# Introduzione

Al di sopra del livello logico digitale si trova il livello di microarchitettura incaricato di implementare il livello ISA (Instruction Set Architecture, “architettura dell’insieme di 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 sono costituiti da istruzioni semplici (principalmente RISC) che generalmente è possibile eseguire in un unico ciclo di clock. Nel caso di ISA più complessi (CISC) l’esecuzione di una singola istruzione può invece richiedere più cicli.

**Esempio di microarchitettura**

Ogni microarchitettura rappresenta un caso a se stante per questo analizzeremo la Java Virtual Machine, in quanto è un esempio pratico molto semplice che opera solo su numeri interi. La chiameremo per questa sua caratteristica IJVM.

Iniziamo descrivendo la microarchitettura sopra la quale implementeremo IJVM.

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 JVM di Sun; questo interprete è stato infatti scritto in C per essere portabile e non può controllare l’hardware al livello di dettaglio che necessitiamo.

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.

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 di 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.

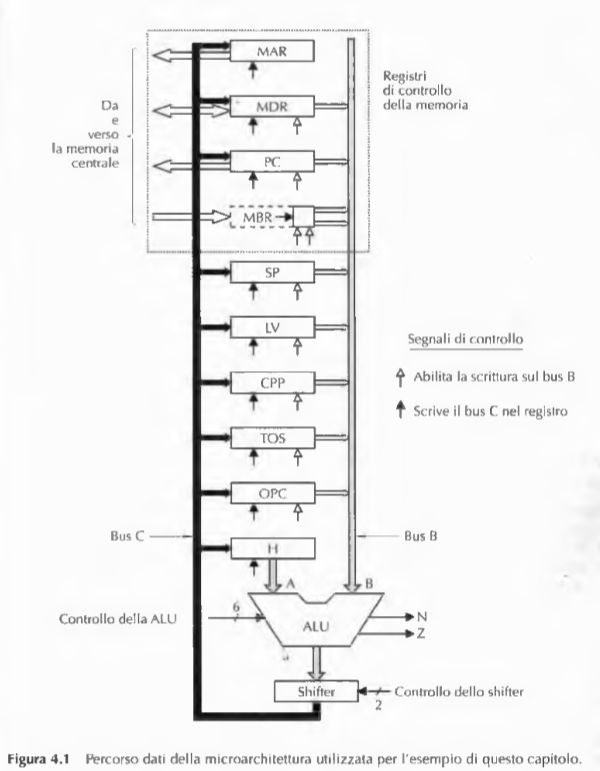
Ogni istruzione IJVM è composta da alcuni campi, di solito uno o due, con uno scopo predefinito. Il primo campo di ogni istruzione è il codice operativo (*opcode*), che identifica il tipo di istruzione, indicando se è di tipo ADD, di tipo BRANCH o altro. Può essere specificato l’operando: per esempio le istruzioni che accedono a una variabile locale devono indicare a quale variabile si riferiscono. Questo modello di esecuzione, chiamato talvolta fetch-decode-execute, è utile a livello astratto e può anche costituire la base per l’implementazione di ISA complessi come IJVM.

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.

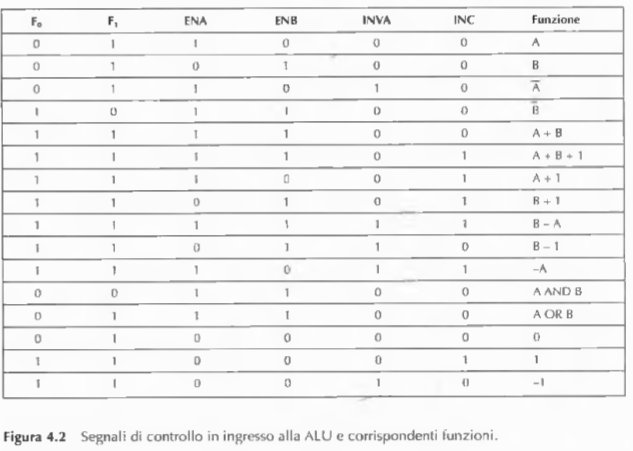
**Percorso dati**

Il percorso dati è quella parte della CPU che contiene la ALU, i suoi input e i suoi output (Figura 4.1).

Contiene dei registri a 32 bit, a cui abbiamo assegnato nomi come PC, SP e MDR. 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.



La funzione dell’ALU 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 F0 e F1 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. 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.

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.

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 essere 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.

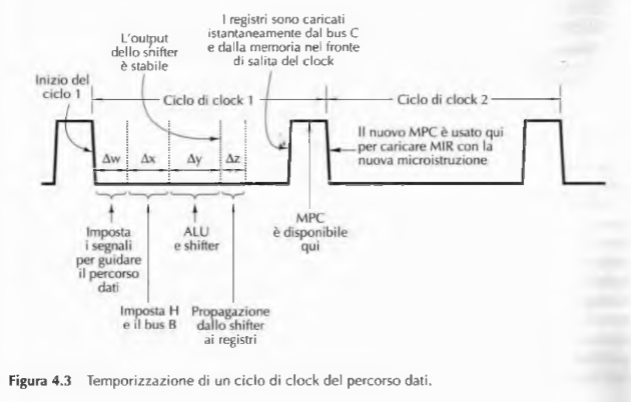
Oltre alle funzioni appena citate, per gestire l’output della ALU è possibile utilizzare altre due linee di controllo. SLL8 (*Shift Left Logical*, “scorrimento logico a sinistra”) trasla il valore a sinistra di un byte, impostando gli 8 bit meno significativi a 0. SRA1 (*Shift Right Arithmetic*, “scorrimento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

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 che può essere determinato dal clock principale. In corrispondenza del fronte di discesa dell’impulso vengono impostati i bit che piloteranno tutte le porte logiche. Questa operazione richiede un intervallo di tempo finito e conosciuto a priori: Δ*w*. 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 Δ*x*.



A questo punto la ALU e lo shifter cominciano a operare sui dati validi e i loro output diventano stabili dopo un altro intervallo temporale, Δ*y*. Passato un ulteriore tempo Δ*z* 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.

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. 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 (Δ*w*).

2. I registri vengono caricati nel bus B (Δ*x*).

3. La ALU e lo shifter svolgono le loro operazioni (Δ*y*).

4. I risultati vengono propagati lungo il bus C e ritornano nei registri (Δ*z*).

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 Δ*w* + Δ*x* dopo il fronte di discesa del clock. Analogamente anche i loro output sono inconsistenti finché non sia trascorso un tempo Δ*w* + Δ*x* + Δ*y* dopo il fronte di discesa del clock. Gli unici segnali espliciti 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 del progettista assicurarsi che il tempo Δ*w* + Δ*x* + Δ*y +* Δ*z* giunga sufficientemente in anticipo rispetto ai 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. Per iniziare una lettura o una scrittura occorre caricare il registro di memoria appropriato e successivamente inviare alla memoria un segnale di scrittura.

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 232, nessuno dei quali è significativo per la nostra macchina che ha un limite di 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.

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.

**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 indicano le operazioni da eseguire durante un ciclo, il quale consiste nel portare i valori dei registri sul bus B, propagarli attraverso la ALU e lo shifter, guidarli sul bus C e riscrivere i risultati negli appositi registri. 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.

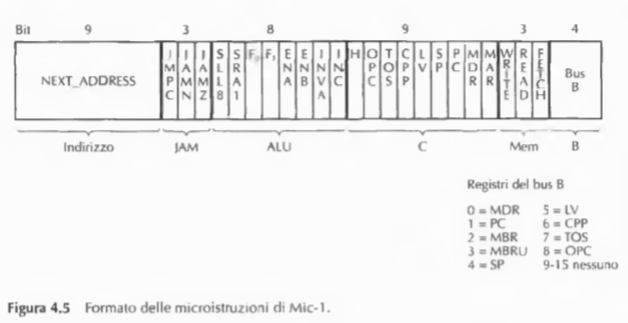
Questo comportamento apparentemente contraddittorio è spiegato nella Figura 4.3. Durante il ciclo 1 i segnali di controllo della memoria vengono generati soltanto verso la fine del ciclo, subito dopo il momento in cui MAR e PC sono caricati in corrispondenza del fronte di salita del clock. Assumeremo 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. Ovviamente 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 restituiscono 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. E’ 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.

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. La Figura 4.5 mostra un possibile formato, diviso in sei gruppi 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.

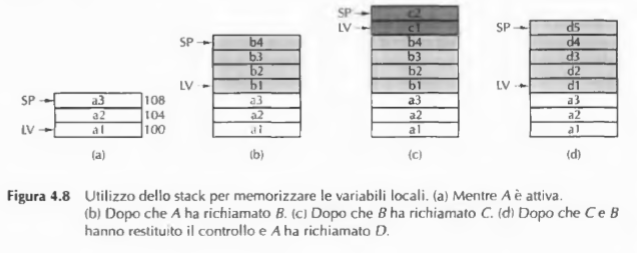
**Esempio di ISA: IJVM**

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 di istruzioni (ISA) con il termine di macroarchitettura, in contrapposizione con la microarchitettura. Prima di descrivere IJVM faremo una breve digressione che ci permetterà di metterne in luce le motivazioni.

**Stack**

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. 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 procedura corrente. Nella Figura 4.8(a) è stata richiamata una procedura A, che possiede tre variabili locali, a1, a1 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.



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).

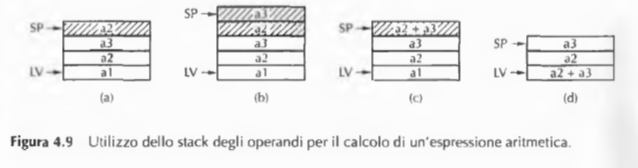
Quando C termina, B diventa nuovamente attiva e lo stack viene modificato per tornare allo stato mostrato nella Figura 4.8(b), in modo che LV torni a puntare alle variabili locali di B. Analogamente, quando B restituisce il controllo, si torna nuovamente alla situazione mostrata nella Figura 4.8(a).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



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 *x2 + f(x)*, una parte dell’espressione (per esempio, *x2*) 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 *x2*, in modo che l’istruzione successiva li possa sommare.

**Modello della memoria di IJVM**

Fondamentalmente essa consiste in una memoria concepibile in due modi distinti: come un array di 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.

1. 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 implicito, CPP, contenente gli indirizzi della prima parola della porzione costante di memoria.

2. 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 con cui è stato invocato il metodo.

Il blocco delle variabili locali non comprende lo stack degli operandi, che è separato.

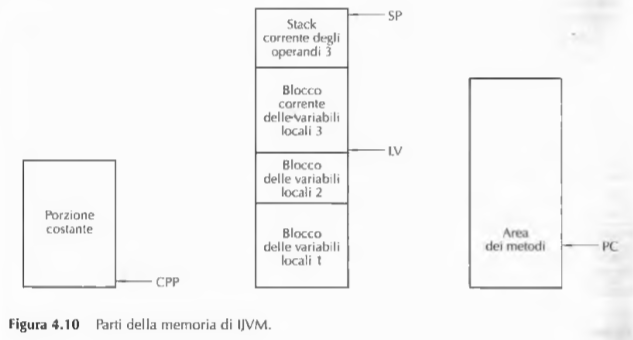
È presente un registro implicito, LV, contenente l’indirizzo della prima locazione all’interno del blocco delle variabili locali.

3. 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).

C’è un registro implicito contenente l’indirizzo della parola in cima allo stack.

4. 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 implicito che contiene l’indirizzo della successiva istruzione da prelevare. Questo puntatore è chiamato Program Counter.



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 di 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: 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.

**Insieme di istruzioni IJVM**

La Figura 4.11 mostra l’insieme di 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.

Alcune istruzioni permettono di inserire nello stack una parola proveniente da varie fonti, come ad 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. Sono fornite quattro istruzioni per i salti, una non condizionale (GOTO) e tre condizionali (IFEQ, IFLT e IF\_ICMPEQ).

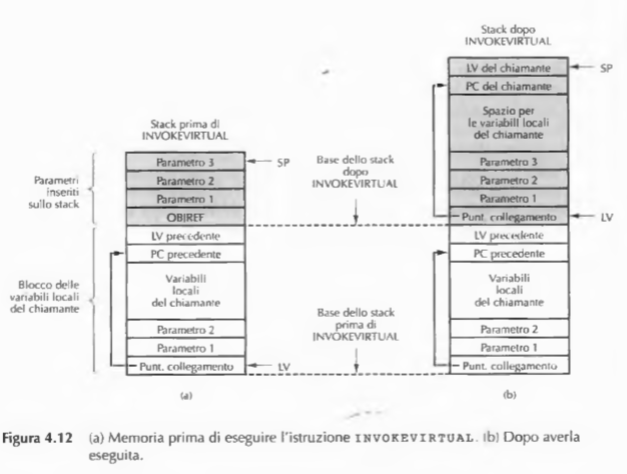


Tutte queste istruzioni, se accettate, modificano il valore di PC in base alla grandezza del loro spiazzamento (16 bit con segno), che si trova nell’istruzione, subito dopo il codice operativo. Questo spiazzamento viene aggiunto all’indirizzo dell’istruzione. Ci sono anche istruzioni IJVM che permettono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

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. Per semplicità 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. 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. 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.

Questa 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, tra cui il numero di parametri del metodo. In questo conteggio OBJREF viene considerato come un parametro: il parametro, che insieme al valore di SP, fornisce la locazione di OBJREF.

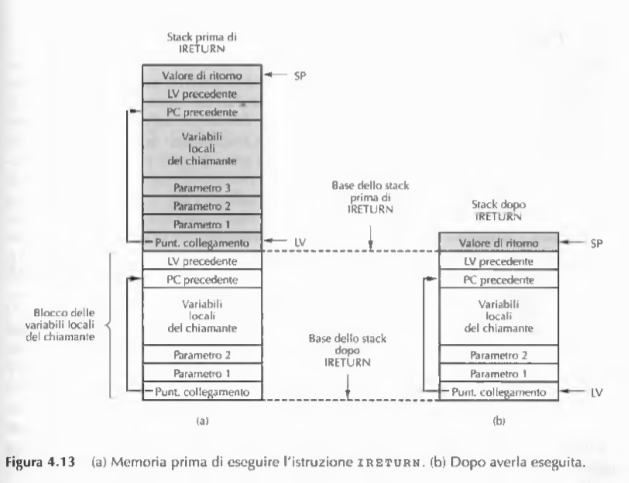


Un altro dato speciale è la dimensione del blocco delle variabili locali per il metodo invocato.

Questo valore è necessario, dato che verrà stabilito un nuovo stack per il metodo, immediatamente sopra il blocco delle variabili locali. Infine viene indicato 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. 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.

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.