunto due registri all'interno del percorso dati, ma come prima approssimazione può essere accettabile.

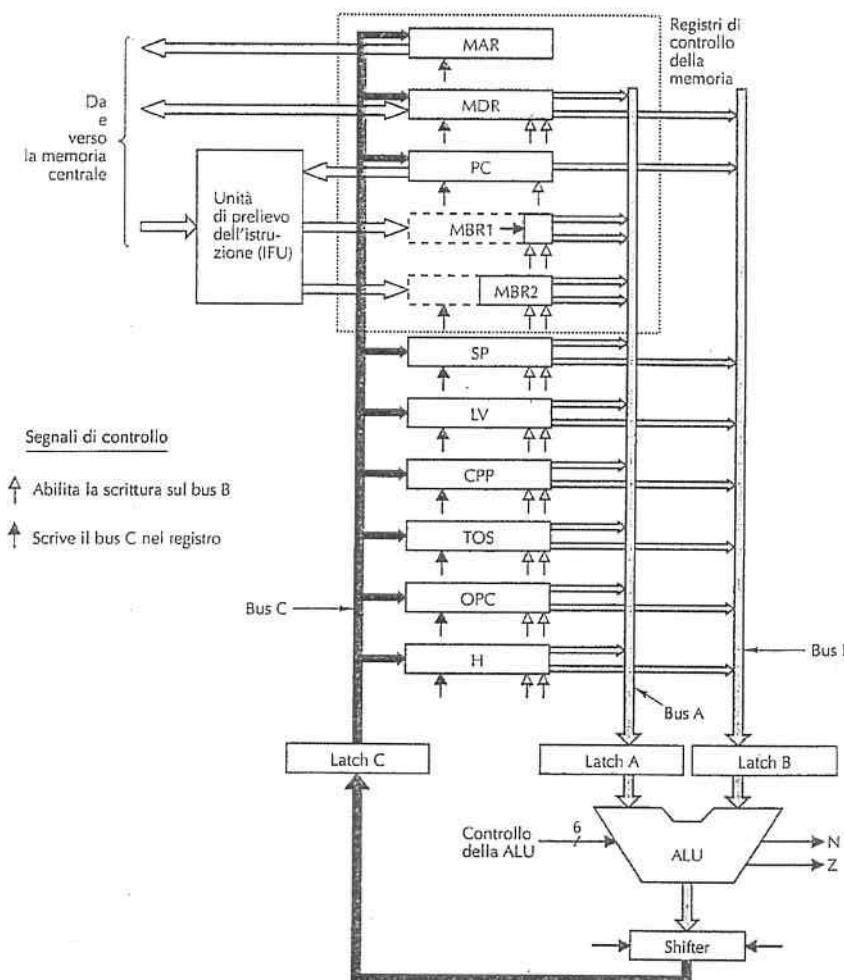


Figura 4.31 Percorso dati a tre bus utilizzato in Mic-3.

Dato che abbiamo assunto che letture e scritture di memoria possono essere soddisfatte dalla cache di primo livello e dato che questa cache è realizzata con lo stesso materiale dei registri, continueremo ad assumere che un'operazione di memoria richieda un solo ciclo (in pratica però ciò potrebbe non essere così semplice).

Il secondo punto riguarda il throughput piuttosto che la velocità delle singole istruzioni. In Mic-2 la ALU resta inattiva durante la prima e la terza parte di ogni ciclo di

clock, ma spezzando il percorso dati in tre parti saremo in grado di utilizzare la ALU in ogni ciclo, ottenendo così una quantità di lavoro tre volte maggiore.

Poniamo ora la nostra attenzione sul percorso dati di Mic-3. Prima di iniziare ci serve però una notazione per i latch. La scelta più ovvia è quella di chiamarli A, B e C e di trattarli come dei registri, tenendo ben presenti i vincoli del percorso dati. La Figura 4.32 mostra la sequenza di codice relativa all'implementazione di SWAP per Mic-2.

Etichetta	Operazioni	Commenti
swap1	MAR = SP - 1; rd	Legge la parola sotto la cima dello stack; imposta MAR a SP
swap2	MAR = SP	Si prepara a scrivere la nuova seconda parola
swap3	H = MDR; wr	Salva il nuovo TOS; scrive la seconda parola nello stack
swap4	MDR = TOS	Copia il vecchio TOS in MDR
swap5	MAR = SP - 1; wr	Scrive il vecchio TOS nella seconda posizione dello stack
swap6	TOS = H; goto (MBR1)	Aggiorna TOS

Figura 4.32 Codice di Mic-2 per SWAP.

Proviamo ora a implementare questa stessa sequenza sulla macchina Mic-3. Ricordiamoci che il percorso dati comprende ora tre cicli: uno per caricare A e B, uno per eseguire l'operazione e caricare C e uno per scrivere i risultati nei registri. Chiameremo passo elementare ciascuna di queste parti. L'implementazione di SWAP per Mic-3 è data nella Figura 4.33. Nel ciclo 1 iniziamo con swap1 copiando SP in B. Non importa quale valore viene memorizzato in A, dato che per sottrarre 1 da B bisogna negare ENA (Figura 4.2). Per semplicità non mostreremo gli assegnamenti che non sono utilizzati. Nel ciclo 2 effettuiamo la sottrazione. Nel ciclo 3 il risultato viene memorizzato in MAR e alla fine del ciclo inizia l'operazione di lettura (dopo che MAR è stato memorizzato). La lettura dalla memoria richiede un solo ciclo, e non sarà completata se non alla fine del ciclo 4; ciò è indicato dall'assegnamento a MDR durante il ciclo 4. Il valore in MDR non può essere letto prima del ciclo 5.

Tornando al ciclo 2, osserviamo che si può dapprima copiare SP in B per farlo passare all'interno della ALU nel ciclo 3 e infine memorizzarlo all'interno di MAR nel ciclo 4. Fin qui, nessun problema. Dovrebbe essere chiaro che se riusciamo a mantenere questo ritmo, iniziando una nuova microistruzione a ogni ciclo, potremo triplicare la velocità della macchina. Questo guadagno deriva dal fatto che possiamo lanciare a ogni ciclo una nuova microistruzione e che Mic-3 ha, rispetto a Mic-2, il triplo di cicli di clock per secondo. Quella che in realtà abbiamo creato è una CPU a pipeline.

Purtroppo nel ciclo 3 sorge un intoppo. Vorremmo cominciare a occuparci di swap3, ma la sua prima operazione consiste nel far passare MDR attraverso la ALU e MDR sarà disponibile dalla memoria solamente all'inizio del ciclo 5. La situazione in cui un passo elementare non può iniziare, dato che è in attesa di un risultato che non è ancora stato prodotto da un precedente passo elementare, viene chiamata dipendenza RAW (o dipendenza effettiva). Spesso ci si riferisce a queste situazioni con il termine *hazard*,

cioè “rischio”. RAW è l’acronimo di *Read After Write* (“lettura dopo scrittura”) e indica che un passo elementare vuole leggere un registro che non è ancora stato scritto. In questo caso l’unica cosa sensata da fare è ritardare l’inizio di swap3 finché MDR non sia disponibile, nel ciclo 5. Siamo cioè in una situazione di stallo. Una volta ottenuto il valore richiesto possiamo ricominciare a far partire una microistruzione per ciclo, dato che non ci sono più dipendenze. La microistruzione swap6 sfiora una dipendenza, in quanto legge H nel ciclo immediatamente successivo a quello in cui swap3 la scrive; se swap5 avesse provato a leggere H, sarebbe rimasta in stallo per un ciclo.

	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6
C	MAR=SP-1; rd	MAR=SP	H=MDR; wr	MDR=TOS	MAR=SP-1; wr	TOS=H; goto (MBR1)
1	B=SP					
2	C=B-1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=Mem	MAR=C				
5		B=MDR				
6		C=B	B=TOS			
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B-1	B=H
9					MAR=C; wr	C=B
10					Mem=MDR	TOS=C
11						goto (MBR1)

Figura 4.33 Implementazione di SWAP su Mic-3.

Il programma di Mic-3 richiede più cicli di quello di Mic-2, ma viene eseguito più velocemente. Se indichiamo con ΔT ns il tempo di ciclo di Mic-3, allora Mic-3 impiega $11\Delta T$ ns per eseguire SWAP. Mic-2 impiega invece 6 cicli, ciascuno dei quali è lungo $3\Delta T$, per un totale di $18\Delta T$. L’uso della pipeline ha reso la macchina più veloce, anche se è necessario rimanere in stallo per risolvere una dipendenza.

Dato che l’uso della pipeline è una tecnica chiave in tutte le CPU moderne è importante comprenderne a fondo il funzionamento. Nella Figura 4.34 vediamo il percorso dati della Figura 4.31 reso graficamente come una pipeline. La prima colonna rappresenta ciò che avviene durante il primo ciclo, la seconda colonna ciò che avviene nel secondo, e così via (assumendo che non si verifichino stalli). La regione in grigio dell’istruzione 1 durante il ciclo 1 indica che la IFU è occupata nel prelievo dell’istruzione 1. Al successivo scoccare del clock, durante il ciclo 2, i registri richiesti dall’istruzione 1 vengono caricati nei latch A e B, mentre la IFU è occupata nel prelievo dell’istruzione 2; anche in questo caso le operazioni sono indicate da rettangoli grigi.

Durante il ciclo 3 l’istruzione 1 utilizza la ALU e lo shifter per eseguire la propria operazione, mentre i latch A e B vengono caricati per l’istruzione 2 e, allo stesso tempo, l’istruzione 3 comincia a essere prelevata. Infine, durante il ciclo 4, ci sono quattro

istruzioni in corso d’esecuzione. I risultati dell’istruzione 1 cominciano a essere memorizzati, il lavoro della ALU per l’istruzione 2 volge al termine, i latch A e B vengono caricati per l’istruzione 3 e si comincia a prelevare l’istruzione 4.

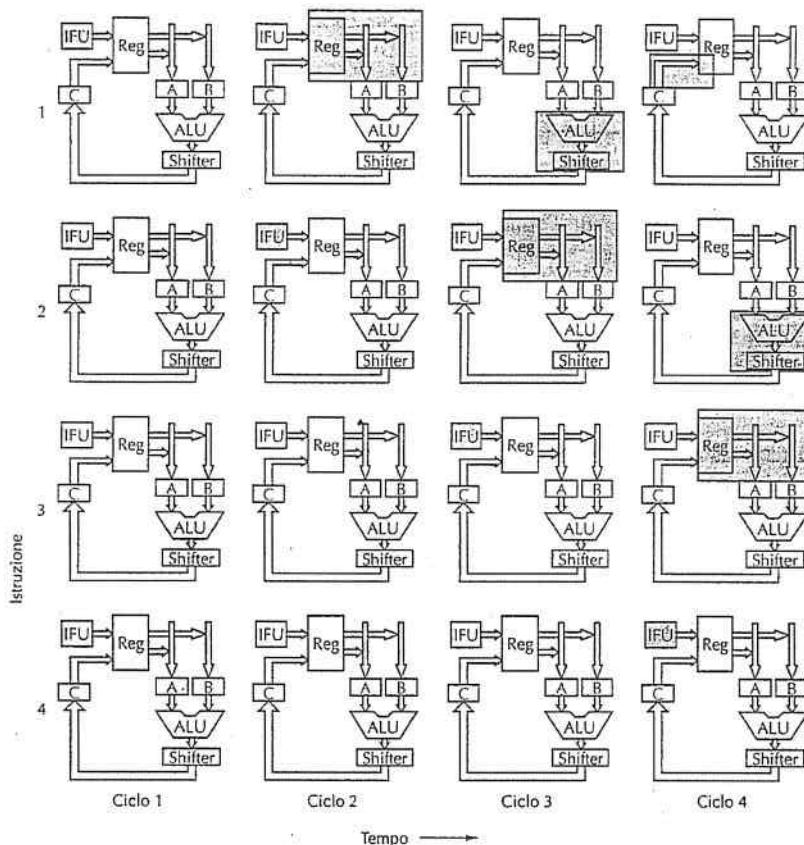


Figura 4.34 Rappresentazione grafica del funzionamento di una pipeline.

Se avessimo mostrato il ciclo 5 e quelli successivi la situazione sarebbe stata la stessa del ciclo 4: tutte e quattro le parti del percorso dati che possono funzionare indipendentemente continuano infatti a lavorare in parallelo. Questa organizzazione rappresenta una pipeline a 4 stadi, in cui uno preleva le istruzioni, uno accede agli operandi, uno esegue le operazioni della ALU e uno scrive i risultati nei registri. La pipeline è simile a quella della Figura 2.4(a), con la differenza che manca lo stadio di decodifica. Nell’uso della pipeline l’aspetto più importante da cogliere è che, anche se una singola istru-

zione richiede quattro cicli di clock per essere eseguita, durante ogni ciclo di clock può iniziare una nuova istruzione e mentre una, iniziata precedente, termina la propria esecuzione.

La Figura 4.34 può essere guardata in un altro modo, seguendo orizzontalmente ciascuna istruzione lungo tutta la larghezza della pagina. Per quanto riguarda l'istruzione 1, durante il ciclo 1 la IFU è in funzione. Nel ciclo 2 i suoi registri cominciano a essere portati sui bus A e B. Nel ciclo 3 la ALU e lo shifter svolgono le operazioni da lei richieste. Infine, nel ciclo 4, i suoi risultati sono memorizzati nuovamente all'interno dei registri. Occorre notare che sono disponibili quattro sezioni dell'hardware e durante ciascun ciclo una data istruzione ne utilizza uno soltanto, lasciando le restanti sezioni a disposizione delle altre istruzioni.

Un'analogia utile a capire come sono organizzate le pipeline è la linea di montaggio di una fabbrica di automobili. Per astrarre gli aspetti essenziali del modello immaginiamo che a ogni minuto venga colpito un grande gong e che in quel momento tutte le macchine si spostino, lungo la linea di montaggio, da un'isola a quella successiva. In ogni stazione i lavoratori eseguono una particolare operazione sull'automobile che hanno di fronte, per esempio aggiungere il volante o installare i freni. A ciascun battito del gong (1 ciclo) viene inserita una nuova automobile nella linea di montaggio e un'altra termina di essere assemblata. Quindi a ogni ciclo viene completata una macchina, anche se l'intero assemblaggio potrebbe constare di centinaia di cicli. La fabbrica può produrre un'automobile al minuto indipendentemente da quanto tempo è effettivamente necessario per assemblare un'automobile. Questa è la potenza della pipeline, e può essere sfruttata tanto per le CPU quanto per le fabbriche di automobili.

4.4.5 Pipeline a sette stadi: Mic-4

Un aspetto sul quale abbiamo intenzionalmente sorvolato riguarda il fatto che ciascuna microistruzione seleziona il proprio successore. La maggior parte di loro seleziona semplicemente quella che la segue nella sequenza corrente, mentre l'ultima, come swap6, spesso esegue una diramazione che blocca la pipeline, dato che è impossibile continuare a prelevare nuove microistruzioni. Occorre trovare un modo migliore per gestire questo punto.

La nostra successiva (e ultima) microarchitettura è Mic-4. Le sue parti principali sono mostrate nella Figura 4.35, nella quale sono stati trascurati molti dettagli per rendere più chiara l'illustrazione. Come per Mic-3, anche questa microarchitettura è dotata di una IFU che preleva dalla memoria le parole e mantiene i vari registri MBR.

La IFU alimenta inoltre con il flusso di byte una nuova componente, chiamata unità di decodifica. Questa unità è dotata di una ROM interna indicizzata attraverso il codice operativo IJVM. Ciascuna riga contiene due parti: la lunghezza dell'istruzione IJVM corrispondente e un indice relativo a un'altra ROM: la ROM delle micro-operazioni. La lunghezza dell'istruzione IJVM è utilizzata per permettere all'unità di decodifica di analizzare il flusso di byte entrante e suddividerlo in istruzioni, sapendo in ogni momento quali byte rappresentano i codici operativi e quali invece gli operandi. Se l'istruzione corrente è lunga 1 byte (per esempio, POP), allora l'unità di decodifica sa che il byte successivo è un codice operativo. Se invece l'istruzione corrente è lunga 2 byte, l'unità

di decodifica sa che il byte successivo è un operando, seguito da un altro codice operativo. Quando incontra il prefisso WIDE, l'unità trasforma il byte successivo in uno speciale codice operativo più ampio (per esempio WIDE + ILOAD diventa WIDE_ILOAD).

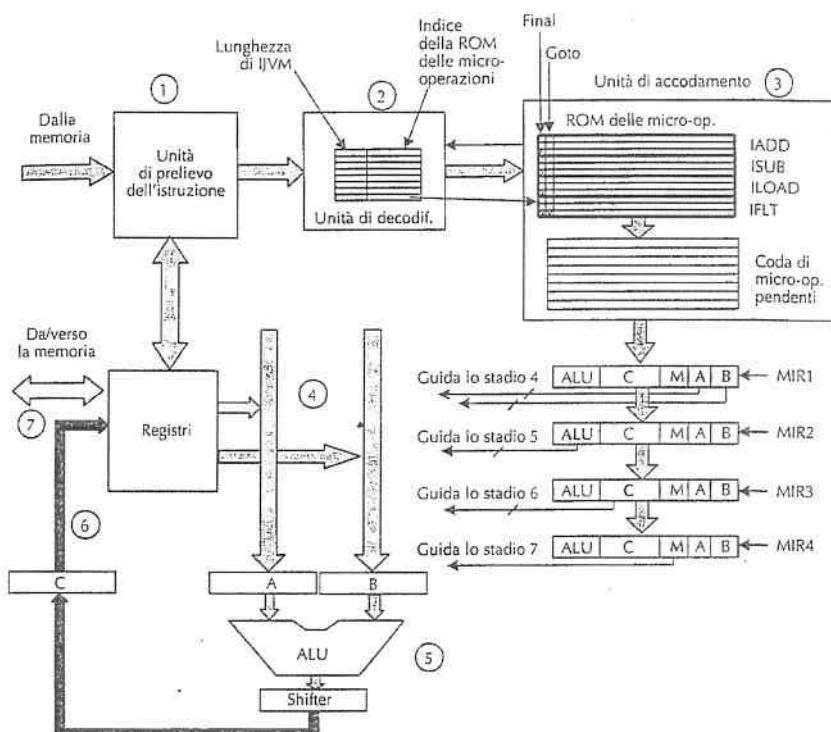


Figura 4.35 Componenti principali di Mic-4.

L'unità di decodifica invia alla componente successiva, chiamata unità di accodamento, l'indice relativo alla ROM delle micro-operazioni che ha trovato nella sua tabella. Oltre a vari circuiti, quest'unità contiene due tabelle interne, una in una RAM e l'altra in una ROM. Quest'ultima tabella contiene il microprogramma, in cui ciascuna istruzione IJVM ha un certo numero di elementi consecutivi, chiamati micro-operazioni. Queste devono essere in ordine e quindi non è possibile usare espedienti come quello che in Mic-2 permette a wide_iload2 di effettuare un salto verso iload2. Ogni sequenza IJVM deve essere scandita interamente. Per tale motivo, alcune sequenze devono essere duplicate.

Le micro-operazioni sono simili alle microistruzioni della Figura 4.5, tranne per il fatto che sono prive dei campi NEXT_ADDRESS e JAM, ma hanno un nuovo campo per

specificare l'input del bus A. Sono anche presenti due nuovi bit: Final e Goto. Il bit Final viene impostato a 1 nell'ultima micro-operazione IJVM di ciascuna sequenza per indicarne la fine. Il bit Goto viene invece impostato a 1 per indicare le micro-operazioni che effettuano microdiramazioni condizionali; il loro formato è diverso da quello delle normali micro-operazioni ed è costituito dai bit JAM e da un indice relativo alla ROM delle micro-operazioni. Le microistruzioni, che precedentemente eseguivano delle operazioni con il percorso dati ed effettuavano allo stesso tempo una microdiramazione condizionale (per esempio, `iflt4`), devono ora essere separate in due micro-operazioni distinte.

L'unità di accodamento funziona nel modo seguente. Riceve dall'unità di decodifica un indice della ROM delle micro-operazioni, cerca la micro-operazione corrispondente e la copia in una coda interna. Successivamente copia nella coda anche le due micro-operazioni seguenti. L'unità continua in questo modo finché non incontra una micro-operazione in cui il bit Final vale 1; in questo caso la copia e si arresta. L'unità di accodamento, assumendo che non abbia incontrato una micro-operazione con il bit Goto attivo e che ci sia ancora molto spazio nella coda, rispedisce all'unità di decodifica un segnale di conferma. Quando vede la conferma, l'unità di decodifica spedisce all'unità di accodamento l'indice della successiva istruzione IJVM.

In questo modo la sequenza d'istruzioni IJVM in memoria viene convertita in una sequenza di micro-operazioni in una coda. Queste micro-operazioni alimentano i registri MIR, che spediscono i segnali che permettono di controllare il percorso dati. Esiste tuttavia un altro fattore che dobbiamo considerare: i campi delle micro-operazioni non sono attivi nello stesso momento. I campi A e B sono attivi durante il secondo ciclo, il campo C lo è durante il terzo ciclo e ogni operazione di memoria si svolge nel quarto ciclo.

Per permettere che ciò funzioni correttamente abbiamo introdotto nella Figura 4.35 quattro registri MIR indipendenti. All'inizio di ciascun ciclo di clock (il tempo Δw nella Figura 4.3) MIR3 viene copiato in MIR4, MIR2 in MIR3, MIR1 in MIR2 e MIR1 viene caricato con una nuova micro-operazione proveniente dalla coda delle micro-operazioni. Ogni MIR emette i propri segnali, ma non tutti vengono utilizzati. I campi A e B provenienti da MIR1 selezionano i registri che guidano i latch A e B, mentre il campo ALU in MIR1 non viene utilizzato e non è connesso a nessuna componente del percorso dati.

Dopo un ciclo di clock, questa micro-operazione si è spostata in MIR2 e i registri che ha selezionato sono stati salvati nei latch A e B, in attesa che l'avventura cominci. Il suo campo ALU viene ora utilizzato per guidare la ALU. Nel ciclo successivo il suo campo C riscriverà il risultato nei registri e poi si sposterà in MIR4 e darà inizio all'operazione di memoria richiesta utilizzando MAR (o MDR, nel caso di una scrittura), che sarà già stato caricato.

C'è un ultimo aspetto di Mic-4 che occorre approfondire: le microdiramazioni. Alcune istruzioni IJVM, come `IFLT`, richiedono di effettuare dei salti condizionati in base, per esempio, al bit N. Quando si verifica una microdiramazione la pipeline non può continuare. Per gestire questo caso abbiamo aggiunto alla micro-operazione il bit Goto. Quando incontra una micro-operazione che possiede questo bit attivo mentre la sta copiando all'interno della coda, l'unità di accodamento capisce in anticipo che ci sarà un problema e non spedisce all'unità di decodifica il segnale di conferma. Il risultato è

che la macchina rimarrà in stallo in questo punto finché la microdiramazione non sarà stata risolta.

Presumibilmente alcune istruzioni IJVM successive alla diramazione saranno già state portate nell'unità di decodifica (ma non nell'unità di accodamento) nel momento in cui si incontra una micro-operazione con il bit Goto attivo e non viene rispedito il segnale di conferma (cioè il permesso per continuare). Per riportare un po' d'ordine nella situazione e ritornare sui propri passi è necessario disporre di hardware speciale; la spiegazione di questi meccanismi esula però dagli scopi del libro. Edsger Dijkstra non ebbe idea di quanto avesse ragione quando scrisse il famoso: "GOTO Statement Considered Harmful" (Dijkstra, 1968a), cioè "L'istruzione GOTO è da considerarsi pericolosa".

Abbiamo percorso un lungo cammino a partire da Mic-1. Mic-1 è un componente hardware molto semplice, in cui quasi tutto il controllo è implementato in software. Mic-4 è organizzata altamente a pipeline, con sette stadi e un hardware molto più complesso. La pipeline è mostrata schematicamente nella Figura 4.36, in cui i numeri cerchiati fanno riferimento ai componenti della Figura 4.35. Mic-4 effettua automaticamente il prefetch di un flusso di byte dalla memoria, li decodifica trasformandoli in istruzioni IJVM che converte in una sequenza di micro-operazioni utilizzando una ROM; accoda poi queste microoperazioni per poterle utilizzare quando sarà necessario. Se lo si desidera è possibile legare i primi tre stadi della pipeline al clock del percorso dati, ma in realtà non c'è sempre del lavoro da eseguire. La IFU, per esempio, può certamente evitare di inviare durante ogni ciclo di clock un nuovo codice operativo IJVM all'unità di decodifica; le istruzioni IJVM richiedono infatti più cicli per essere eseguite e se si spedisce un codice operativo a ogni ciclo si riempirebbe rapidamente la coda.

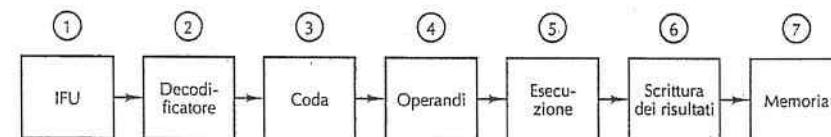


Figura 4.36 Pipeline di Mic-4.

A ogni ciclo di clock vengono traslati in avanti i registri MIR e la micro-operazione che si trova alla fine della coda viene copiata in MIR1 per iniziare la propria esecuzione. I segnali di controllo che partono dai quattro MIR si diffondono dunque lungo il percorso dati generando le azioni desiderate. Ciascun MIR controlla una parte diversa del percorso dati e quindi passi elementari differenti.

In questa architettura abbiamo una CPU organizzata profondamente a pipeline, con passi molto brevi e quindi una frequenza di clock molto elevata. Molte CPU sono progettate in questo modo, specialmente quelle che devono implementare un vecchio (CISC) insieme d'istruzioni. Per esempio, come vedremo più avanti nel corso di questo capitolo, l'implementazione del Core i7 è, per certi aspetti, concettualmente simile a Mic-4.

4.5 Miglioramento delle prestazioni

Tutti i produttori di calcolatori vorrebbero che le loro macchine funzionassero alla massima velocità possibile. In questo paragrafo presenteremo alcune tecniche avanzate sulle quali si sta focalizzando la ricerca attuale al fine di migliorare le prestazioni dei sistemi (principalmente quelle della CPU e della memoria). Per via della natura fortemente competitiva del mercato dei calcolatori l'intervallo di tempo che passa dal momento in cui la ricerca propone nuove idee per rendere più veloce un computer e la loro effettiva implementazione in prodotti commerciali è estremamente breve. Per questo motivo la maggior parte delle idee che tratteremo sono già in uso in un'ampia gamma di prodotti disponibili sul mercato.

È possibile dividere l'esposizione in due categorie principali: i miglioramenti dell'implementazione e quelli dell'architettura. I primi sono i modi di realizzare nuove CPU o nuove memorie che permettono al sistema di essere più veloce senza modificare l'architettura. Ciò significa che i vecchi programmi potranno essere eseguiti anche sulla nuova macchina, aspetto che rappresenta un grande vantaggio per il successo commerciale del prodotto. Un modo, ma non l'unico, per migliorare l'implementazione è l'utilizzo di un clock più veloce. I guadagni di prestazioni ottenuti dall'80386, passando per l'80486 e il Pentium, fino ad arrivare a progetti più recenti come il Core i7, sono dovuti a una migliore implementazione, mentre l'architettura è rimasta essenzialmente la stessa per tutte le macchine.

Alcuni tipi di miglioramenti possono essere ottenuti solamente cambiando l'architettura. A volte questi cambiamenti sono incrementali e consistono per esempio nell'aggiunta di nuove istruzioni o registri, in modo che i vecchi programmi possano continuare a funzionare anche sui nuovi modelli. In questo caso, per trarre pieno vantaggio dalle prestazioni della macchina, è necessario modificare il software o almeno ricompilarlo con un nuovo compilatore che tenga conto delle nuove funzionalità.

A volte i progettisti si rendono tuttavia conto che la vecchia architettura, essendo sopravvissuta fin troppo a lungo, ha esaurito la propria utilità e che l'unico modo per ottenere dei progressi è quello di ricominciare da capo progettandone una nuova. La rivoluzione RISC negli anni '80 fu uno di questi punti di rottura; un altro sta per arrivare, e nel corso del Capitolo 5 ne vedremo un esempio (l'Intel IA-64).

Nel resto del paragrafo analizzeremo quattro diverse tecniche per migliorare le prestazioni della CPU. Inizieremo presentando tre miglioramenti la cui implementazione si è ormai affermata nelle macchine attuali e successivamente passeremo a un miglioramento che, per funzionare al massimo, necessita di un piccolo supporto da parte dell'architettura. Queste tecniche sono la memoria cache, la predizione dei salti, l'esecuzione fuori sequenza con la rinomina dei registri e l'esecuzione speculativa.

4.5.1 Memoria cache

Nella storia della progettazione dei calcolatori una delle sfide più ardue è stata quella di definire un sistema di memoria capace di fornire al processore gli operandi alla velocità alla quale esso li può elaborare. Il recente ed elevato tasso di crescita della velocità dei processori non è stato accompagnato da un analogo incremento della velocità delle

memorie. Negli ultimi decenni le memorie sono diventate più lente rispetto alle CPU. Per via dell'enorme importanza che riveste la memoria primaria questa situazione ha frenato in modo significativo lo sviluppo di sistemi ad alte prestazioni, e allo stesso tempo ha stimolato la ricerca di nuovi modi per aggirare il problema della velocità della memoria, che diventa di anno in anno più lenta rispetto alla CPU.

I processori moderni effettuano una quantità travolgente di richieste al sistema di memoria, sia in termini di latenza (il ritardo nel fornire un operando) sia in termini di banda (la quantità di dati fornita per unità di tempo). Questi due aspetti che caratterizzano un sistema di memoria sono purtroppo in larga parte in conflitto. Molte tecniche che permettono di aumentare la larghezza di banda incrementano allo stesso tempo la latenza. Per esempio, la pipeline utilizzata in Mic-3 può essere applicata a un sistema di memoria in modo da gestire efficientemente la sovrapposizione di più richieste alla memoria. Purtroppo, come per Mic-3, si ottiene una latenza maggiore per le singole operazioni di memoria. Dato che la velocità di clock del processore continua ad aumentare, diventa sempre più difficile realizzare un sistema di memoria in grado di fornire gli operandi in uno o due cicli di clock.

Un modo per combattere questo problema è l'uso delle cache. Come abbiamo visto nel Paragrafo 2.2.5 le cache mantengono all'interno di una piccola e veloce memoria le parole di memoria utilizzate più di recente, in modo da accelerarne l'accesso. Se nella cache è presente una percentuale sufficientemente elevata delle parole di memoria necessarie, è possibile ridurre enormemente l'effettiva latenza della memoria.

Una delle tecniche più efficaci per migliorare sia la larghezza di banda sia la latenza è l'uso di più cache. Una tecnica elementare che funziona in modo efficace consiste nell'introdurre due cache separate, una per le istruzioni e una per i dati. Il sistema con due cache distinte per i dati e per le istruzioni, spesso chiamato a cache separata, fornisce vari vantaggi. In primo luogo è possibile far partire le operazioni di memoria indipendentemente per ciascuna cache, raddoppiando effettivamente la larghezza di banda del sistema di memoria. Questo è il motivo per cui ha senso fornire due porte distinte per la memoria, come abbiamo fatto in Mic-1: ogni porta ha la propria cache. Occorre notare che le due cache hanno accessi indipendenti alla memoria centrale.

Attualmente molti sistemi di memoria sono ancora più complicati. Spesso tra la memoria centrale e le cache delle istruzioni e dei dati è presente un'altra cache, chiamata **cache di secondo livello**. In realtà, per disporre di sistemi di memoria ancora più sofisticati, si possono avere anche tre o più livelli di cache. Il chip stesso della CPU contiene una piccola cache per le istruzioni e una piccola cache per i dati, le cui dimensioni sono in genere comprese tra 16 e 64 KB. Dopo questa piccola memoria c'è una cache di secondo livello, che non si trova nel chip della CPU, ma che può essere inclusa all'interno del suo involucro, vicina al chip e a esso collegata tramite circuiti ad alta velocità. Questa cache è di solito unificata e contiene quindi sia dati sia istruzioni. Generalmente la dimensione delle cache L2 è compresa tra 512 KB e 1 MB. La cache di terzo livello si trova sulla scheda del processore ed è costituita da alcuni megabyte di SRAM, un tipo di memoria molto più veloce rispetto alla memoria DRAM centrale. Di solito le cache sono annidate, nel senso che l'intero contenuto della cache di primo livello è compreso nella cache di secondo livello e tutto il contenuto di quest'ultima è compreso in quella di terzo livello.

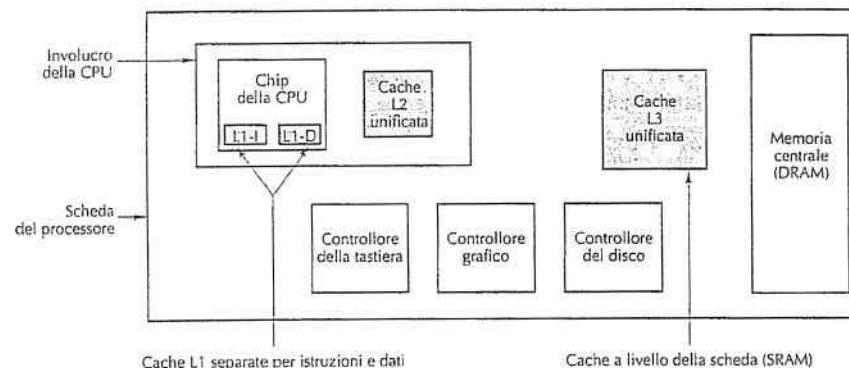


Figura 4.37 Sistema con tre livelli di cache.

Per raggiungere il proprio scopo le cache sfruttano due tipi di località degli indirizzi. La **località spaziale** riflette la considerazione che è molto probabile che in un prossimo futuro si accederà a locazioni di memoria i cui indirizzi sono numericamente simili a quello di una locazione appena utilizzata. La **località temporale** si verifica quando si accede nuovamente a locazioni di memoria già utilizzate di recente. Ciò potrebbe avvenire per esempio nel caso di locazioni di memoria vicine alla cima dello stack oppure nel caso delle istruzioni di un ciclo. Nella progettazione di una cache la località temporale viene principalmente sfruttata scegliendo che cosa scartare nel caso in cui si verifichi un fallimento della cache. Molti algoritmi di sostituzione sfruttano la località temporale scartando gli elementi che non sono stati utilizzati di recente.

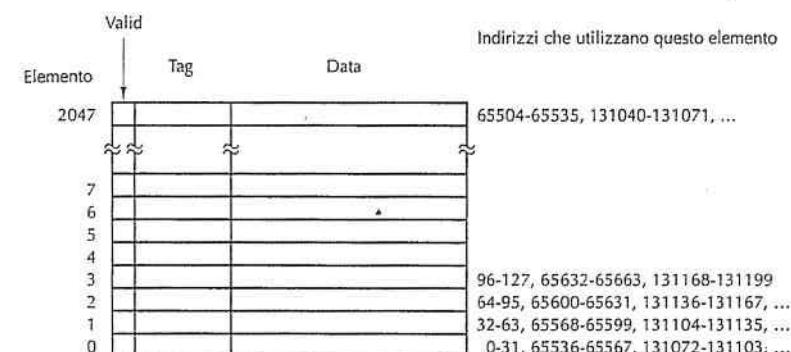
Tutte le cache utilizzano lo stesso modello. La memoria centrale è divisa in blocchi di dimensione fissa chiamati **linee di cache**. Una linea di cache è composta generalmente da 4 a 64 byte consecutivi. Le linee sono numerate consecutivamente a partire da 0: se la dimensione della linea è di 32 byte, la linea 0 conterrà i byte compresi tra 0 e 31, la linea 1 quelli compresi tra 32 e 63, e così via. In ogni momento la cache contiene delle linee. Quando si effettua un riferimento alla memoria il circuito che controlla la cache verifica se la parola richiesta si trova nella cache. Se così è, la parola può essere utilizzata, risparmiando un viaggio fino alla memoria centrale. In caso contrario, viene rimossa una linea dalla cache per sostituirla con la linea richiesta, prelevata dalla memoria centrale oppure da un livello di cache più distante. Esistono molte varianti di questo schema, ma in tutte l'idea base è di tenere il più a lungo possibile nella cache le linee usate più frequentemente, in modo da massimizzare il numero di riferimenti alla memoria che la cache riesce a soddisfare.

Cache a corrispondenza diretta

La cache più semplice è conosciuta con il nome di **cache a corrispondenza diretta** (*direct-mapped cache*). La Figura 4.38(a) mostra un esempio di cache di questo tipo che contiene 2048 elementi. Ciascun elemento (riga) della cache può memorizzare esatta-

mente una linea di cache della memoria centrale. Se ipotizziamo che la linea di cache abbia dimensione di 32 byte, la cache può memorizzare 2048 elementi di 32 byte, per un totale di 64 KB. Gli elementi della cache sono composti da tre parti.

1. Il bit **Valid** che indica se il dato nell'elemento è valido oppure no. All'avvio del sistema tutti gli elementi della cache sono marcati come non validi.
2. Il campo **Tag** che è un valore univoco a 16 bit, corrispondente alla linea di memoria da cui provengono i dati.
3. Il campo **Data** che contiene una copia del dato della memoria. Questo campo memorizza una linea di cache di 32 byte.



(a)

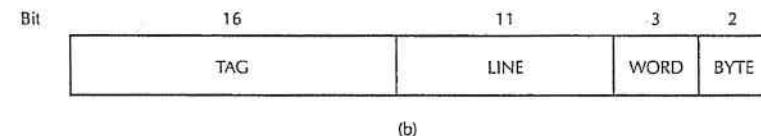


Figura 4.38 (a) Cache a corrispondenza diretta. (b) Indirizzo virtuale a 32 bit.

Nelle cache a corrispondenza diretta una data parola di memoria può essere memorizzata in un'unica posizione della cache. Per ogni indirizzo di memoria esiste un solo posto all'interno della cache in cui cercare il dato. Per memorizzare e prelevare dati dalla cache l'indirizzo è diviso in quattro campi, come mostra la Figura 4.38(b):

1. il campo **TAG** corrisponde ai bit Tag memorizzati in un elemento della cache;
2. il campo **LINE** indica l'elemento della cache contenente i dati corrispondenti, se sono presenti;

3. il campo WORD indica a quale parola si fa riferimento all'interno della linea;
4. il campo BYTE non viene generalmente utilizzato, ma se si richiede un solo byte esso indica quale byte è richiesto all'interno della parola. Per una cache che fornisce soltanto parole a 32 bit, questo campo varrà sempre 0.

Quando la CPU genera un indirizzo di memoria, l'hardware estrae dall'indirizzo gli 11 bit del campo LINE e li usa come indice all'interno della cache per cercare uno dei 2048 elementi. Se questo elemento è valido si effettua un confronto tra il campo TAG dell'indirizzo di memoria e il campo Tag dell'elemento della cache. Se sono uguali, l'elemento della cache contiene la parola richiesta; questa situazione è chiamata *cache hit* ("successo della cache"). In tal caso la parola che si vuole leggere può essere presa direttamente dalla cache, evitando di dover andare fino alla memoria. Dalla linea di cache viene estratta solamente la parola effettivamente richiesta, mentre non viene utilizzato il resto della linea. Se l'elemento della cache non è valido oppure se i due campi "tag" non corrispondono, il dato richiesto non è presente nella cache: questa situazione è chiamata *cache miss* ("fallimento della cache"). In questo caso la linea della cache a 32 byte viene prelevata dalla memoria e memorizzata nell'elemento appropriato della cache, sostituendo quello che era presente precedentemente. In ogni caso se l'elemento della cache era stato modificato dopo averlo caricato, occorre riscriverlo in memoria prima di poterlo sovrascrivere.

Nonostante la complessità della decisione, l'accesso alla parola richiesta può essere notevolmente veloce. Nel momento in cui si conosce l'indirizzo, si conosce anche l'esatta locazione della parola *sempre che si trovi nella cache*. Ciò significa che è possibile leggere la parola e spedirla al processore nello stesso momento in cui esso sta controllando se la parola è quella corretta (confrontando i due "tag"). Il processore riceve quindi dalla cache una parola nello stesso momento (se non addirittura in anticipo) in cui viene a sapere se la parola è effettivamente quella richiesta.

Questo schema di corrispondenza inserisce linee di memoria consecutive in elementi della cache consecutivi. Nella cache è possibile memorizzare effettivamente fino a 64 KB di dati contigui. Tuttavia all'interno della cache non è possibile memorizzare allo stesso tempo due linee i cui indirizzi differiscono esattamente di 64 KB (65.536 byte) o di un multiplo di questo valore (dato che il loro campo LINE è uguale). Per esempio, se un programma accede a dati che si trovano nella locazione X e successivamente esegue un'istruzione che richiede dati presenti alla locazione X + 65.536 (o in qualsiasi altra locazione della stessa linea), la seconda istruzione obbligherà a ricaricare l'elemento della cache, sovrascrivendo quello precedente. Se ciò avviene con una certa frequenza, le prestazioni degradano. Il comportamento di una cache può essere addirittura peggior di quanto non sarebbe in assenza di cache, dato che ogni operazione di memoria comporta la lettura di un'intera linea di cache e non semplicemente di una parola.

Le cache a corrispondenza diretta sono le più comuni e funzionano in modo efficiente, dato che collisioni come quelle descritte precedentemente si verificano molto raramente, se non addirittura mai. Un compilatore molto astuto, nel posizionare i dati e le istruzioni all'interno della memoria, potrebbe per esempio tener conto delle possibili collisioni della cache, cercando di evitarle. Si noti che il caso particolare descritto precedentemente non si potrebbe verificare in un sistema con cache separate per le istruzio-

ni e per i dati, in quanto le richieste che erano in collisione verrebbero in questo caso gestite da cache differenti. Utilizzare due cache al posto di una sola porta un secondo vantaggio: una maggiore flessibilità nella gestione d'indirizzi di memoria in conflitto.

Cache set-associative

Com'è stato detto precedentemente, molte linee della memoria sono in competizione per l'utilizzo di uno stesso elemento della cache. Se un programma che utilizza la cache della Figura 4.38(a) usa intensamente le parole comprese tra l'indirizzo 0 e l'indirizzo 65.536, si verificheranno costantemente dei conflitti e ogni riferimento alla memoria potrebbe potenzialmente espellere un elemento dalla cache. Una soluzione a questo problema è quello di consentire che in ciascun elemento della cache ci possano essere due o più linee. Una cache con n possibili elementi per ciascun indirizzo è chiamata *cache set-associativa a n vie*. La Figura 4.39 mostra una cache associativa a quattro vie.

Queste cache sono più complesse di quelle a corrispondenza diretta. Infatti, anche se l'insieme di elementi della cache da esaminare viene calcolato a partire dall'indirizzo di memoria, è tuttavia necessario controllare un insieme di n elementi per vedere se la linea richiesta è presente nella cache. Questa verifica deve essere effettuata molto rapidamente. L'esperienza e le simulazioni hanno mostrato tuttavia che cache a due o a quattro vie garantiscono prestazioni sufficientemente buone, da rendere accettabile l'aumento dei componenti digitali richiesti.

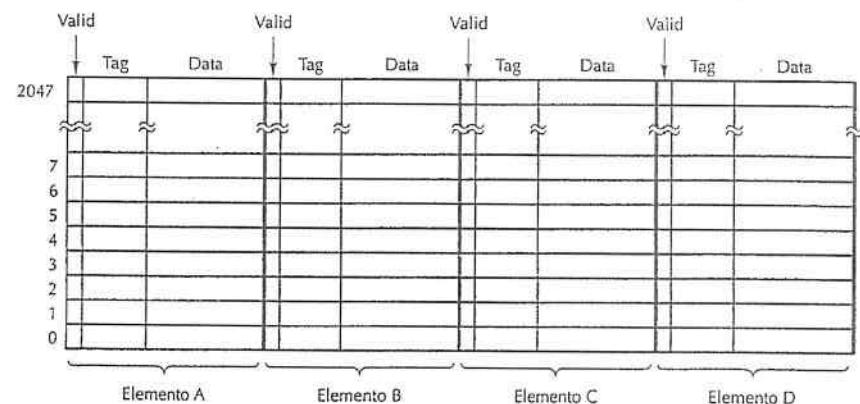


Figura 4.39 Cache associativa a quattro vie.

L'uso di una cache set-associativa pone il progettista di fronte a una scelta: quando viene portata all'interno della cache una nuova linea, quale delle presenti deve essere scartata? Per poter compiere la scelta migliore bisognerebbe poter predire il futuro; un algoritmo che nella maggior parte delle situazioni funziona abbastanza bene è detto *LRU* (acronimo di *Least Recently Used*, "meno recentemente usata"). Questo algoritmo mantiene un

ordinamento temporale di ciascun insieme di elementi ai quali si può far riferimento in base a una certa locazione di memoria. Quando si accede a una delle linee presenti nella cache l'algoritmo aggiorna la lista marcando l'elemento utilizzato più di recente. Quando giunge il momento di sostituire un elemento, l'algoritmo scarta quello che si trova alla fine della lista, cioè quello al quale è stato effettuato un accesso meno di recente.

Portando l'idea all'estremo è possibile immaginare una cache a 2048 vie contenente un unico insieme di 2048 linee. In questo caso tutti gli indirizzi di memoria vengono mappati in un unico insieme e quindi la ricerca all'interno della cache obbliga a confrontare l'indirizzo con 2048 diversi tag. Occorre notare che ciascun elemento deve avere dei componenti logici che permettano di controllare la corrispondenza con il tag. Dato che la lunghezza del campo LINE è pari a 0, il campo TAG costituisce l'intero indirizzo, fatta eccezione per i campi WORD e BYTE. Inoltre quando viene rimpiazzata una linea della cache, tutte e 2048 le locazioni sono candidate alla sostituzione. Per mantenere e gestire una lista di 2048 elementi è necessario un lavoro talmente grande da rendere inapplicabile l'algoritmo LRU. (Ricordiamoci che questa lista deve essere aggiornata per ogni operazione di memoria, non soltanto quando si verifica un fallimento della cache.) Nella maggior parte delle circostanze le cache set-associative a molte vie non hanno prestazioni sensibilmente migliori di quelle con un piccolo numero di vie. Il risultato sorprendente è che in alcuni casi le prestazioni sono addirittura peggiori. Per questi motivi è raro incontrare cache con più di quattro vie.

Infine, anche le operazioni di scrittura pongono un particolare problema nell'uso delle cache. Quando un processore scrive una parola, e questa parola è presente nella cache, ovviamente esso deve aggiornare la parola oppure scaricare l'elemento dalla cache. In quasi tutte le progettazioni si sceglie di aggiornare la cache. Ma che cosa si può dire riguardo l'aggiornamento della copia nella memoria centrale? Questa operazione può essere rinviata finché la linea della cache non sia pronta a essere sostituita dall'algoritmo LRU. La scelta in questo caso è difficile e nessuna opzione è nettamente migliore delle altre. L'aggiornamento immediato nella memoria centrale dell'elemento della cache è chiamato *write-through* ("scrivi attraverso"). Questo approccio è generalmente il più semplice e il più affidabile, dato che il contenuto della memoria rimane sempre aggiornato; ciò è utile, per esempio, se si verifica un errore ed è necessario recuperare lo stato della memoria. Purtroppo però questa tecnica richiede molto traffico verso la memoria; per questo motivo si tende a utilizzare un'implementazione più sofisticata, conosciuta con il nome di *write-deferred* ("scrittura ritardata") o *write-back*.

Sempre al riguardo delle operazioni di scrittura occorre affrontare un ulteriore problema: che cosa fare se si verifica una scrittura in una locazione che in quel momento non è memorizzata nella cache? Bisogna portare il dato nella cache o basta scriverlo in memoria? Anche in questo caso nessuna delle due risposte è migliore dell'altra. La maggior parte dei progetti che ritardano la scrittura in memoria tende a portare i dati nella cache quando si verifica un fallimento in scrittura; questa tecnica è conosciuta con il nome di *write allocation* ("allocazione in scrittura"). Al contrario, quando si impiega la scrittura diretta, si tende generalmente a non allocare un elemento della cache a ogni operazione di scrittura, dato che ciò complicherebbe un'organizzazione altrimenti sem-

plice. L'allocazione in scrittura risulta vincente solo nel caso di ripetute scritture della stessa parola o di diverse parole facenti però parte di una stessa linea della cache.

Le prestazioni della cache influenzano largamente le prestazioni generali del sistema, dato che la differenza tra la velocità della CPU e quella della memoria è molto grande. Per questa ragione la ricerca di migliori strategie per utilizzare le cache è un campo di studio ancora molto attivo (Sanchez e Kozyrakis, 2011; Gaur et al., 2011).

4.5.2 Predizione dei salti

I calcolatori moderni sono profondamente organizzati a pipeline. La pipeline della Figura 4.36 ha sette stadi, mentre quelle dei calcolatori ad alte prestazioni ne hanno spesso 10 o più. L'uso delle pipeline funziona in modo ottimale su codice lineare, dato che l'unità di prelievo può semplicemente leggere dalla memoria parole consecutive e spedirle all'unità di decodifica prima ancora che siano effettivamente richieste.

L'unico, piccolo, problema di questo magnifico modello è che non è, neanche in piccola parte, realistico. I programmi non sono sequenze lineari di codice, ma sono pieni d'istruzioni di salto. Consideriamo le semplici istruzioni della Figura 4.40(a). La variabile *i* viene confrontata con 0 (probabilmente il test più comune) e, a seconda del risultato, si assegna un valore diverso alla variabile *k*.

La Figura 4.40(b) mostra una traduzione in linguaggio assemblativo che verrà esaminato più avanti nel corso del libro; per ora i dettagli non sono importanti, in ogni caso a seconda della macchina e del compilatore con ogni probabilità il codice assomiglierà più o meno a quello mostrato nella Figura 4.40(b). La prima istruzione confronta *i* con 0. La seconda effettua una diramazione verso l'etichetta *Else* (l'inizio della clausola *else*) nel caso in cui *i* sia diverso da 0. La terza istruzione assegna a *k* il valore 1. La quarta effettua una diramazione verso il codice dell'istruzione successiva. Il compilatore ha opportunamente inserito in quel punto un'etichetta, *Next*, in modo che ci sia una destinazione verso la quale effettuare il salto. La quinta istruzione assegna a *k* il valore 2.

<pre> if (i == 0) k = 1; else k = 2; </pre> <p>(a)</p>	<pre> Then: CMP i,0 ; confronta i con 0 BNE Else ; salta a Else se diversi MOV k,1 ; sposta 1 in k BR Next ; salta a Next Else: MOV k,2 ; sposta 2 in k Next: </pre> <p>(b)</p>
--	--

Figura 4.40 (a) Frammento di programma. (b) Il frammento tradotto in un linguaggio assemblativo.

In questo esempio l'aspetto che merita una particolare osservazione è che due delle cinque istruzioni sono salti. Inoltre una di queste, BNE, è un salto condizionale, ovvero un salto effettuato se e soltanto se è soddisfatta una particolare condizione: in questo caso, il fatto che i due operandi della precedente istruzione, CMP, siano diversi. La più lunga sequenza lineare di codice è composta da due sole istruzioni. Per questi motivi

risulta particolarmente difficile prelevare le istruzioni a una velocità elevata in modo da alimentare la pipeline.

A prima vista potrebbe sembrare che i salti incondizionati, come l'istruzione **BR Next** della Figura 4.40(b), non costituiscano un problema. Dopo tutto non c'è alcuna ambiguità sulla destinazione da raggiungere. Perché l'unità di prelievo non può semplicemente continuare a leggere le istruzioni a partire dall'indirizzo destinazione (il punto verso il quale viene effettuato il salto)?

Il problema risiede nella natura stessa della pipeline. Nella Figura 4.36 possiamo vedere per esempio che la decodifica dell'istruzione si svolge nel secondo stadio. L'unità di prelievo deve quindi decidere dove prelevare le istruzioni successive ancor prima di conoscerne il tipo. Può scoprire di aver prelevato un salto incondizionato soltanto dopo un ciclo, nel momento in cui ha già cominciato a prelevare l'istruzione che segue il salto incondizionato. La conseguenza è che un buon numero di macchine organizzate a pipeline (come l'UltraSPARC III) ha la proprietà di eseguire l'istruzione *che segue* un salto incondizionato, anche se logicamente non dovrebbe farlo. La posizione immediatamente successiva a un salto è chiamata **posizione di ritardo**. Il Core i7 (e la macchina usata nella Figura 4.40(b)) non hanno questa proprietà, ma la complessità interna che permette di evitare questo problema è enorme. Un compilatore ottimizzato cercherà di inserire nelle posizioni di ritardo alcune istruzioni utili; spesso però non ci sono istruzioni disponibili e in tal caso il compilatore è obbligato a inserire in quel punto un'istruzione NOP. L'aggiunta di queste istruzioni permette di mantenere corretto il programma, rendendolo però più lento e di dimensioni maggiori.

Se i salti incondizionati provocano delle noie, quelli condizionali ne provocano di più. Non solo occorre considerare anche in questo caso delle posizioni di ritardo, ma oltretutto l'unità di prelievo viene a conoscenza dell'indirizzo in cui deve leggere in un momento della pipeline ancora più ritardato. Le prime macchine a pipeline rimanevano semplicemente in **stallo** finché non venivano a sapere se occorresse o meno effettuare la diramazione. Nel caso in cui i salti condizionali costituiscono il 20% delle istruzioni il fatto di rimanere in stallo per tre o quattro cicli a ogni salto condizionale è letale per le prestazioni.

Per queste ragioni quello che fa la maggior parte delle macchine quando incontra un salto condizionale è predire se la diramazione verrà o meno effettuata. Sarebbe bello se si potesse collegare una sfera di cristallo a uno slot PCIe libero (o meglio, nell'IFU) in modo da aiutare la previsione; finora però nessuno è riuscito a implementare questo approccio.

In mancanza di una simile periferica sono stati escogitati vari metodi per effettuare la predizione. Un metodo molto semplice è il seguente: si assume che debbano essere effettuati tutti i salti condizionali all'indietro, e non tutti quelli in avanti. Il ragionamento alla base della prima parte di questa strategia è che i salti all'indietro sono spesso collocati alla fine di un ciclo. La maggior parte dei cicli viene eseguita più volte e quindi, in genere, ha senso scommettere che occorra compiere un salto all'indietro per tornare all'inizio del ciclo.

La seconda parte del ragionamento si fonda su una base meno solida. Alcuni salti in avanti si verificano quando si rilevano nel software delle condizioni di errore (per esem-

pio, non è possibile aprire un file). Gli errori sono rari, quindi la maggior parte dei salti loro associati non dovrebbe essere effettuata. Ovviamente esiste un gran numero di salti che non sono legati alla gestione degli errori e la frequenza di successi non è altrettanto buona quanto quella dei salti all'indietro. Questa regola, pur non essendo l'ideale, comunque è... meglio di niente.

Se si predice correttamente una diramazione non occorre compiere alcuna operazione particolare. L'esecuzione continua semplicemente a partire dall'indirizzo di destinazione. Il problema sorge quando la predizione si rivela errata. Determinare dove andare, e andarci, non è difficile. La parte complicata è annullare le istruzioni già eseguite che non avrebbero dovuto esserlo.

Esistono due modi di affrontare questa situazione. Il primo consiste nel consentire, dopo aver predetto un salto, l'esecuzione delle istruzioni solo finché queste non si apprestino a modificare lo stato della macchina (per esempio memorizzando un valore in un registro). Invece di sovrascrivere il registro, il valore calcolato viene inserito in un registro temporaneo (nascosto) e copiato in quello reale solo dopo aver verificato che la predizione era corretta. Il secondo metodo consiste nel registrare (per esempio in un registro temporaneo nascosto) il valore di ogni registro in procinto di essere sovrascritto, in modo che la macchina possa essere riportata allo stato in cui era nel momento in cui è stata effettuata la predizione errata. Entrambe le soluzioni sono complesse e per funzionare correttamente devono tener traccia di una quantità industriale d'informazioni. La situazione può diventare veramente complessa se si incontra un secondo salto condizionale ancor prima di scoprire se il primo era stato predetto correttamente o meno.

Predizione dinamica dei salti

Il fatto che le predizioni siano accurate è ovviamente molto importante, dato che in tal caso la CPU ha la possibilità di procedere alla massima velocità. Per questo motivo gran parte della ricerca attuale punta a migliorare gli algoritmi di predizione dei salti (Chen et al., 2003; Falcon et al., 2004; Jimenez, 2003; Parikh et al., 2004). Uno dei possibili approcci prevede che la CPU abbia (in appositi componenti hardware) una tabella della storia dei salti, nella quale tiene traccia dei salti condizionali incontrati in modo da poterli rianalizzare quando si verificano nuovamente. La versione più semplice di questo approccio è mostrata nella Figura 4.41(a). In questo caso la tabella della storia dei salti contiene un elemento per ogni istruzione di salto condizionale; l'elemento contiene l'indirizzo dell'istruzione di salto oltre a un bit che indica se è stata effettuata la diramazione l'ultima volta che l'istruzione è stata eseguita. Usando questo schema la predizione consiste semplicemente nell'assumere che il salto si comporterà allo stesso modo dell'occorrenza precedente. Se la predizione si rivela sbagliata, allora si modifica il bit nella tabella della storia dei salti.

Esistono vari modi per organizzare questa tabella, e corrispondono in sostanza alle possibili organizzazioni delle cache. Consideriamo una macchina con istruzioni a 32 bit, allineate in corrispondenza delle parole, in modo che i 2 bit meno significativi degli indirizzi di memoria siano sempre 00. Con una tabella della storia a corrispondenza diretta che contiene 2^n elementi è possibile estrarre gli $n + 2$ bit meno significativi dell'indirizzo destinazione di un'istruzione di salto e traslarli a destra di 2 bit; questo

numero a n bit può essere utilizzato come indice della tabella della storia. L'indice così ottenuto permette di controllare se l'indirizzo memorizzato corrisponde a quello del salto. Come per le cache, anche in questo caso non è necessario memorizzare gli $n+2$ bit meno significativi, che possono quindi essere omessi (e vengono memorizzati solamente i bit più significativi, cioè il tag). Se l'elemento cercato è presente nella tabella, la predizione utilizza il bit come previsione del futuro comportamento del salto. Se invece è presente un tag errato oppure l'elemento della tabella non è valido, si verifica un fallimento, allo stesso modo in cui avviene nelle cache. In questo caso è possibile utilizzare la regola dei salti in avanti/all'indietro.

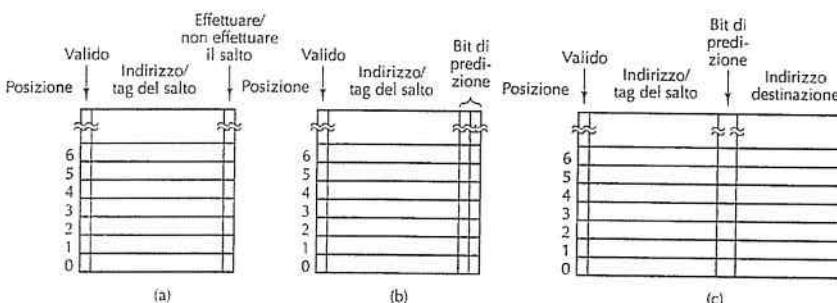


Figura 4.41 (a) Storia dei salti a 1 bit. (b) Storia dei salti a 2 bit. (c) Corrispondenza tra gli indirizzi delle istruzioni di salto e gli indirizzi destinazione.

Se la tabella della storia ha, per esempio, 4096 elementi, i salti che si trovano agli indirizzi 0, 16384, 32768, ... saranno in conflitto, in modo analogo al problema già visto con le cache. È possibile adottare la stessa soluzione: un elemento associativo a 2, a 4 o a n -vie. Come per la cache il fattore limitante è costituito da un singolo insieme a n -vie, in cui occorre confrontare un indirizzo con tutti i tag.

Se la larghezza della tabella è abbastanza grande e se vi è sufficiente associatività, allora questo schema funziona bene nella maggior parte delle situazioni. Ciononostante esiste un problema che può verificarsi in modo sistematico.

Quando si termina l'esecuzione di un ciclo, il salto che si trova alla fine verrà predetto in modo errato; ancor peggio, questo errore di previsione cambierà il bit nella tabella dei salti per indicare che in futuro occorrerà prevedere "di non effettuare il salto". Quando si rientrerà nel ciclo, alla fine della prima iterazione il salto che si trova in fondo verrà predetto in modo errato. Questo problema si può verificare spesso se il ciclo è innestato all'interno di un altro ciclo oppure nel caso in cui faccia parte di una procedura richiamata frequentemente.

Per eliminare questa previsione sbagliata possiamo dare una seconda chance all'elemento della tabella, ovvero modificare la previsione soltanto dopo che due previsioni consecutive si sono rivelate errate. Questo approccio, mostrato nella Figura 4.41(b),

richiede che nella tabella della storia siano presenti due bit di predizione, uno per ciò che "si suppone" occorra fare con il salto e uno per tener traccia di ciò che si è verificato l'ultima volta.

Un modo leggermente diverso per interpretare questo algoritmo consiste nel vederlo come un automa a stati finiti con quattro stati (Figura 4.42). Dopo una serie di corrette previsioni di "non effettuare il salto", l'automa si troverà nello stato 00 che predice di "non effettuare il salto" alla volta successiva. Se quella previsione è errata, passerà allo stato 01, ma continuerà a predire di "non effettuare il salto". Solo se anche questa previsione si rivela errata, allora la macchina passerà allo stato 11 suggerendo di effettuare la diramazione. In sostanza il bit a sinistra nello stato rappresenta la previsione, mentre il bit a destra tiene traccia dell'ultimo salto. Questa strategia prevede di utilizzare soltanto 2 bit per memorizzare la storia dei salti, ma se ne possono usare anche 4 o 8.

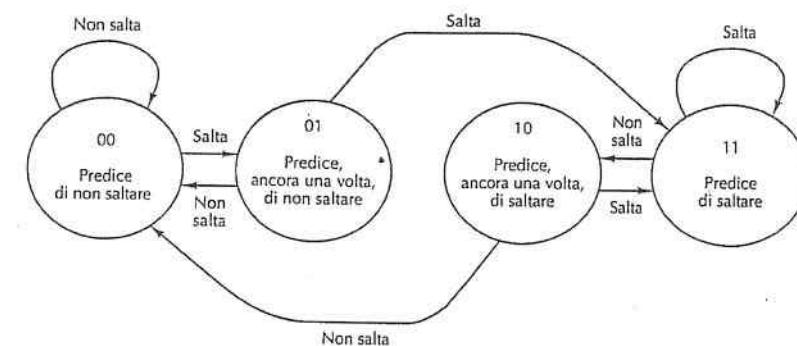


Figura 4.42 Automa a stati finiti a 2 bit per la predizione delle diramazioni.

Già nella Figura 4.28 avevamo visto un automa a stati finiti. In effetti tutti i nostri microprogrammi possono essere descritti da questo modello in cui ogni linea rappresenta un particolare stato nel quale si può trovare l'automa e in cui è possibile compiere delle transizioni verso un insieme finito di stati. Gli automi a stati finiti sono utilizzati diffusamente in varie fasi della progettazione hardware.

Finora abbiamo assunto che la destinazione di un salto condizionale fosse nota, generalmente come indirizzo esplicito al quale andare (contenuto all'interno dell'istruzione stessa) oppure come uno spiazzamento relativo all'istruzione corrente (cioè, un valore con segno da aggiungere al contatore d'istruzioni). Questa assunzione è valida nella maggior parte dei casi, tuttavia esistono alcune istruzioni di salti condizionali che calcolano l'indirizzo di destinazione eseguendo delle operazioni aritmetiche sui registri. L'automa della Figura 4.42 predice in modo accurato se intraprendere oppure no i salti, ma queste previsioni sono inutili se l'indirizzo di destinazione è sconosciuto. Un modo per gestire questa situazione consiste nel memorizzare nella tabella della storia l'effettivo indirizzo verso il quale è stato effettuato il salto l'ultima volta, come mostra la Figura

ra 4.41(c). Seguendo questo metodo, se la tabella indica che l'ultima volta il salto all'indirizzo 516 è stato effettuato e ha portato all'indirizzo 4000, allora, nel caso in cui la predizione dica di effettuare la diramazione, l'esecuzione continuerà nuovamente a partire dall'indirizzo 4000.

Un approccio differente per la predizione dei salti consiste nel tener traccia del fatto che gli ultimi k salti condizionali siano stati oppure no effettuati, indipendentemente dalle istruzioni alle quali erano associati. Questo numero a k bit, memorizzato in un **registro a scorrimento della storia dei salti**, viene successivamente confrontato in parallelo con tutte le chiavi a k bit degli elementi di una tabella della storia; se il confronto ha esito positivo si utilizza la predizione trovata nella tabella. Anche se può sembrare strano questa tecnica funziona piuttosto bene.

Predizione statica delle diramazioni

Tutte le tecniche per la predizione delle diramazioni viste finora sono di tipo dinamico, cioè vengono effettuate durante l'esecuzione del programma. Uno dei vantaggi delle tecniche dinamiche è che si adattano al comportamento corrente del programma. Lo svantaggio è che richiedono un costoso hardware specializzato e una grande complessità del chip.

Una strategia alternativa consiste nel richiedere l'aiuto del compilatore. Quando il compilatore incontra un'istruzione come

```
for (i = 0; i < 1000000; i++) { ... }
```

sa con certezza che la diramazione alla fine del ciclo sarà effettuata quasi tutte le volte. Se esistesse un modo per comunicarlo all'hardware, si potrebbe risparmiare una gran quantità di lavoro.

Anche se ciò rappresenta una modifica architettonica (e non soltanto un problema d'implementazione), alcune macchine, come l'UltraSPARC III, hanno un secondo insieme d'istruzioni di salto condizionale in aggiunta a quelle ordinarie (necessarie per garantire la retrocompatibilità). Le nuove istruzioni contengono un bit attraverso cui il compilatore può specificare se ritiene che la diramazione verrà oppure no effettuata. Quando si incontra una di queste istruzioni l'unità di fetch esegue semplicemente l'azione specificata dal bit. Per di più queste istruzioni non richiedono spazio prezioso nella tabella delle diramazioni, riducendo di conseguenza i conflitti al suo interno.

L'ultima tecnica che consideriamo per effettuare la predizione dei salti si basa sul *profiling* (Fisher e Freudenberger, 1992). Anche questa tecnica è di tipo statico, ma, invece di delegare al compilatore il compito di prevedere quali diramazioni verranno oppure no effettuate, si manda in esecuzione il programma (di solito su un simulatore) per catturare il comportamento delle diramazioni. Questa informazione viene fornita al compilatore, che la utilizza per impostare le istruzioni speciali di salto condizionale mediante le quali può specificare all'hardware in che modo deve comportarsi.

4.5.3 Esecuzione fuori sequenza e rinomina dei registri

La maggior parte delle CPU moderne funziona sia a pipeline sia in modo superscalare, com'è illustrato nella Figura 2.6. In generale ciò comporta la presenza di un'unità di

fetch che preleva istruzioni dalla memoria prima che siano necessarie, in modo da alimentare un'unità di decodifica. L'unità di decodifica attribuisce l'istruzione decodificata all'unità funzionale appropriata perché sia eseguita. In alcuni casi l'unità di decodifica può scomporre le singole istruzioni in micro-operazioni prima di distribuirle alle unità funzionali, secondo le capacità delle unità stesse.

Ovviamente l'architettura più semplice prevede che tutte le istruzioni siano eseguite nell'ordine in cui sono prelevate dalla memoria (ipotizzando per il momento che l'algoritmo di predizione delle diramazioni non sbagli mai). Tuttavia l'esecuzione in ordine non sempre fornisce prestazioni ottimali a causa delle dipendenze tra istruzioni. Le istruzioni che richiedono un valore calcolato da un'istruzione precedente non possono essere eseguite fintanto che quell'istruzione non abbia prodotto il valore desiderato. In tali situazioni (dipendenze RAW) le istruzioni devono aspettare. Esistono però anche altri tipi di dipendenze, come mostreremo in seguito.

Nel tentativo di aggirare questi problemi e ottenere prestazioni migliori, alcune CPU permettono di saltare le istruzioni dipendenti per raggiungere le istruzioni successive che non lo sono. Naturalmente, l'algoritmo interno di scheduling delle istruzioni impiegato deve garantire che l'esecuzione produca lo stesso risultato che produrrebbe l'esecuzione ordinata. Vediamo come funziona il riordinamento delle istruzioni su di un esempio dettagliato.

Per illustrare la natura del problema, cominciamo con una macchina che lancia le istruzioni sempre secondo l'ordine in cui appaiono nel programma e che, inoltre, richiede vengano eseguite nello stesso ordine. L'importanza di quest'ultima caratteristica risulterà evidente in seguito.

La macchina dell'esempio ha otto registri visibili al programmatore, con nomi che vanno da R0 a R7. Tutte le istruzioni aritmetiche usano tre registri: due per gli operandi e uno per il risultato, così come avviene nel Mic-4. Ipotizziamo che, se un'istruzione è decodificata nel ciclo n , allora la sua esecuzione comincia al ciclo $n + 1$. Per un'istruzione semplice, come l'addizione o la sottrazione, la scrittura del risultato nel registro di destinazione avviene alla fine del ciclo $n + 2$. Per un'istruzione più complessa, come la moltiplicazione, la scrittura avviene alla fine del ciclo $n + 3$. Per rendere l'esempio realistico, consentiamo all'unità di decodifica di lanciare fino a due istruzioni per ciclo di clock. Le CPU superscalari in commercio possono lanciare comunemente dalle quattro alle sei istruzioni per ciclo di clock.

La sequenza di esecuzione dell'esempio è mostrata nella Figura 4.43, dove la prima colonna specifica il ciclo e la seconda il numero dell'istruzione. La terza colonna elenca le istruzioni decodificate, la quarta specifica le istruzioni che vengono lanciate (con un massimo di due per ciclo di clock). L'ultima colonna contiene l'istruzione che è stata ritirata (completata) durante il ciclo corrispondente. Si tenga presente che in questo esempio si richiede che sia il lancio sia l'esecuzione avvengano in ordine, cioè l'istruzione $k + 1$ non può essere lanciata prima che sia stata lanciata l'istruzione k , e l'istruzione $k + 1$ non può essere completata (e la scrittura del risultato nel registro apposito non può avvenire) finché non viene completata l'istruzione k . Le altre sedici colonne saranno presentate tra breve.

C	I	Decodificata	L	R	Registri letti								Registri scritti								
					0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	
1	1	R3=R0*R1	1		1	1								1							
	2	R4=R0+R2	2		2	1	1							1	1						
2	3	R5=R0+R1	3		3	2	1							1	1	1					
	4	R6=R1+R4	-		3	2	1							1	1	1					
3					3	2	1							1	1	1					
4					1	2	1	1						1	1						
					2	1	1							1							
					3																
5					4			1	1	1							1				
	5	R7=R1*R2	5			2	1	1							1	1					
6	6	R1=R0-R2	-		2	1		1							1	1					
7					4		1	1									1				
8					5																
9	7	R3=R3*R1	6		1		1							1							
10					-	1	1							1							
11					6																
12	8	R1=R4+R4	7			1		1							1						
13					-	1	1							1							
14						1		1						1							
15						7															
16						8			2					1							
17									2					1							
18						8															

Figura 4.43 CPU superscalare che lancia e ritira le istruzioni in ordine (C = ciclo; L = lanciata; R = ritirata).

Dopo aver decodificato un'istruzione, l'unità di decodifica deve decidere se questa può essere lanciata immediatamente o meno. Per prendere la decisione, l'unità di decodifica deve conoscere lo stato di tutti i registri. Se, per esempio, l'istruzione corrente richiede un registro il cui valore non è stato ancora computato, l'istruzione corrente non può essere lanciata e la CPU è costretta ad attendere.

Manteniamo traccia dell'uso dei registri usando uno strumento chiamato **scoreboard** ("tabella dei punteggi"), utilizzato per la prima volta nel CDC 6600. Questa tabella è provvista di un piccolo contatore per ciascun registro che rappresenta il numero d'istruzioni attualmente in esecuzione che richiedono quel particolare registro come sorgente di un operando. Se, poniamo, sono ammesse al massimo 15 istruzioni in esecuzione simultanea, allora sarà sufficiente un contatore di 4 bit. Quando un'istruzione è lanciata

si incrementano le voci dello scoreboard corrispondenti ai registri dei suoi operandi, e quando un'istruzione è ritirata, le voci vengono decrementate.

Lo scoreboard contiene anche contatori per tener traccia dei registri usati come destinazioni. Poiché è permessa una sola scrittura alla volta, tali contatori possono essere lunghi un solo bit. Le ultime sedici colonne sulla destra della Figura 4.43 mostrano lo scoreboard.

Nelle macchine reali lo scoreboard registra anche l'uso delle unità funzionali, per evitare il lancio di un'istruzione per la quale non vi siano unità funzionali disponibili. Per semplicità, qui consideriamo il caso in cui ci siano sempre unità funzionali appropriate disponibili, così possiamo omettere le unità funzionali dallo scoreboard.

La prima riga della Figura 4.43 mostra I1 (istruzione 1), che moltiplica R0 e R1 e pone il risultato in R3. Dal momento che nessuno di questi registri è già in uso, l'istruzione viene lanciata e lo scoreboard è aggiornato con l'annotazione che R0 e R1 sono usati in lettura e R3 in scrittura. Nessuna delle istruzioni successive può scrivere in alcuno di questi registri o leggere il contenuto di R3 finché I1 non venga ritirata. Trattandosi di una moltiplicazione, l'istruzione verrà completata al termine del ciclo 4. I valori delle righe dello scoreboard riflettono lo stato dei registri al lancio delle diverse istruzioni. Le caselle vuote rappresentano il valore 0 dei contatori.

Poiché la macchina dell'esempio è una macchina superscalare che può lanciare due istruzioni per ciclo, la seconda istruzione viene lanciata anch'essa durante il ciclo 1. Tale istruzione somma R0 e R1, salvando il risultato in R4. Per stabilire se un'istruzione può essere lanciata si applicano le regole seguenti:

1. se un operando è in fase di scrittura, non lanciare (dipendenza RAW);
2. se il registro del risultato è in lettura, non lanciare (dipendenza WAR);
3. se il registro del risultato è in scrittura, non lanciare (dipendenza WAW).

Abbiamo già parlato di dipendenza RAW, che si verifica quando un'istruzione deve usare come operando il risultato di un'altra istruzione non ancora completata. Gli altri due tipi di dipendenze sono meno gravi: si tratta essenzialmente di conflitti di risorse. In una **dipendenza di tipo WAR** (Write After Read) un'istruzione cerca di sovrascrivere un registro che un'altra precedente potrebbe non aver terminato di leggere, e la **dipendenza WAW** (Write After Write) è simile. Entrambe possono essere evitate richiedendo che la seconda istruzione salvi il suo risultato da qualche altra parte (magari temporaneamente). Un'istruzione può essere lanciata, se non si verifica nessuna delle tre dipendenze e l'unità funzionale di cui ha bisogno è disponibile. Nel nostro caso l'istruzione 2 usa il registro R0 che è in lettura da parte di un'altra istruzione in corso di svolgimento, ma questa sovrapposizione è consentita e così l'istruzione viene lanciata. Analogamente, la terza istruzione può essere lanciata durante il ciclo 2.

Veniamo ora all'istruzione 4, che richiede l'uso di R4. Sfortunatamente notiamo che, alla riga 3, il registro R4 è in fase di scrittura. In questo caso ci troviamo di fronte a una dipendenza RAW, perciò l'unità di decodifica va in stallo finché R4 non torni disponibile. Durante l'attesa, l'unità smette di prelevare istruzioni dall'unità di fetch. Quando il buffer dell'unità di fetch si riempie, l'unità smette di eseguire il prefetch.

È utile notare che l'istruzione seguente secondo l'ordine del programma, cioè la 5, non causa alcun conflitto con le istruzioni in corso di svolgimento. Se l'architettura non richiedesse il lancio delle istruzioni in ordine, sarebbe possibile decodificare e lanciare l'istruzione 5.

Osserviamo quindi che cosa succede durante il ciclo 3. L'istruzione 2 termina alla fine del ciclo 3, trattandosi di un'addizione (due cicli). Sfortunatamente però non può essere ritirata (liberando R4 per l'uso da parte dell'istruzione 4). Perché? La ragione è che questa architettura richiede anche il ritiro ordinato delle istruzioni. Perché? Quale problema potrebbe sorgere dal salvare il risultato in R4 e marcarlo come disponibile?

La risposta è sottile e al contempo importante. Se le istruzioni possono essere completate fuori sequenza, in caso venisse sollevato un interrupt, sarebbe molto difficile salvare lo stato della macchina per poterlo ripristinare successivamente. In particolare, non sarebbe possibile stabilire che siano state eseguite tutte le istruzioni fino a un certo indirizzo e non quelle successive. Questa capacità è detta **interrupt preciso** ed è una caratteristica auspicabile in una CPU (Moudgil e Vassiliadis, 1996). Il ritiro delle istruzioni fuori sequenza rende gli interrupt imprecisi, ed è questo il motivo per cui alcune macchine completano le istruzioni in ordine.

Tornando all'esempio, alla fine del ciclo 4 le tre istruzioni in esecuzione possono essere ritirate, e l'istruzione 4 può essere infine lanciata durante il ciclo 5, insieme alla 5 appena decodificata. Ogniqualvolta un'istruzione viene ritirata, l'unità di decodifica deve controllare se c'è un'istruzione in stallo che può essere lanciata.

Durante il ciclo 6 l'istruzione 6 è in stallo perché deve scrivere nel registro R1 che è occupato, e la sua esecuzione comincia solo al ciclo 9. Il completamento dell'intera sequenza delle otto istruzioni richiede diciotto cicli a causa di molte dipendenze, nonostante l'hardware sia capace di lanciare due istruzioni a ogni ciclo. D'altra parte, si noti che scorrendo la colonna *L* della Figura 4.43 si evince come tutte le istruzioni siano state lanciate in ordine; la colonna *R* mostra come siano tutte state ritirate in ordine.

Prendiamo ora in esame un'altra architettura che ammette l'esecuzione fuori sequenza. Ora le istruzioni possono essere sia lanciate sia ritirate fuori sequenza. La stessa sequenza di otto istruzioni è mostrata nella Figura 4.44, laddove sono ora consentiti il lancio e il ritiro fuori sequenza.

La prima differenza si verifica al ciclo 3. Anche se l'istruzione 4 è in stallo, è permesso decodificare e lanciare la 5 visto che non è in conflitto con nessuna istruzione in esecuzione. A ogni modo, la possibilità di saltare alcune istruzioni causa un nuovo problema.

Si immagini che l'istruzione 5 utilizzi un operando calcolato dall'istruzione 4, l'istruzione saltata; con l'attuale scoreboard non lo avremmo notato. Bisogna quindi estendere la tabella per tenere traccia delle scritture effettuati dalle istruzioni saltate. A tal fine è possibile aggiungere un bit per registro (non mostrati nella figura), per annotare le scritture effettuati dalle istruzioni in stallo. La regola per il lancio delle istruzioni deve ora essere estesa per prevenire il lancio delle istruzioni i cui operandi sono il risultato di un'istruzione precedente che è stata saltata.

Tornando alle istruzioni 6, 7 e 8 nella Figura 4.43, notiamo che la 6 restituisce un valore in R1 che è richiesto dall'istruzione 7. Notiamo anche che tale valore non è mai

più richiesto perché l'istruzione 8 sovrascrive R1. Non c'è motivo di usare R1 per contenere il risultato dell'istruzione 6. Per di più R1 è una pessima scelta come registro intermedio, nonostante sia ragionevole per un compilatore o per un programmatore abituato all'idea di esecuzione sequenziale senza sovrapposizione d'istruzioni.

C	I	Decodificata	L	R	Registri letti							Registri scritti								
					0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	$R3=R0*R1$	1	2	1	1							1	1	1	1	1	1	1	
	2	$R4=R0+R2$			2	1	1													
2	3	$R5=R0+R1$	3	-	3	2	1						1	1	1	1	1	1	1	
	4	$R6=R1+R4$			3	2	1													
3	5	$R7=R1*R2$	5	6	3	3	2						1	1	1	1	1	1	1	
	6	$S1=R0-R2$			4	3	3													
4	7	$R3=R3*S1$	4	8	3	4	2		1				1	1	1	1	1	1	1	
	8	$S2=R4+R4$			3	4	2		1											
					1	2	3	2	3											
					3	1	2	2	3											
5			6		2	1		3					1						1	1
6			7		2	1	1	3					1	1					1	1
				4	1	1	1	2					1	1					1	1
					5		1	2					1	1					1	1
				8		1							1							
7							1							1						
8								1							1					
9			7																	

Figura 4.44 Operazioni di una CPU superscalare che lancia e ritira le istruzioni fuori sequenza (C = ciclo; L = lanciata; R = ritirata).

Nella Figura 4.44 introduciamo una nuova tecnica per risolvere questo problema: la **rinomina dei registri**. Una saggia unità di decodifica preferirà all'uso di R1 nell'istruzione 6 (ciclo 3) e 7 (ciclo 4) l'uso di un registro segreto, S1, invisibile al programmatore. In questo modo l'istruzione 6 può essere lanciata in modo concorrente con la 5. Le CPU moderne hanno sovente dozzine di registri segreti destinati alla rinomina di registri. Questa tecnica riesce in genere a eliminare le dipendenze WAR e WAW.

In corrispondenza dell'istruzione 8 usiamo ancora la rinomina dei registri. Questa volta R1 è rinominato come S2 di modo che l'addizione possa cominciare prima che R1 sia libero, alla fine del ciclo 6. Nel caso in cui R1 fosse a questo punto la vera destinazione del risultato, il contenuto di S2 potrebbe sempre essere ricopiato in R1 in tempo utile. Oppure, soluzione ancora migliore, è possibile rinominare le sorgenti di tutte le istruzioni future che avranno bisogno di R1 alla locazione dove il suo valore è effettivamente

È utile notare che l'istruzione seguente secondo l'ordine del programma, cioè la 5, non causa alcun conflitto con le istruzioni in corso di svolgimento. Se l'architettura non richiedesse il lancio delle istruzioni in ordine, sarebbe possibile decodificare e lanciare l'istruzione 5.

Osserviamo quindi che cosa succede durante il ciclo 3. L'istruzione 2 termina alla fine del ciclo 3, trattandosi di un'addizione (due cicli). Sfortunatamente però non può essere ritirata (liberando R4 per l'uso da parte dell'istruzione 4). Perché? La ragione è che questa architettura richiede anche il ritiro ordinato delle istruzioni. Perché? Quale problema potrebbe sorgere dal salvare il risultato in R4 e marcarlo come disponibile?

La risposta è sottile e al contempo importante. Se le istruzioni possono essere completeate fuori sequenza, in caso venisse sollevato un interrupt, sarebbe molto difficile salvare lo stato della macchina per poterlo ripristinare successivamente. In particolare, non sarebbe possibile stabilire che siano state eseguite tutte le istruzioni fino a un certo indirizzo e non quelle successive. Questa capacità è detta **interrupt preciso** ed è una caratteristica auspicabile in una CPU (Moudgil e Vassiliadis, 1996). Il ritiro delle istruzioni fuori sequenza rende gli interrupt imprecisi, ed è questo il motivo per cui alcune macchine completano le istruzioni in ordine.

Tornando all'esempio, alla fine del ciclo 4 le tre istruzioni in esecuzione possono essere ritirate, e l'istruzione 4 può essere infine lanciata durante il ciclo 5, insieme alla 5 appena decodificata. Ogniqualvolta un'istruzione viene ritirata, l'unità di decodifica deve controllare se c'è un'istruzione in stallo che può essere lanciata.

Durante il ciclo 6 l'istruzione 6 è in stallo perché deve scrivere nel registro R1 che è occupato, e la sua esecuzione comincia solo al ciclo 9. Il completamento dell'intera sequenza delle otto istruzioni richiede diciotto cicli a causa di molte dipendenze, nonostante l'hardware sia capace di lanciare due istruzioni a ogni ciclo. D'altra parte, si noti che scorrendo la colonna *L* della Figura 4.43 si evince come tutte le istruzioni siano state lanciate in ordine; la colonna *R* mostra come siano tutte state ritirate in ordine.

Prendiamo ora in esame un'altra architettura che ammette l'esecuzione fuori sequenza. Ora le istruzioni possono essere sia lanciate sia ritirate fuori sequenza. La stessa sequenza di otto istruzioni è mostrata nella Figura 4.44, laddove sono ora consentiti il lancio e il ritiro fuori sequenza.

La prima differenza si verifica al ciclo 3. Anche se l'istruzione 4 è in stallo, è permesso decodificare e lanciare la 5 visto che non è in conflitto con nessuna istruzione in esecuzione. A ogni modo, la possibilità di saltare alcune istruzioni causa un nuovo problema.

Si immagini che l'istruzione 5 utilizzi un operando calcolato dall'istruzione 4, l'istruzione saltata; con l'attuale scoreboard non lo avremmo notato. Bisogna quindi estendere la tabella per tenere traccia delle scritture effettuati dalle istruzioni saltate. A tal fine è possibile aggiungere un bit per registro (non mostrati nella figura), per annotare le scritture effettuati dalle istruzioni in stallo. La regola per il lancio delle istruzioni deve ora essere estesa per prevenire il lancio delle istruzioni i cui operandi sono il risultato di un'istruzione precedente che è stata saltata.

Tornando alle istruzioni 6, 7 e 8 nella Figura 4.43, notiamo che la 6 restituisce un valore in R1 che è richiesto dall'istruzione 7. Notiamo anche che tale valore non è mai

più richiesto perché l'istruzione 8 sovrascrive R1. Non c'è motivo di usare R1 per contenere il risultato dell'istruzione 6. Per di più R1 è una pessima scelta come registro intermedio, nonostante sia ragionevole per un compilatore o per un programmatore abituato all'idea di esecuzione sequenziale senza sovrapposizione d'istruzioni.

C	I	Decodificata	L	R	Registri letti							Registri scritti								
					0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	$R3=R0*R1$	1	2	1	1								1						
	2	$R4=R0+R2$			2	1	1							1	1					
2	3	$R5=R0+R1$	3	-	3	2	1							1	1	1				
	4	$R6=R1+R4$			3	2	1							1	1	1				
3	5	$R7=R1*R2$	5	6	3	3	2							1	1	1			1	
	6	$S1=R0-R2$			4	3	3							1	1	1			1	
4	7	$R3=R3*S1$	4	8	3	4	2		1					1	1	1	1	1	1	
	8	$S2=R4+R4$			3	4	2		3					1	1	1	1	1	1	
5			6	7	1	2	3	2	3					1					1	1
					2	1	2	2	3											
6			6	7	2	1	1	3						1	1			1	1	
					4	1	1	1	2					1	1				1	
7			7	8	1	2	1	2						1	1					
					8	1														
8			7					1							1					
								1								1				

Figura 4.44 Operazioni di una CPU superscalare che lancia e ritira le istruzioni fuori sequenza (*C* = ciclo; *L* = lanciata; *R* = ritirata).

Nella Figura 4.44 introduciamo una nuova tecnica per risolvere questo problema: la **rinomina dei registri**. Una saggia unità di decodifica preferirà all'uso di R1 nell'istruzione 6 (ciclo 3) e 7 (ciclo 4) l'uso di un registro segreto, S1, invisibile al programmatore. In questo modo l'istruzione 6 può essere lanciata in modo concorrente con la 5. Le CPU moderne hanno sovente dozzine di registri segreti destinati alla rinomina di registri. Questa tecnica riesce in genere a eliminare le dipendenze WAR e WAW.

In corrispondenza dell'istruzione 8 usiamo ancora la rinomina dei registri. Questa volta R1 è rinominato come S2 di modo che l'addizione possa cominciare prima che R1 sia libero, alla fine del ciclo 6. Nel caso in cui R1 fosse a questo punto la vera destinazione del risultato, il contenuto di S2 potrebbe sempre essere ricopiatò in R1 in tempo utile. Oppure, soluzione ancora migliore, è possibile rinominare le sorgenti di tutte le istruzioni future che avranno bisogno di R1 alla locazione dove il suo valore è effettiva-

mente memorizzato. In ogni caso, con questa tecnica l'addizione dell'istruzione 8 può essere lanciata più prontamente.

Su molte macchine reali il meccanismo di rinomina è profondamente connaturato con il modo in cui sono organizzati i registri: ci sono molti registri segreti e una tabella che mette in corrispondenza i registri visibili al programmatore con quelli segreti. Così il registro usato effettivamente al posto di R0 viene individuato, per esempio, guardando alla voce di posto 0 nella tabella delle corrispondenze. In questo modo non c'è un vero registro R0, ma solo un legame tra il nome R0 e uno dei registri segreti. Al fine di evitare le dipendenze tale legame cambia frequentemente nel corso dell'esecuzione.

Dalla quarta colonna della Figura 4.44 si evince che le istruzioni non sono state lanciate né ritirate in ordine. La conclusione di questo esempio è semplice: attraverso l'utilizzo dell'esecuzione fuori sequenza e della rinomina di registri è stato possibile abbreviare il calcolo di un fattore due.

4.5.4 Esecuzione speculativa

Nelle sezioni precedenti abbiamo introdotto il concetto di riordino delle istruzioni, allo scopo di migliorare le prestazioni. Sebbene non sia stato dichiarato in modo esplicito, finora l'attenzione è stata rivolta al riordinamento che avviene all'interno di un singolo **blocco elementare**. Passiamo adesso a un'analisi più approfondita di questo aspetto.

I programmi sono suddivisibili in blocchi elementari, ciascuno dei quali è costituito da una sequenza lineare d'istruzioni con un punto d'ingresso all'inizio e un punto di uscita alla fine. Un blocco elementare non contiene al suo interno alcuna struttura di controllo (come i costrutti *if* o *while*) di modo che la sua traduzione in linguaggio macchina non presenta nessuna diramazione. Il collegamento fra diversi blocchi elementari è costituito dalle strutture di controllo.

Un programma in questa forma può essere rappresentato attraverso un grafo orientato (Figura 4.45). In questo esempio le somme dei cubi dei numeri pari e dei numeri dispari vengono calcolate separatamente fino a un certo limite e memorizzate rispettivamente in *evensum* e *oddsum*. All'interno di ciascun blocco elementare il riordino delle istruzioni descritto nella sezione precedente funziona in maniera corretta.

Il problema è che la maggior parte dei blocchi elementari è di breve lunghezza e quindi non vi è al loro interno sufficiente parallelismo da poter sfruttare in modo efficace. Di conseguenza, il passo successivo per cercare di utilizzare tutti i cicli disponibili consiste nel permettere al riordinamento di superare i limiti che separano i blocchi elementari. Il guadagno maggiore si ottiene quando un'istruzione potenzialmente lenta può essere spostata più in alto nel grafo di modo che la sua esecuzione possa iniziare in modo anticipato. Questa potrebbe essere un'istruzione *LOAD*, un'operazione in virgola mobile, oppure l'inizio di una lunga catena di dipendenze. La tecnica di anticipare l'esecuzione del codice prima di una diramazione è nota come *slittamento*.

Immaginiamo che nella Figura 4.45 tutte le variabili siano memorizzate nei registri tranne *evensum* e *oddsum* (per mancanza di registri). Potrebbe aver senso spostare le loro istruzioni di *LOAD* all'inizio del ciclo, prima ancora di calcolare *k*, per farle cominciare anticipatamente di modo che i loro valori siano già disponibili nel momento in cui vengano richiesti. Ovviamente, durante ogni iterazione, solo uno di loro sarà effettiva-

mente necessario, mentre l'altra operazione di *LOAD* verrà sprecata; se però sia la cache sia la memoria sono strutturate a pipeline e se vi sono cicli disponibili, potrebbe valerne la pena. L'esecuzione di parti del codice prima ancora di sapere se saranno effettivamente richieste è chiamata **esecuzione speculativa**. Per poter usare questa tecnica sono necessari sia il supporto da parte del compilatore e dell'hardware sia alcune estensioni dell'architettura.

```
evensum = 0;
oddsum = 0;
i = 0;
while (i < limit) {
    k = i * i * i;
    if (((i/2) * 2) == i)
        evensum = evensum + k;
    else
        oddsum = oddsum + k;
    i = i + 1;
}
```

(a)

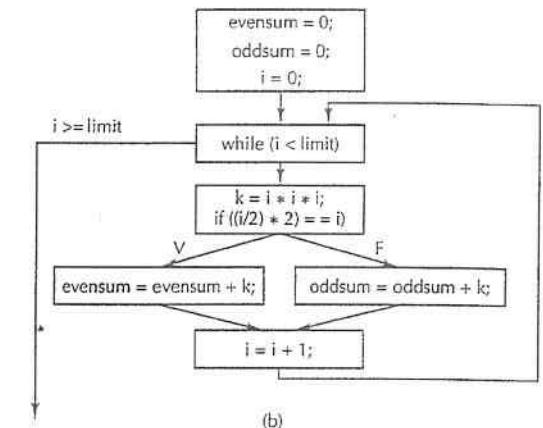


Figura 4.45 (a) Frammento di programma. (b) Grafico a blocchi corrispondente.

È compito del compilatore spostare esplicitamente le istruzioni, dato che un loro riordinamento che scavalchi i limiti dei blocchi elementari è generalmente al di là delle possibilità dell'hardware.

L'esecuzione speculativa introduce alcuni problemi interessanti. Fra questi, è essenziale che nessuna delle istruzioni speculative produca risultati irrevocabili, dato che in un secondo momento potrebbe risultare che non dovevano essere eseguite. Nell'esempio della Figura 4.45 è corretto effettuare il fetch di *evensum* e *oddsum*, e anche calcolare la somma non appena *k* è disponibile (prima del costrutto *if*), ma non è corretto trasferire il risultato in memoria. In sequenze d'istruzioni più complesse, un modo comune per evitare che una parte del codice speculativo scriva nei registri prima ancora di sapere se ciò è voluto, consiste nel rinominare tutti i registri di destinazione utilizzati dal codice speculativo. In questo modo vengono modificati solo i registri di lavoro, così che non sorgano problemi se in ultima istanza il codice risulta non richiesto. Se al contrario il codice è necessario, allora i registri di lavoro vengono copiati nei registri di destinazione corretti. Come si può immaginare, non è semplice utilizzare la tecnica dello scoreboard per tener traccia di tutto ciò, ma può tuttavia essere realizzato disponendo di hardware sufficiente.

In ogni caso esiste un altro problema introdotto dal codice speculativo che non può essere risolto attraverso la semplice rinomina dei registri. Che cosa succede se l'esecu-

zione di un'istruzione speculativa causa un'eccezione? Una situazione fastidiosa, anche se non irrimediabile, si verifica quando un'istruzione LOAD causa un fallimento in una cache con blocchi di grandi dimensioni (per esempio 256 byte) e di una memoria molto più lenta sia della CPU sia della cache. Se un'istruzione di LOAD realmente necessaria pone la macchina in attesa per vari cicli, durante la fase di caricamento del blocco della cache, ciò deve essere considerato normale, dato che la parola è stata effettivamente richiesta. Al contrario, mettere in ballo una macchina durante il fetch di una parola che successivamente non risulterà necessaria è controproducente, al punto che un numero eccessivo di queste "ottimizzazioni" potrebbe rendere la CPU più lenta di quanto non lo sarebbe se non si facesse ricorso alle stesse. Se la macchina è dotata di memoria virtuale (trattata nel Capitolo 6) allora un'istruzione di LOAD speculativa potrebbe addirittura causare un errore di pagina e quindi richiedere un accesso al disco per ottenere la pagina richiesta. Dato che i falsi errori di pagina possono avere un effetto devastante sulle prestazioni è importante evitare che si manifestino.

Una soluzione utilizzata in alcune macchine moderne consiste nell'avere una speciale istruzione speculativa di LOAD, SPECULATIVE-LOAD, che cerca di prelevare la parola dalla cache, ma che non compia alcuna azione se non la trova. Se, nel momento in cui viene richiesto, il valore è presente, allora può essere utilizzato; altrimenti l'hardware deve andare a recuperarlo sul momento. In questo modo il fallimento della cache non produce alcuna penalità se alla fine il valore risulta essere non necessario.

Una situazione decisamente peggiore è rappresentata dalla seguente istruzione

```
if (x > 0) z = y/x;
```

dove x , y e z sono variabili in virgola mobile. Si supponga che tutte le variabili siano prelevate e memorizzate in anticipo nei registri e che venga usata la tecnica dello slittamento del codice per spostare l'esecuzione della lunga operazione in virgola mobile prima del test if. Sfortunatamente x vale 0 e l'eccezione generata dalla divisione per zero fa terminare il programma. Il risultato è che l'uso dell'esecuzione speculativa ha fatto fallire un programma corretto. La cosa peggiore è che ciò è avvenuto nonostante il programmatore avesse previsto il problema e programmato esplicitamente il codice in modo da evitarlo. Difficilmente una simile situazione può rendere felici i programmatore.

Una possibile soluzione consiste nell'avere delle versioni speciali delle istruzioni che possano sollevare delle eccezioni, e nell'aggiungere a ogni registro un bit chiamato **poison bit** ("bit avvelenato"). Quando un'istruzione speculativa speciale fallisce, essa, al posto di causare un'eccezione, assegna il valore 1 al poison bit del registro risultato. Se in un secondo momento un'istruzione regolare utilizza questo registro allora viene sollevata un'eccezione (com'è giusto che sia). In ogni caso, se il risultato non è mai utilizzato, allora il poison bit può essere cancellato senza provocare alcun danno.

4.6 Esempi del livello di microarchitettura

In questo paragrafo mostreremo alcuni brevi esempi di tre processori all'avanguardia e vedremo come vengano impiegati ai loro interno i concetti analizzati nel corso di questo capitolo. Dato che le macchine reali sono estremamente complesse (contengono milioni

di porte logiche) la loro descrizione sarà necessariamente limitata agli aspetti essenziali. Le macchine in esame sono le solite usate finora: Core i7, OMAP4430 e ATmega168.

4.6.1 Microarchitettura della CPU Core i7

Dall'esterno il Core i7 appare come una macchina CISC tradizionale, dotata di processori che supportano un insieme d'istruzioni molto esteso e poco maneggevole, con operazioni su interi a 8, 16 e 32 bit e operazioni in virgola mobile a 32 e 64 bit. I registri visibili sono solo otto e sono tutti diversi tra loro. La lunghezza delle istruzioni varia da 1 a 17 byte. In breve, si tratta di un'architettura ereditata dal passato che sembra fare ogni cosa in modo errato.

Ciononostante il Core i7 contiene al suo interno un nucleo RISC moderno, semplice, efficace e altamente strutturato a pipeline; la velocità di clock è estremamente elevata ed è presumibile che essa continuerà ad aumentare anche nei prossimi anni. È sorprendente come gli ingegneri della Intel siano riusciti a costruire un processore moderno per implementare un'architettura di vecchia concezione. In questo paragrafo studieremo il funzionamento dell'architettura Core i7.

Descrizione della microarchitettura Sandy Bridge del Core i7

La microarchitettura Core i7, chiamata microarchitettura Sandy Bridge, rappresenta un notevole perfezionamento delle precedenti microarchitetture Intel, tra cui P6 e P4. La Figura 4.46 mostra una visione semplificata di questa architettura.

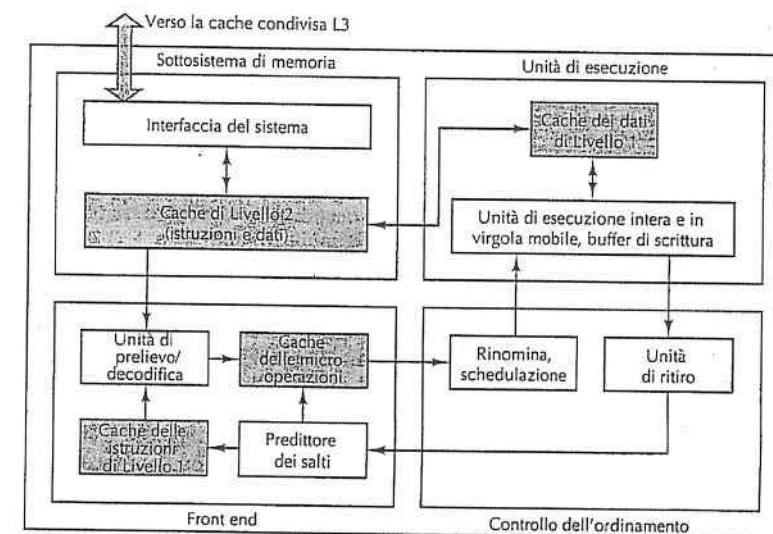


Figura 4.46 Diagramma a blocchi della microarchitettura Sandy Bridge del Core i7.

Il Core i7 è costituito da quattro parti principali: il sottosistema di memoria, il *front end*, il controllo dell'esecuzione fuori sequenza e le unità esecutive. Passiamo ad analizzare questi componenti in senso antiorario a partire da quello situato in alto a sinistra nella Figura 4.46.

Ogni processore del Core i7 contiene un sottosistema di memoria con una cache L2 (di livello 2) unificata e la logica per l'accesso alla cache L3 (di livello 3). Tutti i processori condividono un'unica grande cache L3, l'ultima prima di lasciare il chip della CPU e intraprendere il lungo viaggio verso la RAM esterna attraverso il bus di memoria. Le cache L2 del Core i7 sono di 256 KB e sono organizzate come cache associative a 8 vie con linee di cache da 64 byte. La dimensione della cache L3 varia da 1 a 20 MB: pagando di più si ottiene una cache più grande. Indipendentemente dalla dimensione, la L3 è organizzata come cache associativa a 12 vie con linee da 64 byte. Se un accesso alla L3 fallisce, la richiesta viene inviata alla RAM esterna attraverso il bus DDR3.

Alla cache L1 sono associate due unità di prefetch (non mostrate nella figura) che tentano di prelevare i dati da livelli di memoria più bassi e copiarli nella L1 prima che siano richiesti. Una di queste unità rileva i casi in cui il processore sta prelevando dalla memoria un flusso sequenziale di dati. In questi casi, l'unità preleva il blocco di memoria successivo. Una seconda e più sofisticata unità di prefetch tiene traccia della sequenza di indirizzi utilizzati dalle istruzioni di lettura e scrittura dello specifico programma. Se questi indirizzi si succedono con regolarità (per esempio, 0x1000... 0x1020... 0x1040...) l'unità preleva in anticipo il prossimo elemento a cui con una certa probabilità il programma farà accesso. Questa tecnica funziona benissimo con programmi che operano su vettori.

Il sottosistema di memoria nella Figura 4.46 è connesso sia al front end che alla cache dati L1. Il front end è il responsabile del prelievo delle istruzioni dal sottosistema di memoria, della loro decodifica in micro operazioni di tipo RISC e della loro memorizzazione in due cache istruzioni. Tutte le istruzioni prelevate vengono poste nella cache istruzioni di livello 1, delle dimensioni di 32 KB, organizzata come cache associativa con blocchi da 64 byte. Le istruzioni prelevate dalla cache L1 vengono passate ai decodificatori che determinano la sequenza di micro-operazioni che la pipeline deve eseguire per implementarle. Questo meccanismo di decodifica colma il divario tra un vecchio insieme di istruzioni CISC e un moderno percorso dati RISC.

Le micro-operazioni ottenute dalla decodifica alimentano la cache delle micro-operazioni, chiamata da Intel cache delle istruzioni di livello 0 (L0). La cache delle micro-operazioni è simile a una tradizionale cache istruzioni, ma è anche dotata di molto spazio "di respiro" per memorizzare la sequenza di micro-operazioni prodotte da ogni singola istruzione. Quando si memorizzano micro-operazioni decodificate al posto delle istruzioni originali non è più necessaria la decodifica delle istruzioni in ordine d'esecuzione. A prima vista si può pensare che questo sistema sia stato ideato per velocizzare la pipeline (e in effetti velocizza il processo di produzione di un'istruzione), ma Intel sostiene che la cache delle micro operazioni è stata aggiunta per ridurre il consumo energetico del front end. Con la cache delle istruzioni in ordine il resto del front end resta per l'80% del tempo sospeso in modalità di risparmio energetico.

Il front end realizza anche la predizione dei salti. Il predittore dei salti deve indovinare quando il flusso d'esecuzione di un programma interrompe un andamento puramente sequenziale e deve farlo molto prima che l'istruzione di salto sia eseguita. Il predittore dei salti del Core i7 è degno di nota. Purtroppo per noi, le specifiche dei predittori di salti sono, nella maggioranza dei casi, mantenute gelosamente segrete, perché le prestazioni di un predittore sono spesso la componente più critica nella velocità complessiva di un progetto. Maggiore è la precisione della predizione che i progettisti riescono a ottenere da ogni micrometro quadrato di silicio, migliori saranno le prestazioni dell'intero sistema. Alla luce di ciò, le società mantengono sotto chiave questi segreti e minacciano di perseguire penalmente qualunque dipendente che decidesse di rendere noti questi gioielli di conoscenza. A noi basta dire che tutti i predittori tengono traccia dell'effetto dei precedenti salti e utilizzano questa informazione per fare predizioni. I dettagli su quali informazioni vengono registrate, come sono memorizzate e come vengono utilizzate costituiscono proprio la parte top secret del progetto. Dopo tutto, se si avesse un modo fantastico per predire il futuro, probabilmente non lo si pubblicherebbe sul Web per farlo conoscere al mondo intero.

Le istruzioni sono trasferite dalla cache delle micro-operazioni allo schedulatore fuori-sequenza secondo l'ordine del programma, ma non sono necessariamente lanciate in quell'ordine. Quando lo schedulatore incontra una micro-operazione che non può essere eseguita, la trattiene, pur continuando a processare il flusso d'istruzioni per lanciare le istruzioni successive per cui sono disponibili tutte le risorse (registri, unità funzionali, e così via). Anche la rinomina dei registri viene effettuata in questa fase per permettere alle istruzioni con dipendenza WAR o WAW di continuare senza alcun ritardo.

Sebbene le istruzioni possano essere lanciate fuori sequenza, i requisiti dell'architettura Core i7 per gestire le interruzioni precise fanno sì che le istruzioni ISA debbano essere ritirate (cioè aver reso visibili i loro risultati) in ordine. L'unità di ritiro gestisce questo compito.

Nel back end del processore si trovano le unità di esecuzione che processano istruzioni su interi, in virgola mobile e specializzate. Sono presenti varie unità di esecuzione che funzionano in parallelo. Esse ottengono i loro dati dall'insieme dei registri (*register file*) e dalla cache di dati L1.

Pipeline Sandy Bridge del Core i7

Nella Figura 4.47 è mostrata una versione semplificata della microarchitettura Sandy Bridge in cui è rappresentata anche la pipeline. Nella parte alta dell'immagine si trova il front end, il cui compito è quello di prelevare istruzioni dalla memoria e prepararle per l'esecuzione. Il front end è alimentato dalle nuove istruzioni x86 che provengono dalla cache delle istruzioni L1 e decodifica queste istruzioni in micro-operazioni per memorizzarle nella cache delle micro-operazioni, che ne può contenere circa 1,5K. Le prestazioni fornite da una cache di queste dimensioni sono paragonabili a quelle di una cache convenzionale L0 della dimensione di 6 KB. La cache delle micro-operazioni mantiene, all'interno di ogni linea della traccia, gruppi di sei micro-operazioni. Se la sequenza di micro-operazioni è più lunga, è possibile collegare fra loro più linee della traccia.

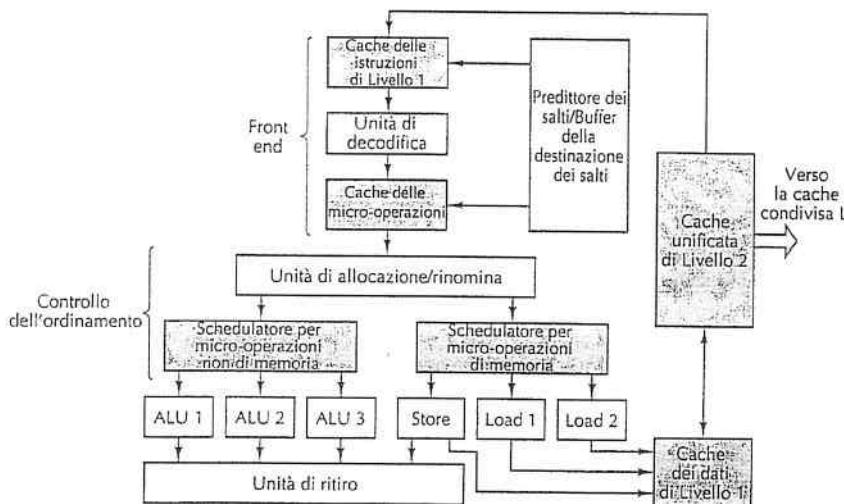


Figura 4.47 Vista semplificata del percorso dati del Corei7.

Se l'unità di decodifica incontra un salto condizionale, cerca nel predittore dei salti la destinazione predetta. Il predittore dei salti contiene la storia di tutte le diramazioni incontrate in passato e la utilizza per indovinare se un nuovo salto condizionale dovrà essere eseguito oppure no. È proprio per queste operazioni che si usa l'algoritmo top secret.

Se l'istruzione di diramazione non è presente nella tabella, si utilizza la predizione statica. Quando si incontra un salto all'indietro si assume che faccia parte di un ciclo e viene quindi effettuato; l'accuratezza di questo tipo di predizione è estremamente elevata. Quando invece si incontra un salto in avanti si assume che faccia parte di un'istruzione if e si sceglie di non effettuare la diramazione. In questo caso l'accuratezza della predizione statica è molto più bassa rispetto a quella dei salti all'indietro.

In caso di salto si consulta il BTB (*Branch Target Buffer*, "buffer della destinazione dei salti") per determinare l'indirizzo di destinazione. Il BTB mantiene l'indirizzo di destinazione dell'ultima occorrenza del salto. Nella maggior parte dei casi questo indirizzo è corretto (di fatto è sempre corretto in caso di salti con spiazzamento costante). I salti indiretti, come quelli utilizzati dalle chiamate di funzioni virtuali o dal costrutto switch del linguaggio C++, hanno molteplici destinazioni. Ecco che in questi casi le predizioni del BTB possono essere errate.

La seconda parte della pipeline, l'unità di controllo dell'esecuzione fuori sequenza, è alimentata dalla cache delle micro-operazioni. Man mano che le micro-operazioni giungono dal front end, fino a 4 per ogni ciclo, l'unità di allocazione e rinomina ne tiene traccia in una tabella, chiamata ROB (*ReOrder Buffer*, "buffer di riordinamento"), di 168 elementi. Ognuno di questi memorizza lo stato della micro-operazione prima che

essa venga ritirata. L'unità di allocazione e rinomina effettua quindi un controllo per verificare se le risorse richieste dalla micro-operazione sono disponibili. Se lo sono, la micro-operazione viene inserita in una delle due code dello schedulatore: una per le micro-operazioni di memoria e una per tutte le altre. In caso contrario la micro-operazione viene ritardata, ma quelle successive vengono in ogni caso elaborate, portando dunque a un'esecuzione fuori sequenza delle micro-operazioni. L'obiettivo di questa strategia è quello di tenere il più possibile occupate le unità funzionali. In un qualsiasi momento è possibile gestire fino a 154 istruzioni, di cui fino a 64 possono essere di lettura dalla memoria e 36 di scrittura in memoria.

A volte una micro-operazione può rimanere in stallo, poiché deve scrivere in un registro che in quel momento viene letto o scritto da una micro-operazione precedente. Come abbiamo già visto questi conflitti sono chiamati dipendenze WAR e WAV. Attraverso la rinomina della sua destinazione è possibile effettuare lo scheduling della micro-operazione affinché venga immediatamente eseguita; con questa tecnica la scrittura avviene in uno dei 160 registri di lavoro invece che all'interno della destinazione prevista, che risulta essere occupata. Se non ci sono registri disponibili o se la micro-operazione ha una dipendenza RAW (che non può mai essere risolta), l'allocatore annota la natura del problema all'interno dell'elemento della tabella ROB. La micro-operazione viene poi inserita in una delle code di esecuzione nel momento in cui tutte le risorse richieste si rendono disponibili.

Quando le micro-operazioni sono pronte per essere eseguite, l'unità di allocazione e rinomina le inserisce nelle due code dello schedulatore. Queste code inviano le micro-operazioni pronte per l'esecuzione alle sei unità funzionali che seguono:

1. ALU 1 e unità per moltiplicazioni in virgola mobile;
2. ALU 2 e unità per addizioni e sottrazioni in virgola mobile;
3. ALU3, trattamento dei salti e unità per confronti in virgola mobile;
4. istruzioni di memorizzazione;
5. istruzioni di caricamento 1;
6. istruzioni di caricamento 2.

Dato che gli scheduleri e le ALU possono elaborare un'operazione per ciclo, lo schedulatore di un Core i7 a 3 GHz è in grado di emettere 18 miliardi di operazioni al secondo, ma questo livello di throughput non verrà in pratica mai raggiunto. Dal momento che il front end fornisce un massimo di quattro micro-operazioni per ciclo, blocchi da sei micro-operazioni possono essere rilasciati solo per brevi periodi, perché le code dello schedulatore si svuotano rapidamente. Inoltre, ogni unità di memoria impiega quattro cicli per elaborare le operazioni e non è dunque in grado di contribuire costantemente alla velocità di esecuzione di picco. Pur non essendo in grado di saturare completamente le risorse di esecuzione, le unità funzionali forniscono una notevole capacità di esecuzione e per questo motivo l'unità di controllo fuori-sequenza va incontro a non pochi problemi per riuscire a tenerle occupate.

Le tre ALU non sono identiche. La ALU 1 può eseguire tutte le operazioni aritmetiche e logiche, oltre alle moltiplicazioni e alle divisioni. La ALU 2 può invece eseguire

solamente le istruzioni aritmetiche e logiche. La ALU3 può eseguire operazioni aritmetiche e logiche e risolvere i salti. Esistono delle differenze anche tra le due unità in virgola mobile.

La prima può eseguire operazioni aritmetiche in virgola mobile, tra cui le moltiplicazioni, mentre la seconda può eseguire solamente addizioni, sottrazioni e operazioni di spostamento (sempre in virgola mobile).

Le ALU e le unità in virgola mobile sono alimentate da una coppia di banchi di registri (*register file*) di 128 elementi; uno per i valori interi, e l'altro per quelli in virgola mobile. Questi registri forniscono tutti gli operandi per le istruzioni da eseguire e servono anche da deposito per i risultati. Al livello ISA sono visibili otto registri (EAX, EBX, ECX, EDX, e così via), ma, per via della rinomina dei registri, gli otto che contengono effettivamente i “veri” valori variano nel tempo a seconda di come cambia la corrispondenza durante l'esecuzione.

L'architettura Sandy Bridge ha introdotto le istruzioni AVX (*Advanced Vector Extension*) che supportano operazioni parallele a 128 bit sui vettori. Queste operazioni riguardano sia vettori di interi sia vettori di numeri in virgola mobile. Questa nuova estensione ISA raddoppia la dimensione possibile dei vettori rispetto alle precedenti versioni SSE e SSE2. Come può questa architettura implementare operazioni a 256 bit con percorsi dati e unità funzionali di 128 bit? Coordinando abilmente due porte dello schedulatore a 128 bit per creare un'unità funzionale a 256 bit.

La cache dati L1 è strettamente accoppiata al back-end della pipeline Sandy Bridge. Si tratta di una cache di 32 KB che contiene numeri interi, numeri in virgola mobile e altri tipi di dati e, a differenza della cache delle micro-operazioni, non viene decodificata in alcun modo, ma mantiene semplicemente una copia dei byte in memoria. La cache dati L1 è una cache associativa a 8 vie con 64 byte per linea di cache. Si tratta di una cache write-back, il che significa che quando una linea di cache viene modificata, il dirty bit della linea viene asserito, mentre i dati vengono ricopiatati sulla cache L2 quando vengono eliminati dalla cache dati L1. La cache è in grado di gestire due operazioni di lettura e una di scrittura per ogni ciclo di clock. Questi accessi multipli vengono implementati utilizzando il *banking*, che divide la cache in sottocache separate (8 nel caso di Sandy Bridge). Se i tre accessi avvengono su banchi distinti possono procedere in simultanea, altrimenti gli accessi conflittuali al banco, tranne uno, dovranno essere in stallo. Se una parola non è presente nella cache L1, viene inviata una richiesta alla cache L2, che risponde immediatamente oppure recupera la linea di cache dalla cache condivisa L3 e quindi risponde. Nello stesso istante possono aver luogo fino a dieci richieste dalla cache L1 per la cache L2.

Dato che le micro-operazioni vengono eseguite fuori sequenza, le operazioni di scrittura nella cache L1 non sono permesse finché non siano state ritirate tutte le istruzioni che precedono una particolare scrittura. L'unità di ritiro (“completamento”) ha il compito di ritirare in ordine le istruzioni e di tener traccia del punto in cui ci si trova. Se si verifica un interrupt le istruzioni non ancora ritirate vengono annullate; in questo modo il Core i7 gode della proprietà che, quando si verifica un interrupt, esiste un punto in cui tutte le istruzioni precedenti sono state completate e nessuna di quelle successive produce alcun effetto.

Se viene ritirata un'istruzione di memorizzazione mentre qualche istruzione che la precede è ancora in esecuzione, non è possibile aggiornare la cache L1. I risultati vengono quindi inseriti in un buffer per le memorizzazioni pendenti: questo buffer è composto da 36 elementi che corrispondono a 36 possibili memorizzazioni contemporaneamente in esecuzione. In seguito, se un'istruzione di caricamento prova a leggere i dati memorizzati, può esserne restituito il valore contenuto nel buffer per le memorizzazioni pendenti, anche se non è ancora stato aggiornato all'interno della cache L1. Questo processo è chiamato avanzamento store-to-load. Anche se questo meccanismo di avanzamento può sembrare semplice, la sua implementazione è piuttosto complicata perché le scritture che intervengono potrebbero non aver ancora calcolato i loro indirizzi. In questo caso, la microarchitettura non può sapere quale operazione presente nel buffer produrrà il valore necessario. Il processo di determinazione dell'operazione di scrittura che fornisce il valore per la lettura viene chiamato disambiguazione.

A questo punto dovrebbe essere evidente che il Core i7 ha una microarchitettura molto complessa, progettata per poter eseguire il vecchio insieme d'istruzioni Pentium su un moderno nucleo di tipo RISC con una profonda struttura a pipeline. Per raggiungere questo scopo suddivide le istruzioni Pentium in micro-operazioni, le memorizza all'interno di una cache e ne inserisce tre alla volta nella pipeline, per essere eseguite da un insieme di ALU; in condizioni ottimali queste unità sono in grado di eseguire fino a sei microoperazioni per ciclo. Le micro-operazioni sono eseguite fuori sequenza, ma vengono ritirate in ordine e anche la memorizzazione dei risultati all'interno delle cache L1 e L2 avviene in ordine.

4.6.2 Microarchitettura della CPU OMAP4430

Il cuore del processore OMAP4430 è costituito da due processori ARM Cortex A9, microarchitetture ad alte prestazioni che implementano le istruzioni ARM (versione 7). Il Cortex A9, progettato da ARM ltd., è utilizzato in diverse varianti in una vasta gamma di dispositivi integrati. ARM non produce il processore, ma fornisce il progetto ai produttori che vogliono includerlo nei loro SoC (nel nostro caso alla Texas Instruments).

Il processore Cortex A9 è a 32 bit, ha registri e un percorso dati a 32 bit. Come l'architettura interna, anche il bus di memoria è a 32 bit. A differenza del Core i7, il Cortex A9 è una vera e propria architettura RISC; ciò significa che non è necessario adottare per l'esecuzione un complicato meccanismo di conversione di vecchie istruzioni CISC in sequenze di micro-operazioni, perché le istruzioni ARM sono già micro-operazioni. Tuttavia nel corso degli anni sono state aggiunte istruzioni per la grafica e la multimedialità la cui esecuzione richiede la presenza di hardware specializzato.

Descrizione della microarchitettura Cortex A9 dell'OMAP4430

La Figura 4.48 mostra il diagramma a blocchi del Cortex A9. Nel complesso è molto meno complicato della microarchitettura Sandy Bridge del Core i7, perché deve implementare un'architettura ISA decisamente più semplice. Ciononostante alcuni dei componenti chiave sono simili a quelli del Core i7. Le somiglianze sono determinate principalmente dalla tecnologia disponibile, dai limiti imposti dal consumo energetico e da fattori economici. Per esempio, entrambi i progetti adottano gerarchie di cache multili-

vello per andare incontro agli stretti vincoli di costo delle tipiche applicazioni integrate; tuttavia, l'ultimo livello del sistema di memoria cache del Cortex A9 (L2) è solamente di 1MB, significativamente più piccolo rispetto all'ultimo livello di cache (L3) del Core i7, che può essere 20 volte più grande. Le differenze risiedono invece principalmente nella necessità, o meno, di dover realizzare un ponte tra un vecchio sistema d'istruzioni CISC e un moderno nucleo di tipo RISC.

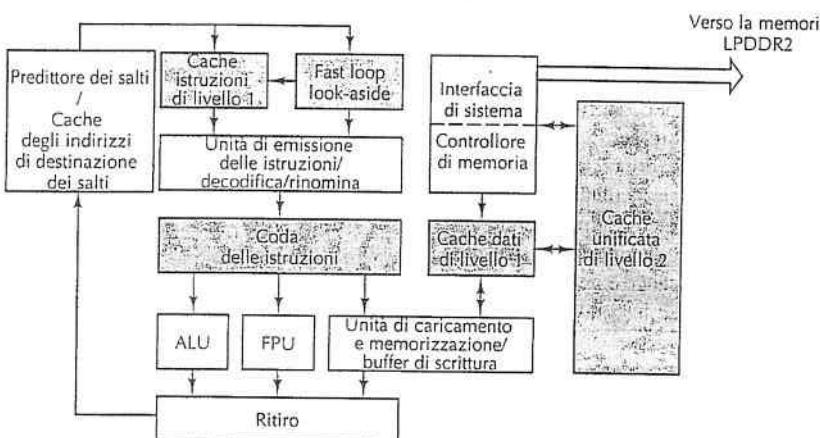


Figura 4.48 Diagramma a blocchi della microarchitettura Cortex A9 dell'OMAP 4430.

Nella parte alta della Figura 4.48 si trova la cache delle istruzioni a 32 KB, di tipo associativo a 4 vie, che utilizza linee di cache da 32 byte. Dato che la maggior parte delle istruzioni ARM sono di 4 byte, questa cache può mantenere circa 8K istruzioni, leggermente più della cache delle micro-operazioni del Core i7.

L'unità di rifornimento delle istruzioni prepara fino a quattro istruzioni per ciclo per l'esecuzione. Nel caso in cui si verifichi un fallimento della cache L1, viene emesso un numero minore d'istruzioni. Quando si incontra un salto condizionale si consulta un predittore di salti con 4K voci per predire se il salto verrà effettuato o meno.

Se la predizione dice che avverrà il salto, viene consultata la cache degli indirizzi di destinazione dei salti, con 1K elementi, per preuire l'indirizzo di destinazione. Inoltre, se il front-end rileva che il programma sta eseguendo un *ciclo stretto* (ovvero un piccolo ciclo non innestato), carica il ciclo nella cache *fast-loop look-aside*. Questa ottimizzazione velocizza il prelievo delle istruzioni e riduce i consumi, perché le cache e i predittori possono restare in modalità di sospensione durante l'esecuzione del ciclo stretto.

L'output dell'unità di emissione delle istruzioni confluisce nei decodificatori, che determinano di quali risorse e di quali input ha bisogno un'istruzione. Come nel caso del Core i7, le istruzioni vengono rinominate dopo la decodifica per eliminare i rischi di WAR che possono rallentare l'esecuzione fuori sequenza. Dopo la rinomina, le istruzio-

ni vengono poste nella coda di consegna delle istruzioni, che le rilascerà, potenzialmente fuori sequenza, quando i loro input saranno pronti per le unità funzionali.

La coda di emissione delle istruzioni invia le istruzioni alle unità funzionali, come mostra la Figura 4.48. L'unità di esecuzione intera contiene due ALU, una piccola pipeline per le istruzioni di salto e il banco dei registri che mantiene i registri ISA e alcuni registri di lavoro. La pipeline del Cortex A9 può eventualmente contenere circuiti di calcolo aggiuntivi che agiscono come unità funzionali. ARM supporta un motore per il calcolo in virgola mobile chiamato VFP e una elaborazione vettoriale SIMD su interi chiamata NEON.

L'unità di caricamento e memorizzazione gestisce varie istruzioni che si riferiscono alla memoria, e ha percorsi dati verso la cache dei dati e il buffer di scrittura. La cache dei dati è una tradizionale cache L1 da 32 KB di tipo associativo a 4 vie e con linee di 32 byte. Il buffer di scrittura mantiene le istruzioni di memorizzazione che non hanno ancora scritto il loro valore nella cache dei dati. Un'istruzione di load in esecuzione proverà prima a prelevare il valore dal buffer di scrittura utilizzando l'avanzamento store-to-load, come nel Core i7, e poi, se il valore non è disponibile nel buffer di scrittura, lo preleverà dalla cache dei dati. Un possibile esito dell'esecuzione di una load è l'indicazione, proveniente dal buffer di scrittura, di una necessaria attesa, dovuta al fatto che una precedente istruzione di scrittura con indirizzo sconosciuto sta bloccando l'esecuzione. Nel caso in cui l'accesso alla cache dei dati L1 fallisca, il blocco di memoria viene prelevato dalla cache L2 unificata. In alcune circostanze il Cortex A9 è in grado di eseguire il prefetching hardware dalla cache L2 nella cache L1, per migliorare le operazioni di lettura e scrittura.

Il chip OMAP4430 contiene anche la logica per il controllo degli accessi a memoria. Tale logica è divisa in due parti: l'interfaccia di sistema e il controllore di memoria. L'interfaccia permette il collegamento alla memoria su un bus LPDDR2 a 32 bit e tutte le richieste di memoria all'ambiente esterno passano attraverso questa interfaccia. Il bus LPDDR2 supporta un indirizzamento a 26 bit verso 8 banchi che restituiscono parole di 32 bit. La memoria centrale può arrivare, teoricamente, alla dimensione di 4GB per ogni canale LPDDR2. L'OMAP ha due di questi canali e può così indirizzare fino a 8 GB di memoria esterna.

Il controllore della memoria mappa indirizzi virtuali a 32 bit in indirizzi fisici a 32 bit. Il Cortex A9 supporta la memoria virtuale (trattata nel Capitolo 6) con pagine di 4 KB. Per accelerare la corrispondenza ci sono delle tabelle speciali, chiamate TLB (*Translation Lookaside Buffers*), per confrontare l'indirizzo virtuale al quale si sta facendo riferimento con quelli referenziati nel passato più recente. Sono presenti due di queste tabelle per gestire la mappatura degli indirizzi di istruzioni e dati.

Pipeline Cortex A9 dell'OMAP4430

Il Cortex A9 dispone di una pipeline a 11 stadi. La Figura 4.49 ne mostra una rappresentazione semplificata, in cui gli 11 stadi sono indicati, sul lato sinistro, da abbreviazioni. Esaminiamo brevemente ogni stadio. Lo stadio *Fel* (Fetch #1) si trova all'inizio della pipeline ed è il punto in cui l'indirizzo della successiva istruzione da prelevare viene utilizzato per indicizzare la cache delle istruzioni e dare il via a una predizione dei salti. Di solito questo indirizzo corrisponde a quello che segue l'istruzione corrente.

Tuttavia questo ordinamento sequenziale può essere alterato per varie ragioni; per esempio quando l'istruzione precedente è una diramazione che secondo la predizione deve essere eseguita, oppure quando occorre gestire un interrupt o un'eccezione. Dato che non è possibile effettuare il prelievo dell'istruzione e la predizione dei salti in un solo ciclo, lo stadio *Fe2* (Fetch #2) offre ulteriore tempo per portare a termine queste operazioni. Nello stadio *Fe3* (Fetch #3) le istruzioni prelevate (al massimo quattro) vengono inserite nella coda delle istruzioni.

Negli stadi *De1* e *De2* (*Decode*, “decodifica”) vengono decodificate le istruzioni. In questo passo vanno determinati gli input di cui le istruzioni hanno bisogno (registri e memoria) e quali risorse richiederanno per la loro esecuzione (unità funzionali). Una volta che la decodifica è completata, le istruzioni entrano nello stadio *Re* (*Rename*, “rinomina”), in cui i registri utilizzati vengono rinominati al fine di eliminare dipendenze WAR e WAW durante l'esecuzione fuori sequenza. Questo stadio contiene la tabella di rinomina, che cataloga quali registri fisici mantengono al momento tutti i registri dell'architettura. Con l'utilizzo di questa tabella ogni registro di input può essere facilmente rinominato. Al registro di output deve essere assegnato un nuovo registro fisico, scelto all'interno di una collezione di registri inutilizzati, che sarà utilizzato dall'istruzione fino al suo completamento.

Successivamente le istruzioni entrano nello stadio *Iss* (*Instruction Issue*, “rilascio dell'istruzione”), in cui vengono inserite nella coda di rilascio delle istruzioni. Questa coda controlla le istruzioni per le quali gli input sono tutti pronti. Al verificarsi di questa condizione, gli input delle istruzioni vengono acquisiti (dal banco dei registri fisici o dal bus bypass) e le istruzioni vengono inviate allo stadio di esecuzione. Come il Core i7, anche il Cortex A9 può emettere le istruzioni in ordine diverso rispetto a quello del programma. In ogni ciclo possono essere rilasciate fino a 4 istruzioni. La scelta delle istruzioni è vincolata dalla disponibilità delle unità funzionali.

Lo stadio *Ex* (*Execute*, “esecuzione”) è quello in cui le istruzioni vengono realmente eseguite. La maggior parte delle istruzioni aritmetiche, booleane e di shift utilizzano le ALU intere e vengono completate in un ciclo, i caricamenti e le memorizzazioni impiegano due cicli (se trovano i valori nella cache L1) e le moltiplicazioni richiedono tre cicli. Lo stadio *Ex* contiene diverse unità funzionali:

1. ALU 1 intera;
2. ALU 2 intera;
3. unità di moltiplicazione;
4. ALU in virgola mobile e vettoriale SIMD (opzionalmente con supporto VFP e NEON);
5. unità di caricamento e memorizzazione.

Nel primo stadio *Ex* vengono anche processate le istruzioni di salto condizionato e ne viene determinata la direzione (salto oppure nessun salto). In caso di errata predizione viene inviato un segnale allo stadio *Fe1* e la pipeline viene svuotata.

Al completamento della loro esecuzione le istruzioni entrano nello stadio *WB* (*WriteBack*, “riscrittura”), in cui ogni istruzione aggiorna immediatamente il banco dei registri fisici. Quando un'istruzione diventa la più vecchia può scrivere i registri risultato nel

banco registri architettonicale. In caso di eccezione o interrupt, sono questi ultimi valori, e non quelli nei registri fisici, a essere resi visibili. Questa azione di memorizzare i registri nel banco architettonicale corrispondente al ritiro nel Core i7. In aggiunta, nello stadio *WB*, ogni istruzione di memorizzazione completa la scrittura dei suoi risultati nella cache L1.

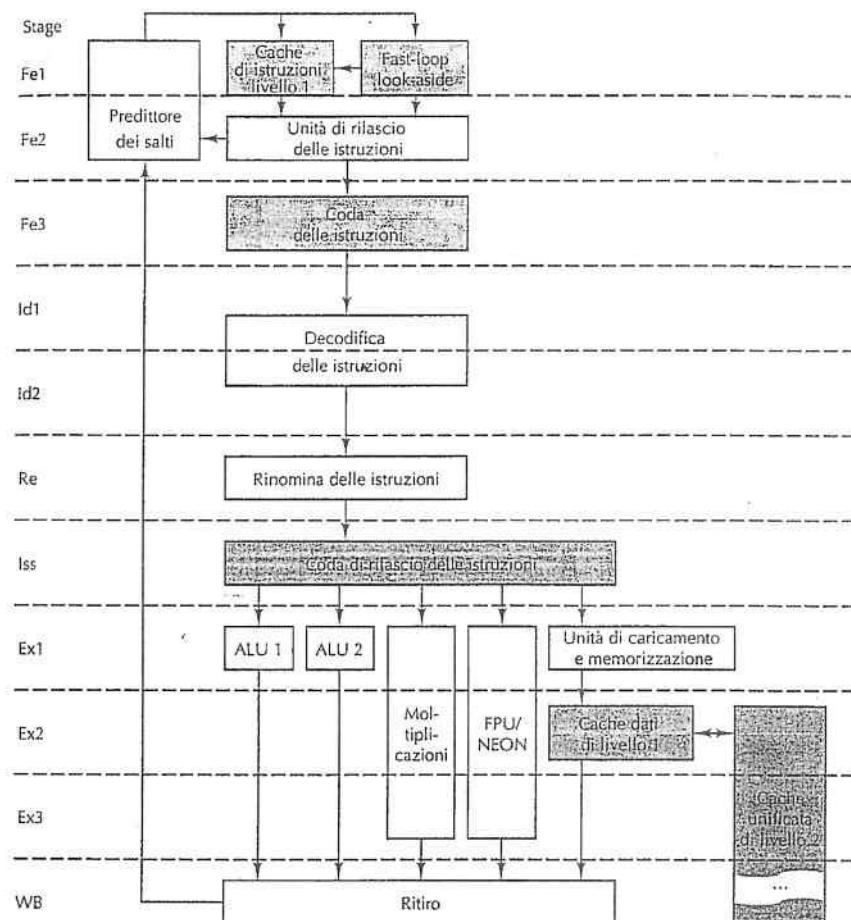


Figura 4.49 Rappresentazione semplificata della pipeline Cortex A9 dell'OMAP4430.

Questa descrizione del Cortex A9 è lontana dall'essere completa, ma permette di avere un'idea sostanzialmente corretta del suo funzionamento e delle differenze che la contraddistinguono rispetto alla microarchitettura del Core i7.

4.6.3 Microarchitettura del microcontrollore ATmega168

Il nostro ultimo esempio è la microarchitettura dell'Atmel ATmega168 (Figura 4.50) che è considerevolmente più semplice delle due precedenti. Il motivo risiede nelle piccole dimensioni del chip e nella sua economicità, vincoli dovuti al suo utilizzo nei sistemi integrati. Uno dei principali obiettivi della progettazione di questo chip era di renderlo economico, non veloce. Economico e Semplice sono sempre dei buoni amici. Economico e Veloce invece non sempre lo sono.

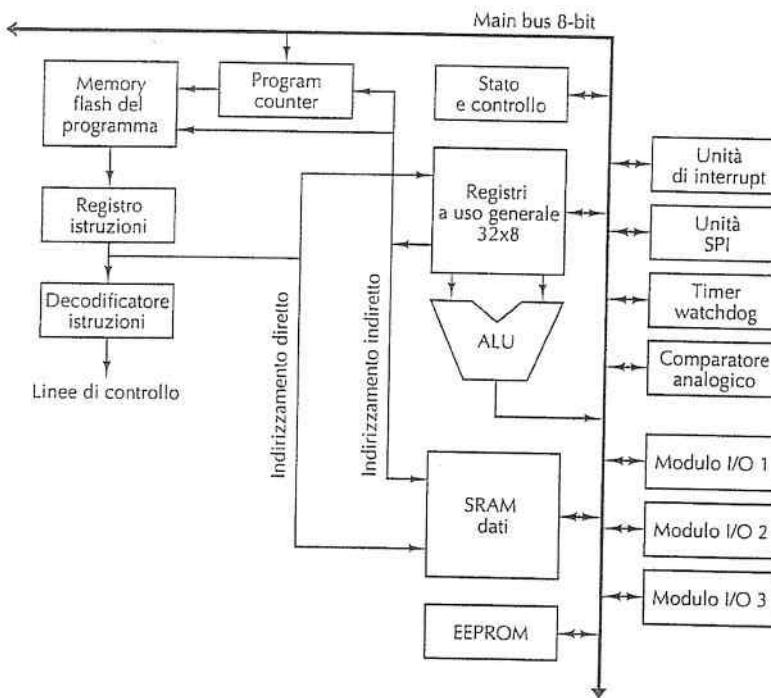


Figura 4.50 Microarchitettura dell'ATmega168.

Il cuore dell'ATmega168 è rappresentato dal bus principale a 8 bit, cui sono collegati i bit dei registri e di stato, le ALU, la memoria e i dispositivi di input/output. Descriviamoli brevemente. Il banco dei registri contiene 32 registri a 8 bit, utilizzati per memorizzare valori temporanei del programma. Il registro di stato e controllo mantiene i codici condizione dell'ultima operazione della ALU (per esempio segno, overflow, negativo, zero, riporto) più un bit che indica se vi è un interrupt pendente. Il program counter contiene l'indirizzo dell'istruzione in esecuzione. Per realizzare una operazione

ALU gli operandi vengono letti dai registri e inviati alla ALU. L'output della ALU può essere scritto in qualsiasi registro scrivibile attraverso il bus principale.

L'ATmega168 ha memorie per dati e istruzioni. La SRAM dei dati è da 1KB, troppo grande per essere interamente indirizzata con un indirizzo a 8 bit sul bus principale. L'architettura AVR permette quindi di costruire indirizzi con coppie consecutive di registri da 8 bit, producendo così indirizzi di 16 bit che permettono di supportare fino a 64 KB di memoria dati. La EEPROM offre fino a 1KB di spazio di memorizzazione non volatile in cui i programmi possono scrivere le variabili che devono essere mantenute anche in assenza di corrente.

Un meccanismo similare viene utilizzato per indirizzare la memoria del programma. In questo caso però 64 KB di spazio non sono sufficienti, nemmeno per sistemi integrati a basso costo. Per permettere una memoria delle istruzioni più grande l'architettura AVR utilizza tre registri di pagina RAM (RAMPX, RAMPY, RAMPZ), ognuno da 8 bit. Il registro di pagina RAM viene concatenato con una coppia di registri per formare un indirizzo di 24 bit, permettendo così uno spazio di indirizzamento per le istruzioni di 16 MB.

Interrompiamoci e riflettiamo un attimo. 64 KB di codice sono troppo pochi per un microcontrollore che deve essere utilizzato per un gioco o per piccole applicazioni. Nel 1964, IBM ha rilasciato il System 360 model 30, che aveva in totale 64 KB di memoria (senza meccanismi per poterla aumentare). Veniva venduto per \$ 250.000, corrispondenti al valore a oggi di circa 2 milioni di dollari. L'ATmega168 costa \$ 1 circa, anche meno se per ordini di grandi quantità. Se si guarda, per esempio, il listino prezzi della Boeing si scopre che il prezzo degli aeroplani non è sceso di 250.000 volte negli ultimi 50 anni; nemmeno quello delle macchine, né il prezzo dei televisori: solo il prezzo dei computer è sceso di così tanto.

In aggiunta, l'ATmega168 ha integrati sul chip un controllore di interrupt, un'interfaccia SPI (*serial port interface*) e dei timer, essenziali per le applicazioni in tempo reale. Ci sono anche tre porte digitali di I/O a 8 bit, che permettono all'ATmega168 di controllare fino a 24 collegamenti esterni verso pulsanti, sensori, luci, attuatori, ecc. È proprio la presenza di queste porte e dei timer che rende possibile l'utilizzo dell'ATmega168 per applicazioni integrate senza bisogno di chip supplementari.

L'ATmega168 è un processore sincrono in cui la maggior parte delle istruzioni, anche se non tutte, viene eseguita in un ciclo di clock. Il processore è dotato di una pipeline a due stadi, stadio di prelievo e stadio di esecuzione, in modo che mentre un'istruzione viene prelevata, l'istruzione precedente viene eseguita. Per eseguire le istruzioni in un ciclo, il ciclo di clock deve far spazio alla lettura dei registri dal banco dei registri, seguita dall'esecuzione dell'istruzione nella ALU, seguita poi dalla riscrittura dei registri nel banco dei registri. Visto che tutte queste operazioni avvengono in un unico ciclo, non vi è alcuna necessità di una logica di bypass o del rilevamento degli stalli. Le istruzioni del programma sono eseguite in ordine, in un unico ciclo e senza sovrapposizioni con altre istruzioni.

Anche se potremmo fornire maggiori dettagli su questo processore, la descrizione data in questo paragrafo e la Figura 4.50 sono sufficienti per avere un'idea di base della sua microarchitettura. L'ATmega168 ha un unico bus principale (per ridurre l'area del

chip), al quale sono collegati un insieme eterogeneo di registri e una varietà di memorie e dispositivi di I/O. Durante ogni ciclo del percorso due operandi vengono letti dal banco dei registri, fatti passare attraverso la ALU e il risultato viene memorizzato all'interno di un registro, proprio come nei calcolatori moderni.

4.7 Confronto tra i7, OMAP4430 e ATmega168

I tre esempi considerati sono molto differenti tra loro, pur avendo dei punti in comune. Il Core i7 è un insieme d'istruzioni CISC datato che gli ingegneri di Intel amerebbero fervidamente poter gettare nella Baia di San Francisco, se ciò non violasse la legge della California sull'inquinamento delle acque. L'OMAP4430 è una vera e propria architettura RISC, con un insieme d'istruzioni semplice e ordinato. L'ATmega168 è invece un semplice processore a 8 bit per le applicazioni integrate. Nel cuore di ognuno ci sono dei registri e una o più ALU che eseguono semplici operazioni aritmetiche e booleans.

Nonostante le ovvie differenze esterne il Core i7 e l'OMAP4430 hanno delle unità di esecuzione abbastanza simili. Le unità di esecuzione di entrambi accettano micro-operazioni che specificano un codice operativo, due registri sorgenti e un registro destinazione. Esse sono in grado di eseguire in ciascun ciclo una micro-operazione; hanno inoltre pipeline con un elevato numero di stadi, effettuano la predizione dei salti e sono dotate di cache separate per i dati e per le istruzioni.

Queste analogie tra i componenti interni non sono casuali e neanche dovuti all'interminabile passaggio degli ingegneri della Silicon Valley da una società a un'altra. Come abbiamo visto nelle architetture di Mic-3 e Mic-4, è facile e naturale costruire un percorso dati a pipeline che prende due registri sorgente, li fa passare attraverso la ALU e ne memorizza il risultato nuovamente in un registro. Nella Figura 4.34 è possibile vedere una rappresentazione grafica di una pipeline di questo tipo. Con la tecnologia attualmente disponibile, questa architettura è la più efficiente.

La differenza principale tra il Core i7 e l'OMAP4430 è nel modo in cui prendono le istruzioni del loro livello ISA e le assegnano alle unità funzionali. Il Core i7 deve suddividere le sue istruzioni CISC per portarle nel formato a tre registri richiesto dall'unità funzionale. Questo è ciò che fa il front end della Figura 4.47: trasforma istruzioni grandi in micro-operazioni dal formato semplice e regolare. L'OMAP4430 non deve effettuare questa conversione, dato che le sue istruzioni native ARM hanno già un formato semplice e regolare. Questo è il motivo per cui la maggior parte dei nuovi ISA sono di tipo RISC: permettono una miglior corrispondenza tra le istruzioni ISA e l'unità di esecuzione interna.

Può essere istruttivo confrontare la nostra architettura finale, il Mic-4, con i due esempi tratti dal mondo reale. Il Mic-4 assomiglia al Core i7, dato che entrambi devono interpretare un insieme d'istruzioni ISA di tipo non-RISC. Per far ciò entrambi suddividono le istruzioni ISA in micro-operazioni composte da un codice operativo, due registri sorgenti e un registro destinazione. In entrambi i casi le micro-operazioni vengono accodate prima di essere eseguite. Mic-4 emette le istruzioni, le esegue e le ritira sempre in ordine, mentre il Core i7 utilizza una politica di emissione in ordine, esecuzione fuori sequenza e ritiro in ordine.

Mic-4 e OMAP4430 non sono effettivamente confrontabili, dato che l'insieme d'istruzioni ISA dell'OMAP4430 è composto da istruzioni RISC (cioè micro-operazioni a tre registri) che non devono essere suddivise. Ognuna di queste istruzioni può essere eseguita, senza alcuna modifica, in un unico ciclo del percorso dati.

Tuttavia, diversamente dal Core i7 e dall'OMAP4430, l'ATmega168 è una macchina molto semplice. È più di tipo RISC che di tipo CISC, dato che la maggior parte delle sue istruzioni può essere eseguita in un singolo ciclo di clock e non deve essere suddivisa. Non sono presenti né pipeline né cache; inoltre l'emissione, l'esecuzione e il ritiro avvengono tutte in ordine. Nella sua semplicità assomiglia a Mic-1.

4.8 Riepilogo

Il cuore di ogni calcolatore è rappresentato dal percorso dati, che contiene dei registri, uno, due o tre bus, e una o più unità funzionali come le ALU e gli shifter. Il ciclo di esecuzione principale consiste nel prelevare operandi dai registri e nello spedirli, utilizzando i bus, alle ALU e alle altre unità funzionali per l'esecuzione. I risultati vengono poi memorizzati nuovamente nei registri.

Il percorso dati può essere controllato da un sequenzializzatore che preleva le micro-istruzioni da una memoria di controllo. Ogni microistruzione contiene dei bit che controllano il percorso dati durante un ciclo e specificano quali operandi selezionare, quale operazione eseguire e che cosa occorre fare con il risultato. Inoltre, ogni microistruzione specifica il proprio successore, di solito in modo esplicito, indicandone l'indirizzo. Alcune microistruzioni modificano questo indirizzo di base calcolandone l'OR con alcuni bit prima di utilizzarlo.

La macchina IJVM è una macchina a stack con codici operativi da 1 byte che permettono di usare lo stack per inserire, estrarre e combinare (cioè addizionare) parole. Nella microarchitettura Mic-1 è stata presentata un'implementazione microprogrammata. Aggiungendo un'unità per il prelievo delle istruzioni, in grado di caricare in anticipo i byte del flusso delle istruzioni, è possibile eliminare un gran numero di riferimenti al contatore d'istruzioni, accelerando notevolmente la macchina.

Esistono diversi modi per progettare il livello di microarchitettura. Occorre compiere molte scelte, come quella tra un'architettura a due bus e una a tre bus, tra i campi codificati oppure decodificati delle microistruzioni, tra l'utilizzo o meno del prefetching, sulla profondità delle pipeline, e molte altre ancora. Mic-1 è una semplice macchina controllata via software, in cui l'esecuzione è sequenziale e non vi è parallelismo. Al contrario Mic-4 è una microarchitettura altamente parallela con una pipeline a sette stadi.

Le prestazioni possono essere migliorate agendo sotto diversi aspetti della microarchitettura. Il modo principale per migliorare le prestazioni consiste nell'uso di memorie cache. Per accelerare i riferimenti alla memoria si usano comunemente cache a corrispondenza diretta e cache set-associative. Anche la predizione dei salti, sia statica sia dinamica, è importante, così come l'esecuzione fuori sequenza e l'esecuzione speculativa.

Le tre macchine di esempio, Core i7, OMAP4430 e ATmega168, hanno una microarchitettura invisible ai programmati del linguaggio assemblativo ISA. Il Core i7 utilizza uno schema complesso per convertire le istruzioni ISA in micro-operazioni, memorizzarle in una cache e alimentare con loro un nucleo RISC superscalare. Questo schema permette inoltre l'esecuzione fuori sequenza, la rinomina dei registri e tutti gli altri meccanismi descritti nel libro che consentono di ottenere anche un ultimo, minimo, miglioramento della velocità dell'hardware. L'OMAP4430 ha una pipeline profonda, ma a parte ciò è relativamente semplice, emettendo, eseguendo e ritirando le istruzioni in ordine. L'ATmega168 è decisamente semplice: è composto da un singolo bus principale al quale sono collegate una manciata di registri e una ALU.

PROBLEMI

- Quali sono i quattro passi utilizzati dalle CPU per eseguire le istruzioni?
- Nella Figura 4.6 il registro del bus B è codificato in un campo a 4 bit, mentre il bus C è rappresentato come una mappa di bit. Perché?
- Nella Figura 4.6 è rappresentato un rettangolo etichettato come "Bit alto". Si disegni il circuito che lo realizza.
- Quando viene abilitato il campo JMPC di una microistruzione, viene calcolato l'OR logico tra MBR e NEXT_ADDRESS, per formare l'indirizzo della microistruzione successiva. Esistono situazioni in cui ha senso che il campo NEXT_ADDRESS valga 0x1FF e che si utilizzi JMPC?
- Supponiamo che nell'esempio della Figura 4.14(a) si aggiunga l'istruzione

$$k = 5;$$

dopo l'istruzione if. Quale dovrebbe essere il nuovo codice assemblativo? Si assuma che il compilatore sia ottimizzato.

- Si forniscano due differenti traduzioni IJVM della seguente istruzione Java:

$$i = k + n + 5;$$

- Si scriva l'istruzione Java che produce il seguente codice IJVM:

```
ILOAD j  
ILOAD n  
ISUB  
BIPUSH 7  
ISUB  
DUP  
IADD  
ISTORE i
```

- Nel testo abbiamo affermato che quando si traduce in binario l'istruzione

If (Z) goto L1; else goto L2

L2 deve trovarsi nelle prime 256 parole della memoria di controllo. Non sarebbe stato ugualmente possibile avere, per esempio, L1 all'indirizzo 0x40 e L2 all'indirizzo 0x140? Si motivi la risposta.

- Nel microprogramma Mic-1, MDR viene copiato all'interno di H, durante if_icmpq3. Dopo qualche linea viene sottratto da TOS per effettuare un test di uguaglianza. Sicuramente sarebbe stato meglio avere in questo punto la seguente istruzione:

if_icmpq3 Z = TOS - MDR; rd

Perché ciò non è stato fatto?

- Si calcoli quanti nanosecondi impiega Mic-1 a 2.5 GHz per eseguire la seguente istruzione Java
 $i = j + k;$
- Si risponda nuovamente alla domanda precedente considerando questa volta Mic-2 a 2.5 GHz. In base al calcolo effettuato quanto dovrebbe impiegare su Mic-2 un programma che viene eseguito da Mic-1 in 100 s?
- Si scriva il microcodice che implementa per Mic-1 l'istruzione JVM POPTWO (che rimuove due parole dalla cima dello stack).
- Su una macchina JVM completa esistono codici operativi speciali per caricare nello stack, senza utilizzare l'istruzione generale ILOAD, le variabili locali da 0 a 3. Come dovrebbe essere modificata IJVM per sfruttare nel modo migliore queste istruzioni?
- L'istruzione ISHR (scorrimento aritmetico a destra su interi) esiste in JVM, ma non in IJVM. Essa utilizza i due valori in cima allo stack, sostituendoli con un unico valore, il risultato. La seconda parola a partire dalla cima dello stack è l'operando da traslare. Il suo contenuto è spostato a destra di un valore compreso tra 0 e 31; questo valore è specificato nei 5 bit meno significativi della parola in cima allo stack (gli altri 27 bit sono invece ignorati). Il bit del segno viene replicato a destra per tanti bit quanti sono quelli da spostare. Il codice operativo di ISHR è I22 (0x7A).
 - Qual è l'operazione aritmetica che equivale a traslare a destra di un valore 2?
 - Si estenda il microcodice per includere questa istruzione in IJVM.
- L'istruzione ISHL (scorrimento a sinistra su interi) esiste in JVM, ma non in IJVM. Essa utilizza i due valori in cima allo stack, sostituendoli con un unico valore, il risultato. La seconda parola a partire dalla cima dello stack è l'operando da traslare. Il suo contenuto è spostato a sinistra di un valore compreso tra 0 e 31; questo valore è specificato nei 5 bit meno significativi della parola in cima allo stack (gli altri 27 bit sono invece ignorati). Sulla destra vengono inseriti tanti valori zero quanti sono i bit da spostare. Il codice operativo di ISHL è I20 (0x78).
 - Qual è l'operazione aritmetica che equivale a traslare a sinistra di un valore 2?
 - Si estenda il microcodice per includere questa istruzione in IJVM.
- L'istruzione JVM INVOKEVIRTUAL deve conoscere il numero dei suoi parametri. Perché?
- Si implementi per Mic-2 l'istruzione JVM DLOAD. Essa ha un indice a 1 byte e inserisce nello stack la variabile locale che si trova in questa posizione. Successivamente inserisce nello stack anche la parola che si trova nell'indirizzo successivo, più in alto.
- Si disegni un automa a stati finiti per gestire il punteggio di una partita di tennis. Le regole del tennis sono le seguenti. Per vincere occorrono almeno quattro punti e occorre avere almeno due punti più dell'avversario. Si inizia nello stato (0, 0), che indica che non è stato ancora attribuito alcun punteggio. Si aggiunga quindi uno stato (1, 0) che indica che A ha segnato un punto; si etichetti con A l'arco che collega (0, 0) con (1, 0). Si aggiunga poi uno stato (0, 2) che indica che B ha segnato un punto e si etichetti con B l'arco proveniente da (0, 0). Si continui ad aggiungere stati e archi finché non siano stati inclusi tutti i possibili stati.
- Si consideri nuovamente il problema precedente. Ci sono degli stati che potrebbero essere condensati in un unico stato senza cambiare il risultato delle partite? Se sì, quali sono?
- Si disegni un automa a stati finiti per la predizione dei salti che sia più tenace di quello della Figura 4.42 nel senso di modificare le predizioni soltanto dopo tre previsioni errate consecutive.
- Il registro di scorrimento della Figura 4.27 ha una capacità massima di 6 byte. Si potrebbe costruire una versione più economica della IFU con un registro di scorrimento a 5 byte? E che cosa si può dire di una versione con registro a 4 byte?