Un esempio di microarchitettura (approfondimenti)

ST

Definizioni e componenti

La microarchitettura è uno strato dell'elaboratore (si appoggia sul livello logico-digitale), la cui funzione è quella di interpretare le istruzioni definite all'interno o al livello dell'insieme delle istruzioni che quel processore è in grado di eseguire. Tale insieme di istruzioni viene identificato dalla sigla ISA che il processore è in grado di eseguire.
Non esistono delle microarchitetture 'standard' ossia non vengono utilizzate o che siano di validità generali per tutti i tipi di processori. Invece esistono famiglie di micro architetture, alle quali corrispondono i relativi Instruction Set che ad esempio un processore Intel o Motorola è in grado di riconoscere ed eseguire. Per la micro architettura ISA utilizzeremo un processore sul quale funziona Integer Java Virtual Machine. IJVM identifica una macchina virtuale Java in grado di eseguire istruzioni su operandi interi, pertanto utilizzeremo questo sottoinsieme dell'ISA.

Osservazione
La macchina IJVM è un processore didattico, ovvero sia non è presente dal punto di vista industriale sul mercato.

Il livello di microarchitettura prevede quattro fasi:

- ✓ Un microprogramma presente nella memoria a sola lettura (ROM). Si tratta di un insieme di microistruzioni che il processore è in grado di eseguire. Questo microprogramma in ROM è presente sullo stesso chip del microprocessore.
- ✓ Lettura delle singole istruzioni che compongono il programma che dee essere eseguito. Questa operazione di lettura delle istruzioni fa riferimento alle istruzioni memorizzate nella RAM. Le istruzioni lette dalla RAM costituiscono il vero è proprio programma che il microprogramma deve eseguire. La fase delle singole istruzioni della memoria centrale è detta anche fase di fetch.
- ✓ In seguito, dopo che il processore abbia eseguito il fetch, cioè ha trasferito dalla memoria RAM l'istruzione che deve essere eseguita, vi è una fase di decodifica, la quale si fa carico di riconoscere l'istruzione che è stata trasferita dalla RAM, e che specificata tramite il suo codice operativo (op-code). Nella fase di decodifica il processore 'osserva' l'istruzione che ha trasferito la RAM e cerca di identificarla.
- ✓ L'ultima fase prevede l'esecuzione della singola istruzione, cioè dopo averla riconosciuta il processore la esegue.

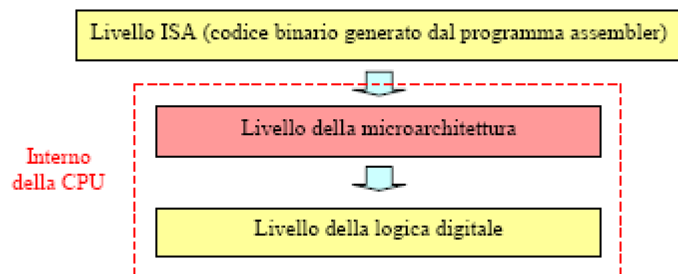
Componenti della microarchitettura

Per eseguire ogni singola istruzione il processore gestisce un insieme di variabili le quali possano essere modificate mediante opportune funzioni. L'insieme delle variabili definisce lo stato del processore. Ogni variabile gestita dal processore è memorizzata in un registro, il quale prevede l'utilizzo di un insieme di bistabili che sono integrati sullo stesso chip del processore. Inoltre su esso vi è un opportuno registro (o variabile) che conserva traccia dell'istruzione che viene eseguita. L'obiettivo di tale registro, detto anche Program Counter è quello di identificare la cella di memoria e quindi l'indirizzo della parola di memoria dove è memorizzata la prossima istruzione che deve essere eseguita.
Durante la prima fase dell'esecuzione dell'istruzione, ovvero durante la fase di fetch, il processore trasferisce dalla RAM all'interno del processore stesso l'istruzione che deve essere eseguita. Per disporre dell'indirizzo di memoria dove è memorizzata la prossima istruzione da eseguire, viene proprio utilizzato il PC.

Formato delle istruzioni

Le istruzioni sono memorizzati all'interno della RAM tramite opportuni formati. In particolare ogni istruzione prevede che sia indicato un codice operativo, il quale specifica il tipo dell'operazione e quindi indica se si tratta di un'operazione di somma (ADD), di sottrazione (SUB) oppure di salto (Branch). Possono essere specificati uno (o eventualmente) più operandi per riconoscere ad esempio quali sono gli operandi che devono essere utilizzati per eseguire l'operazione di somma, nonché dove viene memorizzato il risultato.
L'esecuzione di ogni singola istruzione richiede uno o più cicli di clock, in base al tipo di istruzione che deve essere eseguita. Per accedere ad operandi o registri, ogni istruzione richiede anche l'accesso eventuale ad uno o più operandi e pertanto ad uno o più registri, nonché la modifica del valore memorizzato in ognuno di questi registri.

In altri termini: Il livello della ***microarchitettura*** *descrive il funzionamento interno di una* ***CPU****, e in particolare come le istruzioni* ***ISA*** *(****I****nstruction* ***S****et* ***A****rchitecture) vengono interpretate ed eseguite dall'hardware (livello della logica digitale) che costituisce la CPU.*
Diverse CPU moderne, in particolare quelle ***RISC*** *(****R****educed* ***I****nstruction* ***S****et* ***C****omputer), hanno istruzioni semplici che possono essere eseguite in un singolo ciclo di clock. D'altro canto le CPU* ***CISC*** *(****C****omplex* ***I****nstruction* ***S****et* ***C****omputer) forniscono istruzioni anche molto complesse che sono interpretate da microprogrammi che richiedono diversi cicli di clock. In altre parole:*



- *le CPU **RISC** (es. SGI MIPS, Digital Alpha, Sun UltraSparc, IBM PowerPC) semplificano il disegno della CPU che per questo motivo è spesso molto "snella" e veloce, delegando ai compilatori il compito di tradurre con pochi mattoncini di base programmi anche molto complessi.*
- *le CPU **CISC** (Motorola 68xxx, Intel x86, Intel Pentium) forniscono ai compilatori molte più possibilità di traduzione e molte semplificazioni (ad. esempio gestione context-switch); questo però va spesso a discapito dell'efficienza e della semplicità di progetto.*

Progetto di una microarchitettura

Risulta in generale estremamente difficile dare criteri generali per il progetto di microarchitetture. Per una comprensione dei problemi coinvolti è dunque preferibile analizzare un esempio semplice ma abbastanza generale da poter essere espanso e "complicato" a piacere...

Una CPU che implementa istruzioni IJVM

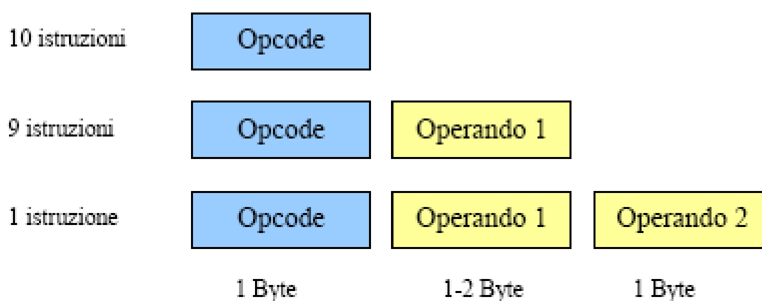
***Java** è un linguaggio di alto livello per calcolatori (simile al C) introdotto da **Sun** nei primi anni '90 e divenuto piuttosto popolare a partire dal '95 quando i browser Netscape e Internet Explorer decisero di adottarlo. La principale caratteristica del linguaggio Java è quella di essere indipendente dalla piattaforma hardware.*
Come far eseguire a macchine con diversi ISA, programmi scritti nello stesso linguaggio ?
*• Semplice, basta fornire un **interprete**, per ciascuna architettura, che traduca il codice Java in istruzioni ISA di quell'architettura. Questo approccio ha però il grosso svantaggio di non permettere la compilazione dei programmi e quindi di introdurre forte rallentamento a causa dell'interpretazione "run-time" del codice Java di alto livello.*
*• La soluzione scelta da Sun, più complessa ma molto più efficiente, consiste nel compilare il programma Java in un codice binario (di basso livello) detto **Bytecode** che viene eseguito da una **JVM** (**J**ava **V**irtual **M**achine). La compilazione risulta quindi indipendente dalla piattaforma hardware scelta per ognuna delle quali dovrà però essere fornita una JVM.*

Come realizzare una JVM ?
*Una JVM non è altro che un **interprete** (di basso livello) da istruzioni in formato Bytecode-Java a istruzioni ISA.*
Esistono in realtà alcuni processori (es. PicoJava II) il cui ISA coincide con le istruzioni JVM; in questo caso non c'è necessità di interprete!

IJVM come ISA

*Nel seguito ci concentreremo dunque sullo studio della microarchitettura di una CPU in grado di eseguire un sottoinsieme delle istruzioni Bytecode-Java: questo sottoinsieme contiene solo istruzioni su numeri interi (e non floating point) e per questo motivo è denominato **IJVM***

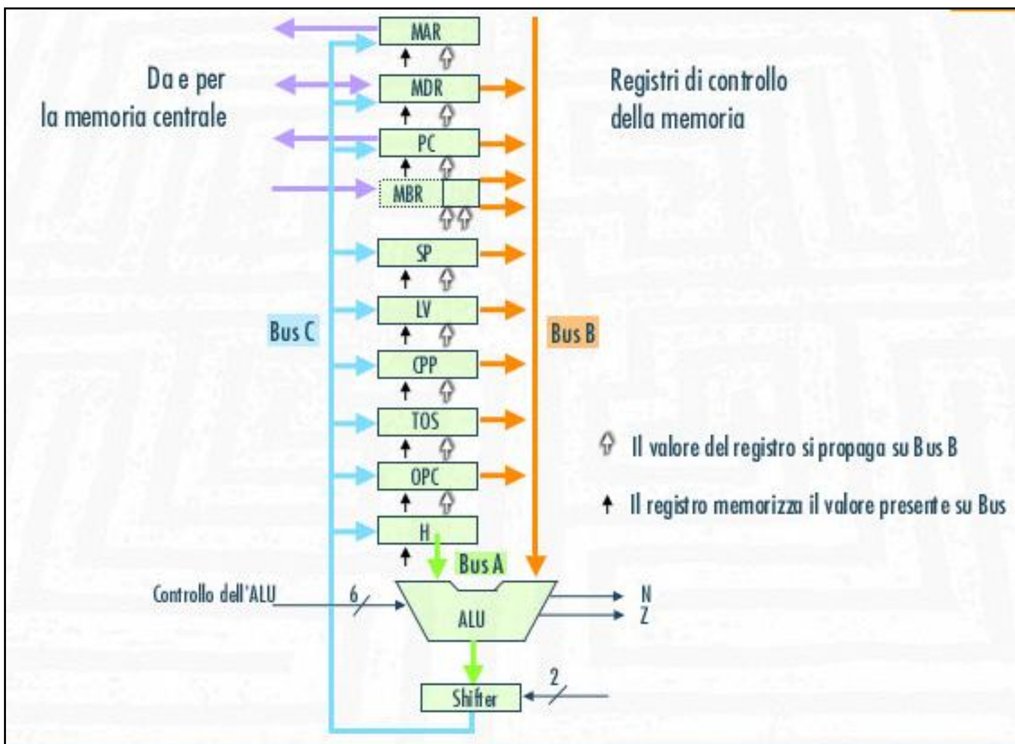
*(**I**nteger **JVM**)*
*Le istruzioni ISA della nostra CPU saranno dunque i Bytecode IJVM, chiamati nel seguito semplicemente **istruzioni IJVM**.*
*Le (**20**) istruzioni IJVM sono brevi; ogni istruzione è dotata di alcuni campi, solitamente 1 o 2, ognuno dei quali ha uno scopo preciso:*
*• Il primo campo dell'istruzione è l'**op-code** (abbreviazione di **op**eration **code**) che identifica l'istruzione specificando se si tratta di ADD, BRANCH o altro.*
*• Molte istruzioni dispongono di un campo supplementare che specifica un **operando**. Ad esempio un'istruzione che deve accedere a una variabile locale deve specificare tramite un operando dove questa variabile si trova (indirizzo di memoria).*



*Come vedremo, le istruzioni IJVM fanno largo uso dello stack. Molte istruzioni vengono infatti eseguite prelevando operandi dalla cima dello stack e salvando sullo stack il risultato. Ciò consente di limitare (solitamente a max. 1) il numero di operandi delle istruzioni (non è necessario specificare indirizzi di memoria) e quindi di produrre **Bytecode** molto **compatti**.*

Data path

Il data path è il cuore della CPU, ovvero quella parte che contiene l'**ALU** con i suoi input e output, e i **registri** interni.
I primi componenti che troviamo nel data path sono i registri di controllo della memoria. Il registro MAR (Memory Address Register) viene utilizzato per specificare qual è l'indirizzo della cella di memoria a cui si vuole fare riferimento. È un registro di 32 bit, ciò consente di indirizzare una cella di memoria all'interno di uno spazio di indirizzamento, cioè di una memoria composto da 4 Giga parole.
Il registro MDR (Memory Data Register) è il registro che contiene il dato ovvero la parola di memoria. L'MDR conterrà la parola di memoria specifica dal registro MAR, cioè l'indirizzo di memoria specificato da MAR nel caso di operazioni di lettura.
Nel caso di operazioni di scrittura l'MDR conterrà il valore che deve essere memorizzato nella parola di memoria il cui indirizzo è specificato dal registro MAR il quale è un registro a 32 bit.
Il registro PC (Program Counter) o contatore di programma, il quale indica qual è la prossima che deve essere eseguita, cioè indica qual è la parola di memoria dove è memorizzata la prossima istruzione.
Il registro MBR (Memory Byte Register) è un registro a 8 bit che viene utilizzato per trasferire dalla memoria gli 8 bit relativi alla parola specificata nel registro PC. All'interno del data path saranno presenti delle connessioni da e per la memoria centrale utilizzando questi registri di controllo di memoria.
Vi sono altri registri come SP (Stack Pointer), utilizzato ad es. come registro puntatore alla pila.
Il registro H (holding – mantenimento) viene utilizzato dall'ALU che esegue i calcoli. Ognuno dei registri (compresi LV, CPP, TOS, OPC, oltre al citato SP, costituiscono l'ISA) sono dotati di opportuni segnali di controllo che vengono utilizzati o per memorizzare all'interno del registro il valore che viene presentato ai suoi ingressi oppure per propagare alle uscite il valore memorizzato all'interno del registro.
Il simbolo 🡅 fa riferimento ad un segnale che consente al registro di memorizzare il valore che si presenta ai suoi ingressi.
Il simbolo ⇧ fa riferimento ad un segnale che abilita il registro a propagare sulla sua uscita il valore memorizzato all'interno dello stesso registro.
All'interno del data path, abbiamo tra l'altro i segnali di controllo che abilitano la memorizzazione del valore nel registro che si presenta ai suoi ingressi, e la propagazione alle uscite del valore memorizzato da ogni singolo registro.



È importante sottolineare che, mentre più registri possono memorizzare contemporaneamente al loro interno lo stesso valore che si presenta sul Bus C, quindi ad es. MDR e MAR possono assumere lo stesso valore (leggono lo stesso valore presente sul Bus C simultaneamente), oppure il registro LV è il MAR possono contemporaneamente memorizzare lo stesso valore che è presente sul Bus C. Mentre non è assolutamente vero, per quanto riguarda la propagazione del valore memorizzato da un registro. Infatti è necessario che un solo registro per volta possa propagarsi col il proprio valore sulla linea Bus B.
Se più registri volessero simultaneamente propagare il proprio valore memorizzato sulla linea del Bus B si avrebbe un corto circuito. La propagazione di valori non uguali, genera conflitti tra essi sul Bus B.
Se per es. MBR vuole trasmettere il valore 0 sul Bus B, e un altro registro ad es. SP vuole trasmettere un valore differente, genera un corto circuito per ogni bit differente tra il valore memorizzato di SP e il valore memorizzato tra MBR. I registri sono tutti a 32 bit.

Sincronizzazione del data path

La figura in basso mostra la temporizzazione degli eventi sul precorso dati. all'inizio di ogni ciclo di clock viene generato un breve impulso. 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. Questa operazione richiede un intervallo di tempo finito e conosciuto a priori: **Δw**.
Il registro richiesto viene 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 cominciano a operare sui dati validi e i loro output diventano stabili dopo un altro intervallo di temporale **Δy**.
Trascorso 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 discesa dell'impulso successivo.

Il caricamento

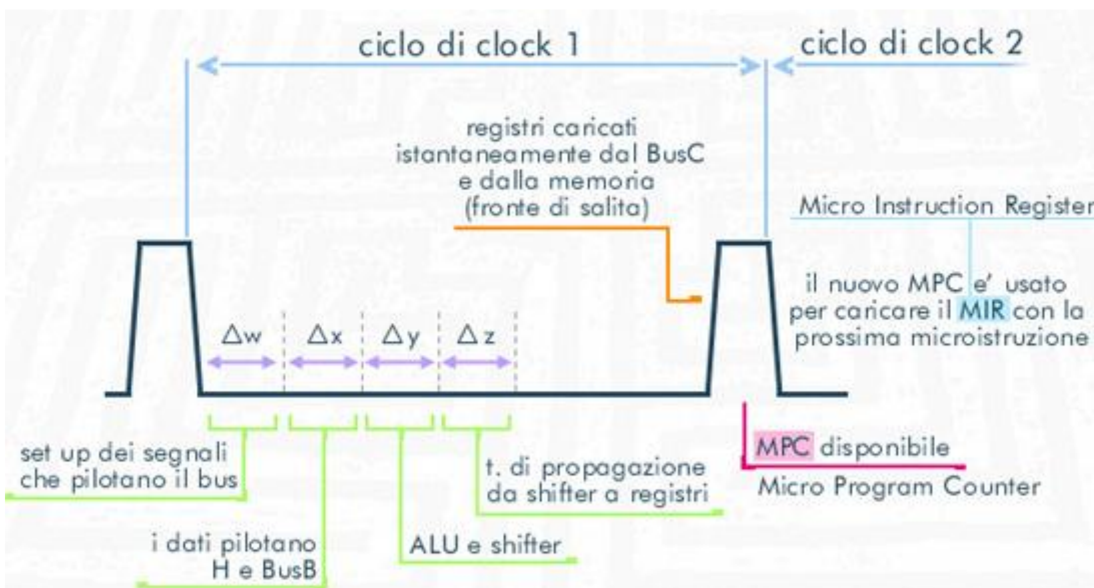
Caratteristiche principali all'interno dei registri dovrebbe essere pilotato dal fronte del segnale ed essere veloce; in questo modo, anche se alcuni dei registri di input vengono modificati, gli effetti di queste modifiche giungeranno sul Bus C, solo dopo un tempo sufficientemente lungo rispetto al 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. In 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.
Affinché questa architettura funzioni è necessaria *una rigida sincronizzazione*, *un lungo ciclo di clock*, *un ritardo di propagazione attraverso la ALU* conosciuto a priori e *un veloce caricamento dei registri sul Bus C*.
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 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 (Δx)**

In corrispondenza del fronte di salita del ciclo successivo i risultati vengono memorizzati nei registri

Osservazione

La ALU e lo shifter funzionano in continuazione; tuttavia i loro input vanno considerati come inconsistenti fino al tempo Δw + Δx dopo il fronte di salita del clock. Analogamente anche i loro output sono inconsistenti finché non sia trascorso un tempo Δw + Δx + Δy 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. Per velocizzare il data path, occorre scegliere la frequenza di clock più alta possibile, per avere più cicli di clock nella stessa unità di tempo, e quindi disporre di un processore più veloce.
La scelta deve essere tale da assicurare che l'intervallo Δw + Δx + Δy + Δz accada con sufficiente anticipo rispetto al fronte di salita che è presente sul ciclo di clock successivo *(deve essere minore del tempo in cui il clock è 0).*
Pertanto tale vicolo tende a limitare il valore massimo della frequenza di clock che possa essere adottato per il funzionamento del data path.



In breve

- Tutti i cicli di clock hanno la stessa durata
- I segnali di controllo si stabilizzano durante **Δw**
- I registri propagano i propri valori sul Bus B durante **Δx**
- ALU e shifter operano durante **Δy**
- I risultati si propagano sul Bus C durante **Δz**
- I registri memorizzano i valori osservati sul Bus C durante il fronte di salita sul ciclo successivo

Quindi si verifica il fronte di salita che è presente fra il ciclo di clock 1 e il ciclo di clock 2, in quel preciso istante memorizzano al loro interno ciò hanno sugli ingressi (del Bus C).

Sequenze

- Ogni **nuovo ciclo** ha inizio sul fronte di discesa del clock (che come si può notare rimane basso per circa i ¾ del periodo).
- Durante il fronte di discesa vengono memorizzati i bit che controllano il funzionamento delle porte (segnali di controllo). E' necessario attendere un primo intervallo di tempo (**Δw**) affinché i segnali si propaghino e raggiungano uno stato di stabilità.
  I valori (gli operandi) sono letti sul fronte di discesa del segnale di clock.
- Il registro selezionato per essere inviato sul bus B insieme ad H viene reso disponibile alla ALU. Dopo un intervallo di tempo (**Δx**) la ALU può iniziare ad operare sui dati.
- Un ulteriore intervallo di tempo (**Δy**) è necessario affinché le porte logiche di ALU e shift register producano un output stabile.
- Infine, dopo l'intervallo di tempo (**Δz**) l'output dello shift register è stato propagato lungo il bus C e in corrispondenza del fronte di salita del successivo impulso di clock i registri selezionati vengono caricati dal bus C. *I valori (quindi i risultati) sono scritti sul fronte di salita del segnale di clock.*

Data path e modalità di accesso alla memoria (breve 1)

Richiami

La RAM può essere vista come una tabella, in cui sono presenti le righe orizzontali che sono dette parole e dove tutte hanno la medesima lunghezza detta lunghezza di parola (o dimensione orizzontale della memoria).
Ogni parola è individuata da un indirizzo. Il numero di parole fornisce la dimensione verticale della memoria. La prima parola incontrata ha un indirizzo 0, pertanto con 1 bit posso indirizzare 2^n parole (0 … 2^n - 1).
La nostra CPU ha due modi per comunicare con la memoria:
Una **porta da 32 bit** per la lettura/scrittura dei **dati** del livello ISA. La porta viene controllata da due registri:
MAR (**M**emory **A**ddress **R**egister) specifica l'indirizzo di memoria in cui si desidera leggere o scrivere una parola.
MDR (**M**emory **D**ata **R**egister) ospita la parola (32 bit) che sarà letta o scritta all'indirizzo di memoria specificato da MAR.
Una **porta da 8 bit** per leggere (solo lettura!) il programma eseguibile (**fetch** delle istruzioni ISA). Anche questa porta è controllata da 2 registri:
PC (**P**rogram **C**ounter) è un registro a 32 bit che indica l'indirizzo di memoria della prossima istruzione ISA da caricare (fetch).
MBR (**M**emory **B**yte **R**egister) contiene il byte letto dalla memoria durante il fetch. MBR è in realtà un registro a 32 bit, pertanto il byte letto viene memorizzato negli 8 bit meno significativi.
Per indirizzare un'intera parola di memoria viene utilizzato il registro MAR (32 bit) tramite esso viene letta una parola dalla memoria e vi accede tramite l'MDR (32 bit). Con una modalità di accesso di tipo ***Word-addressable***, specifico una parola di memoria tramite un registro a 32 bit (MAR fornisce l'indirizzo, MDR contiene il risultato dell'operazione di accesso alla memoria).
I registri MAR e MDR funzionano come acceso alla memoria in modalità scrittura, allora utilizzerò il registro MDR come il registro sorgente e scriverò il contenuto di MDR nella parola di memoria dell'indirizzo di memoria fornita da MAR, sia in modalità lettura. Pertanto trasferirò la parola di memoria di indirizzo MAR nel registro MDR.
La modalità di scambio dati tra memoria e CPU del tipo ***Byte-addressable*** si attua utilizzando il registro PC a 32 bit, il quale indirizza 1 Byte di memoria e vi accede attraverso un registro a 8 bit detto MBR. Quindi il PC specifica qual è il Byte di memoria che deve essere utilizzato, il contenuto di questo Byte di memoria trasferito nel registro di 8 bit MBR.

Esempio

Scriviamo nel registro MAR il valore 2. In questo caso accedo alla memoria con modalità word-addressable. L'operazione di lettura porterà a far si che nel registro MDR la parola 2 della memoria centrale. Poiché la RAM è mappata, e in un certo modo, vuol dire che con questa operazione trasferiremo la parola 2 della memoria e quindi i Byte della RAM il cui indirizzo è compreso tra 8 e 11. Pertanto la parola 2 è formata dai Byte 8, 9, 10 e 11 della stessa RAM.

Esempio

Modalità di accesso alla memoria con modalità Byte-addressable. Scriviamo nel registro PC il valore 2. Effettuiamo l'operazione di lettura. Il risultato dell'operazione porterà gli 8 bit del registro MBR ad assumere il valore del Byte 2 della RAM. In questo caso, leggeremo Byte 2 della RAM.

Riepilogo
Per l'accesso alla memoria vengono utilizzati:

- MAR/MDR (per gli indirizzi/per i dati) quando faccio riferimento a dati che sono specificati nel livello ISA (linguaggio Assembler).
- PC/MBR (per gli indirizzi/per i dati) per accedere alla memoria, faccio riferimento alla lettura del programma eseguibile del livello ISA.

Tutti i rimanenti registri sono di 32 bit e quindi word-addressable. Per essi avremo un trasferimento di 32 bit. Per passare da un conteggio in parole ad un conteggio in Byte bisogna moltiplicare per 4. Tale numero è dovuto al fatto che la parola è composta da 32 bit, mentre il Byte è formato da 8 bit.
La moltiplicazione di un numero binario per 4 si ha aggiungendo due zeri a destra dell'ultimo bit. Se aggiungessi un solo zero a destra dell'ultimo bit moltiplicherei per 2 il numero dato.
Analogamente, per moltiplicare per 10 si aggiunge uno 0 a destra dell'ultima cifra, etc. in generale la moltiplicazione di un numero per un multiplo della sua base viene effettuata aggiungendo uno 0 a destra dell'ultima cifra.
In questo caso disponendo del registro MAR formato da 32 bit, il quale conta in parole, se voglio tradurlo in un indirizzo a 32 bit sul Bus dove il conto viene effettuato in Byte, basta spostare a sinistra di 2 bit tutti gli zeri dentro MAR; portando a zero i 2 bit meno significativi, quindi 2 bit sulla destra e scartando di conseguenza i 2 bit più significativi dei 2 bit più a sinistra, sempre nel registro MAR. in questo modo posso tradurre il contenuto del registro MAR espresso in parole in un indirizzo presente sul Bus a 32 bit espresso in Byte.
Abbiamo visto che i vari registri possono assegnare e propagare il proprio valore memorizzato attraverso il Bus B ad altri registri. In modo analogo il registro MBR può propagare sul Bus B il proprio valore. Esiste una considerazione di base da tenere presente: l'MBR è formato da 8 bit. Gli altri registri da 32 bit propagano il loro valore utilizzando le 32 linee di Bus B. per l'MBR occorre distinguere se la rappresentazione al suo interno è in formato:
a) **Unsigned**
b) **Signed** (con segno)
Nel primo caso, gli 8 bit dell'MBR rappresentano un numero che è compreso tra 0 e 255 ($2^8 - 1$), essendo di 8 bit il numero massimo che può essere rappresentato, quindi 0 e $2^8 - 1$. La propagazione sul Bus B di un numero compreso tra 0 e 255 (numero senza segno) comporta l'operazione seguente: i 24 bit superiori o più significativi sono forzati a 0, gli altri bit inferiori (o meno significativi) contengono tali e quali il valore di MBR
Nel caso di rappresentazioni di un numero all'interno di MBR in formato Signed (dotato di segno), i numeri che possono essere rappresentati sono compresi tra -128 … +127 ($2^n - 1$ e $2^{n-1} - 1$). Il problema che si pone è in che modo rappresentare il valore che si presenta sul Bus B di MBR in formato Signed.
Sul Bus, i 24 bit più significativi devono duplicare il segno di MBR, mentre gli 8 bit meno significativi contengono il valore di MBR. In questo è necessario effettuare un'estensione del segno portando il registro MBR anziché ad essere rappresentato su 8 bit, lo si rappresenta su 32 bit. Il meccanismo di estensione del segno prevede di prendere il bit più significativo e quindi rappresenta il segno ed applicarlo su tutti i 9 bit che vengono ad essere aggiunti e quindi sui 24 bit più significativi del Bus.
Ritornando al data path, il registro MBR dispone di due segnali di controllo in scrittura, i quali abilitano rispettivamente la scrittura del valore di MBR sul Bus B, rappresentando MBR o in formato Unsigned o formato Signed.

Data Path e accesso alla memoria (breve 2)
Le due porte **MAR/MDR** e **PC/MBR** indirizzano la memoria a parole (32 bit) e a byte (8 bit) rispettivamente.
• Questo significa che gli indirizzi memorizzati in MAR sono espressi in termini di parole: *ad esempio se MAR contiene il valore 2, una lettura*
dalla memoria causa il caricamento in MDR dei byte di memoria 8, 9, 10 e 11.
• Indirizzando invece PC la memoria in termini di byte, *la lettura dall'indirizzo 2, causa il trasferimento negli 8 bit meno significativi di*
MBR del byte di memoria 2.
Questa apparente complicazione, semplifica in realtà il funzionamento interno in quanto il fetch delle istruzioni può avvenire un byte per volta,
mentre risulta possibile leggere o scrivere in memoria una parola (32 bit) in un unico ciclo.
Fisicamente la memoria è realizzata come un unico spazio lineare organizzato in byte. Adottando il semplice accorgimento di figura è possibile indirizzare con MAR la memoria in termini di parole senza bisogno di introdurre nessun particolare circuito. Il trucco consiste nel collegare MAR sul bus indirizzi sfalsato di due posizioni, ovvero di non utilizzare i due bit più significativi di MAR e di collegare il bit 0 di MAR con il bit 2 degli indirizzi, il bit 1 di MAR con il bit 3 degli indirizzi e così via. I due bit meno significativi degli indirizzi vengono semplicemente impostati a 0.

Segnali di controllo del data path

Abbiamo fino ad ora analizzato il comportamento del data path, delle sue sincronizzazioni e dei suoi segnali di controllo. Riepilogando, il nostro data path possiede **29 segnali di controllo** suddivisibili in cinque gruppi funzionali:

9 segnali di controllo (frecce piene) occorrono per controllare la memorizzazione dei valori di Bus C nei registri (essendo 9 i registri);

9 segnali di controllo (frecce vuote) per abilitare i registri sul bus B (propagazione e abilitazione di un solo valore dei registri per volta)

8 segnali di controllo per il funzionamento dell'ALU (6) e per lo shifter register (2)

2 segnali di controllo (che non appaiono in figura) per indicare i segnali read/write della RAM attraverso MAR/MDR

1 segnale di controllo (che non appaiono in figura) per indicare la fase di fetch attraverso il PC/MBR

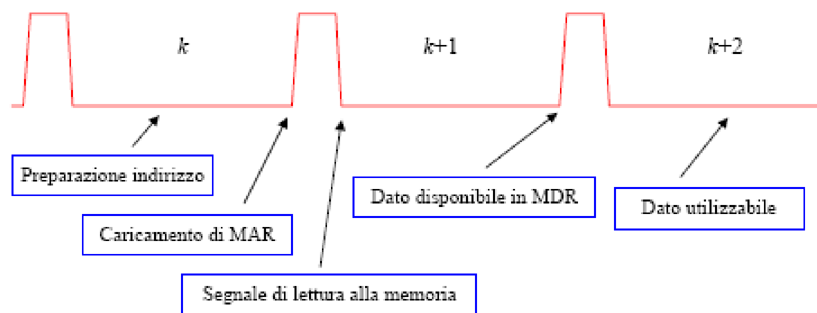
Come già detto in un singolo ciclo di data path è possibile leggere valori da registri, passarli alla ALU e memorizzare su registri il valore calcolato.

I **29 segnali** suddetti specificano il comportamento della CPU per un singolo ciclo di data path. Nella nostra CPU (come nella maggior parte delle CPU) il periodo di data path coincide con il periodo di clock.

Come vedremo, un ciclo di data path non corrisponde però a un'istruzione ISA, per l'implementazione della quale generalmente occorrono diversi cicli di data path (specie quando risulta necessario reperire valori dalla memoria). La sequenza di cicli di data path necessari all'esecuzione di un'istruzione ISA assume il nome di **microprogramma** di quell'istruzione ISA. Un microprogramma è costituito da **microistruzioni**.

Data Path e accesso alla memoria (3)

Un ciclo consiste nel portare i valori dei registri sul Bus B, propagare i segnali attraverso la ALU e lo shifter, guidarli sul Bus B e infine scrivere i risultati nel registro o nei registri appropriati. Inoltre, nel caso in cui asserito il segnale per una lettura della 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 della sincronizzazione di un ciclo di clock. 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, nel successivo fronte di salita del clock. In altri termini, carichiamo MAR alla fine del ciclo del percorso dati e poco dopo facciamo partire l'operazione di memoria. Di conseguenza non possiamo aspettarci 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, purché queste non necessitano di parole della 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, essi possono essere letti nei cicli in cui si sta svolgendo una nuova lettura della memoria. Essi restituiscono valori vecchi in quanto una lettura della memoria non ha 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 abilitare nello stesso momento più di un registro sul Bus B! 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. A questo punto possiamo controllare il percorso dati con 9 + 4 + 8 + 2 + 1 = 24. Tuttavia questi bit controllano il percorso dati soltanto per un ciclo.

La seconda parte del controllo consiste invece nel determinare che cosa deve fare essere svolto durante il ciclo successivo. Per includere questo aspetto nel progetto del controllore definiremo un formato un formato che ci permetterà di descrivere le operazioni da eseguire da utilizzando i 24 bit di controllo più due campi aggiuntivi: NEXT_ADDRESS e JAM.

In breve
Sebbene nello stesso ciclo di data path sia possibile eseguire più operazioni (lettura di registri, utilizzo dell'ALU e dello shift register e scrittura su registri), in un ciclo **NON è possibile completare** la lettura o scrittura di parole in memoria.
Infatti, come già detto a proposito di bus sincroni e asincroni, le memorie non sono in grado di far fronte istantaneamente a una richiesta di lettura o scrittura che non può quindi essere conclusa nello stesso ciclo di clock nel quale è stata inoltrata la richiesta.
Nella nostra microarchitettura se viene attivato un segnale di lettura dalla memoria dati (MAR/MDR), l'operazione di lettura ha inizio al termine del ciclo del data path, dopo aver caricato in MAR l'indirizzo. I dati sono disponibili al termine del ciclo seguente in MDR, e quindi possono essere utilizzati solo **due cicli dopo**.
In altre parole una lettura che ha inizio al ciclo k, fornisce dati alla fine del ciclo $k+1$ (quando oramai non possono più essere utilizzati in quel ciclo) e quindi potranno essere utilizzati solo al ciclo $k+2$.
Nel ciclo $k+1$, la CPU non deve necessariamente rimanere inattiva aspettando la memoria, ma può eseguire un ciclo di data path che non necessita del dato in corso di lettura. Per il **fetch di istruzioni** (come vedremo) le cose sono diverse: il "ritardo" è di 1 solo ciclo.

Cicli di data path e microistruzioni
Il microprogramma
Il microprogramma è un insieme di micro istruzioni, la cui funzione è quella di controllare il funzionamento del data path, quest'ultimo riesca ad effettuare i seguenti servizi:

- Il fetch, cioè la fase che consente di trasferire le istruzioni del livello Assembler (ISA) che costituisce il vero programma da eseguire.
- La fase di decode viene effettuata tramite microistruzioni nel data path, ed è utilizzata per riconoscere le istruzioni del livello ISA.
- La fase di excute per eseguire le istruzioni del livello ISA.

Che cosa è una microistruzione?
Una microistruzione (che fa parte del microprogramma, il quale controlla il funzionamento del data path) è una sequenza di bit, formata da tre parti:
Control: stato e l'insieme dei segnali di controllo nel data path per un certo clock.
Address: indirizzo della prossima microistruzione da eseguire.

Attenzione: come vedremo non si riferisce a un indirizzo di memoria esterna ala CPU, ma all'indirizzo di una ROM interna dove sono memorizzati i microprogrammi.

JAM: bit per la gestione di salti condizionali a seconda dei bit di stato N (negative), Z (Zero bit) dell'ALU.
Una microistruzione viene eseguita in un solo ciclo di clock. È possibile che per l'esecuzione di un istruzione del livello Assembler vengono richieste più microistruzioni (o più passi a livello del micro programma).
I possibili segnali di controllo che vengono utilizzati nel data path per abilitare il trasferimento dei vari registri sono 29. Essi servono:

- ✓ per controllare la memorizzazione dei valori sul Bus C nei vari registri
- ✓ per controllare la propagazione di un valore sul Bus B dei registri, scegliendone solo uno per volta
- ✓ per indicare le operazioni di lettura e scrittura verso la memoria attraverso il registro MAR e l'MDR
- ✓ per indicare il fetch attraverso i registri PC e MBR

Esecuzione di ogni singola microistruzione
In ogni ciclo di clock e quindi durante l'esecuzione di ogni singola microistruzione presente nel microprogramma sono letti questi 29 segnali di controllo da una opportuna memoria che registra i vari segnali per ogni microistruzione.
Qualora l'istruzione al livello ISA prevede un'operazione di lettura dalla RAM, essa viene generata nei seguenti tre cicli di clock:
1° ciclo di clock, viene definito il valore per il registro MAR, ovvero sia specifico l'indirizzo della parola di memoria che voglio considerare.
2° ciclo di clock, la memoria assegna il valore della parola specificata nel registro MAR al registro MDR (il trasferimento avviene a 32 bit).
3° ciclo di clock, corrispondente all'esecuzione di un'operazione del livello ISA che prevede la lettura informazione dalla RAM, il valore che è stato letto e che risulta pronto dopo il 2° ciclo di clock, viene utilizzato per effettuare l'operazione.
Da notare che durante il 2° ciclo di clock, prima dell'inizio del 3° ciclo di clock, il processore possa effettuare altre operazioni poiché i Bus del data path risultano indisponibili, e purché esso non acceda alla memoria, in quanto la RAM è già impegnata per leggere la parola di memoria il cui indirizzo è specificato dal registro MAR.
Accesso alla memoria

Nel caso di accesso alla memoria sia in lettura che in scrittura, si verifica che non sempre la memoria riesce a rispondere ad una richiesta di accesso in un solo ciclo di clock; se ciò avvenisse avremmo un hit-rate del 100%. Per disporre di hit-rate elevati si usano memorie cache.

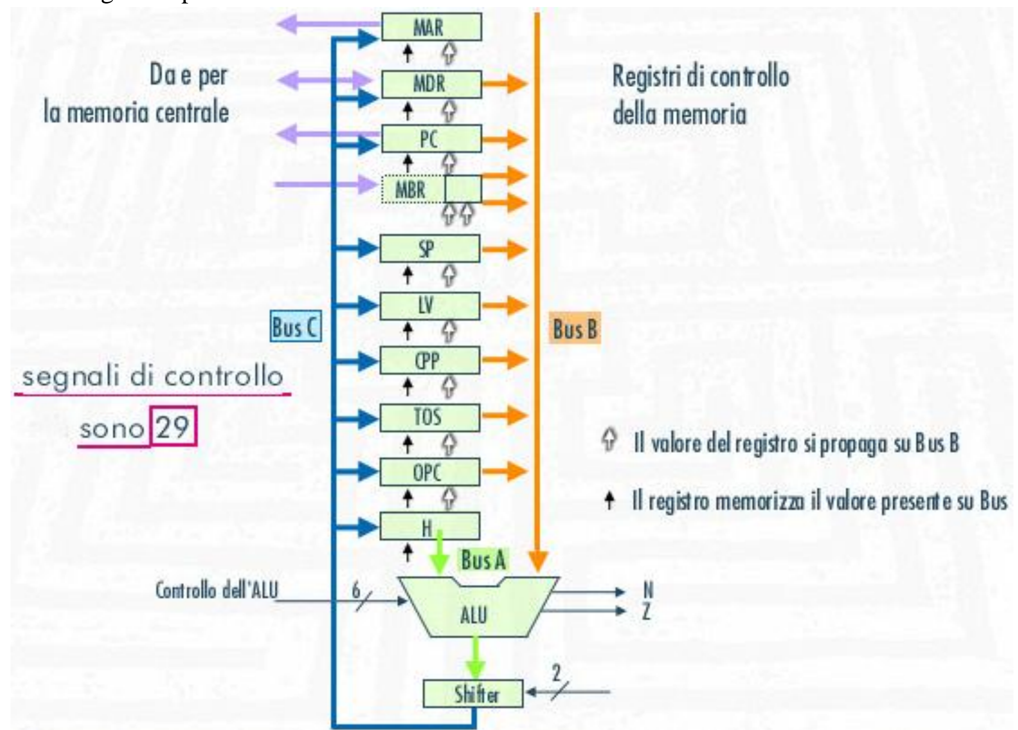
Considerazioni
Sul Bus B un solo registro per volta può propagare il proprio valore memorizzato. In pratica se ad es. è il registro PC che sta propagando il proprio valore sul Bus B nessun altro registro può accede in modalità scrittura allo stesso Bus B.
Al contrario più registri possono memorizzare contemporaneamente al proprio interno il valore che essi osservano sul Bus C.

Codifica delle microistruzioni
Consideriamo ora, come codificare, e poi decodificare, il controllo di Bus B da parte di 9 registri. In particolare disponendo di 9 registri che alternativamente propagano il proprio valore sul Bus B sono necessari 4 bit. Infatti con 4 bit possiamo codificare 10 valori, ed è il numero esatto di bit che dobbiamo utilizzare per poter distinguere i 9 registri in modo univoco. Sarà poi presente all'interno della CPU un decoder che riceve in ingresso questi 4 bit e genera sulle uscite un valore 1 per una sola delle sue uscite. Poiché è uno solo il registro che può scrivere sul Bus B, i segnali di controllo del data path sono 24, in particolare:

- ✓ Disporremo di 9 bit per effettuare la lettura dal Bus C, e quindi per specificare 1 o 2 o tutti i 9 registri che memorizzeranno al proprio interno il valore osservato sul Bus C
- ✓ Disporremo di 4 bit per codificare l'assegnamento al Bus B, e quindi 4 bit per precisare uno solo tra i 9 registri che assegneranno al Bus B il proprio valore
- ✓ Disporremo di 8 bit di controllo per l'ALU e shifter
- ✓ Disporremo di 2 segnali per read/write
- ✓ Disporremo di 1 segnale di controllo per il fetch

In definitiva per ridurre il numero di bit di controllo (da 29 a 24 bit di controllo) la nostra microarchitettura utilizza un **decoder** che con soli 4 bit è in grado di specificare quale dei 9 registri abilitare sul bus B ($2^4 = 16 > 9$!). Ogni microistruzione deve disporre di questi segnali. In aggiunta ogni microistruzione deve indicare quale sarà l'operazione da compiere nel ciclo successivo.
Utilizzando 9 bit per l'indirizzo della prossima microistruzione, 3 bit per il JAM e 24 bit per il controllo, ciascuna delle nostre microistruzioni avrà una lunghezza pari a **36 bit**.

Ovviamente solo un piccolo sottoinsieme delle 2^{36} possibili microistruzioni saranno di una qualche utilità e verranno utilizzate dai microprogrammi.
La specifica di quale sia la microistruzione che deve essere utilizzata e quindi eseguita nel ciclo di clock successivo, viene svolta attraverso due informazioni aggiuntive presenti all'interno della microistruzione.
Le scelte che possono essere applicate sono quattro.



Attenzione!
In questo caso stiamo facendo sempre riferimento al funzionamento del data path e quindi siamo sempre al livello di microarchitettura. Ogni singola microistruzione oltre a specificare i 24 bit di controllo, deve anche specificare qual è la prossima istruzione da eseguire. L'insieme delle microistruzioni costituiscono il microprogramma.

Il linguaggio macchina del processore IJVM

Sunto

Sono illustrati la struttura e il funzionamento del linguaggio macchina (linguaggio Assembler interpretato dal microprogramma eseguito nella microarchitettura Mic-1) del processore IJVM (la versione ridotta per numeri interi della Java Virtual Machine).
Il linguaggio macchina di questo processore è basato sull'uso della pila. Si vedrà un preliminare di come questo processore organizza la struttura della memoria e in particolare come la divide in aree, ciascuna destinata ad una funzione differente, come ad es. contenere il codice eseguibile, i dati etc. continuando esamineremo un aspetto molto delicato che presenta una certa complessità e riguarda il meccanismo di chiamata al sottoprogramma contenuto in un qualsiasi linguaggio macchina che si basa sull'uso di una apposita coppia di istruzioni macchina.
Infine illustreremo le classi di istruzioni del linguaggio macchina, e per ciascuna classe verranno dichiarate le istruzioni componenti fondamentali. Per ogni istruzione è illustrata la struttura e una simulazione del funzionamento.
Saranno mostrati degli esempi per capire come funzionano i programmi scritti in linguaggio macchina IJVM, il quale è una versione adeguata ai numeri interi. Tuttavia il processore è in grado di operare anche con dati diversi dai numeri interi, come per es. numeri reali e dati strutturati e tipo più complessi.

Nota: per praticità a volte ci riferiamo all'architettura dell'insieme d'istruzioni (ISA) con il termine di macroarchitettura, in contrapposizione con la microarchitettura.

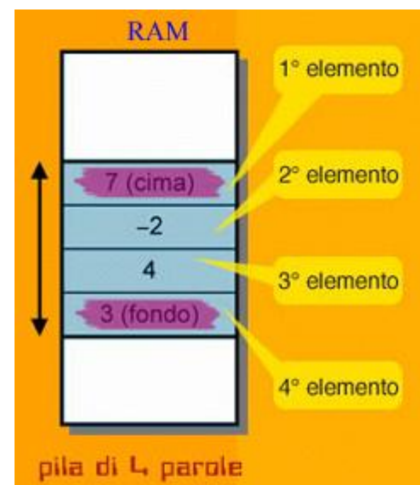
La pila del processore IJVM

Esempio di ISA: IJVM
Obiettivo: illustrare l'architettura di Insieme di Istruzioni.
Problema: servirsi dell'architettura di processore IJVM per numeri interi al fine di realizzare un es. ISA completo.
Supponendo già nota la struttura del processore IJVM, occorre spiegare:
- ✓ La struttura della pila di IJVM e i suoi usi per l'allocazione delle variabili e il calcolo delle espressioni logico-matematiche
- ✓ Il modello di memoria utilizzato dal processore è IJVM, cioè come questo processore utilizza (suddivide) la memoria centrale
- ✓ Il passaggio a sottoprogramma in IJVM
- ✓ L'insieme delle istruzioni che il processore IJVM è in grado di eseguire

Infine mostrerò un es. di programma Java e la traduzione in linguaggio macchina IJVM

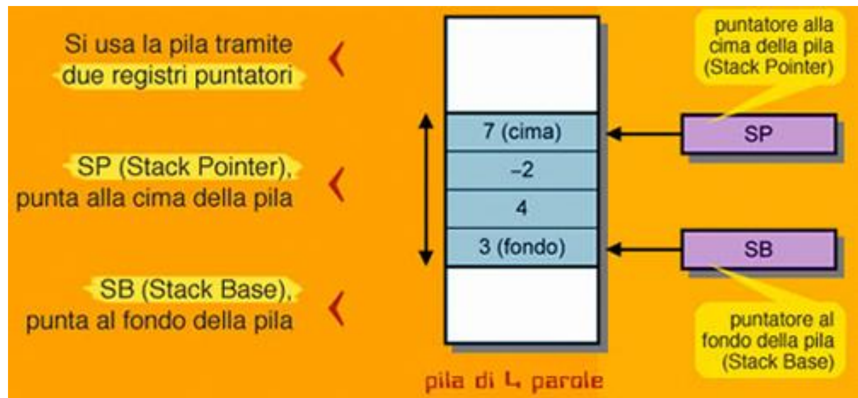
Struttura e funzionamento della pila
La pila è una struttura dati (fondamentale) di memoria per il processore. È formata da una successione di parole di memoria contigue, inoltre ha un fondo (stack base) ed una cima (stack top). In IJVM, si può supporre che ogni elemento (parola/e vista come una successione di bit) che costituisce la pila sia un numero intero decimale.

Esempio di pila
La figura mostra una certa zona della memoria centrale, che costituisce una piccola pila di memoria, inclusa nella RAM. In questo caso la pila è formata da quattro parole. La parte colorata indica il 1° elemento, il 2°, il 3° e il 4°. Quest'ultimo è il fondo della pila, mentre il primo è la cima della pila.



I numeri presenti in questi elementi rappresentano il numero memorizzato della parola. Si vede un -2, è ciò fa pensare che il contenuto di una parola sia codificato in complemento a due, quindi possa rappresentare numeri negativi. Diremo che la cima della pila contiene il numero 7; il fondo contiene il numero 3.

Per utilizzare la pila il processore fa uso di due registri puntatori, i quali contengono gli indirizzi che si riferiscono alla memoria.



Il registro SP (Stack Pointer) o puntatore alla cima della pila (punta sempre alla parola che si trova in cima alla pila).
Il registro SB (Stack Base) o punta al fondo della pila (puntatore alla parola che si trova al fondo della pila).

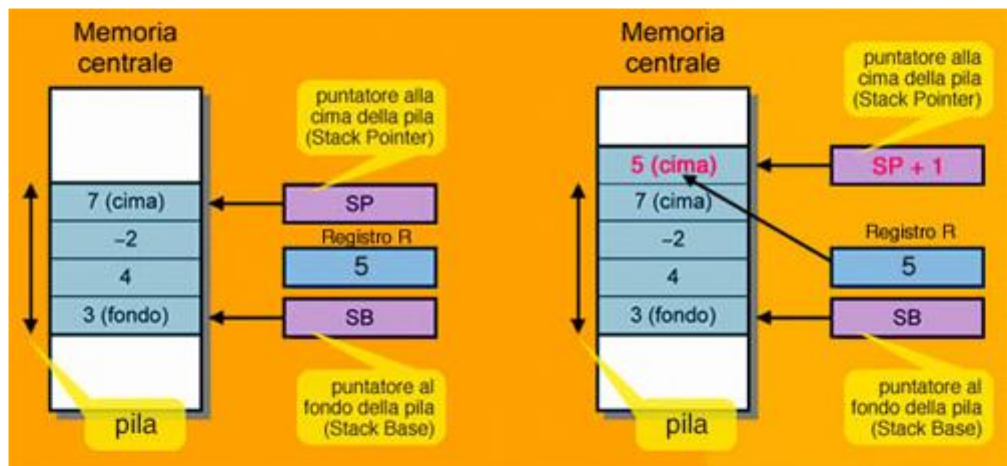
Da ricordare che:
SP contiene l'indirizzo di memoria della cima della pila, che è il 1° elemento della pila.
SB contiene l'indirizzo di memoria del fondo della pila, che è l'ultimo elemento della pila.

Funzionamento della struttura di memoria
Alla pila si possono:
- ✓ Aggiungere elementi sulla cima
- ✓ Oppure togliere elementi dalla cima (se la pila non è già stata vuotata)

L'operazione di aggiunta di un elemento in cima si chiama: PUSH.
L'operazione di cancellazione di un elemento della cima si chiama: POP.
Operazione accessoria: si deve sapere se la pila è venuta oppure se contiene almeno un elemento. Nel caso che la pila fosse vuota, l'operazione POP non si potrebbe applicare.

Funzionamento dell'istruzione PUSH
Si supponga di voler copiare in pila il contenuto del registro R: L'operazione è PUSH R, che significa "metti il contenuto del processore sulla cima della pila".
Per effettuare questa operazione si deve incrementare di 1 il registro puntatore SP (puntatore alla cima della pila).
Scrivere il valore contenuto nel registro R nella cella di memoria che si trova in cima alla pila che è proprio quella puntata da SP.

Esempio di simulazione di operazione sull'uscita R

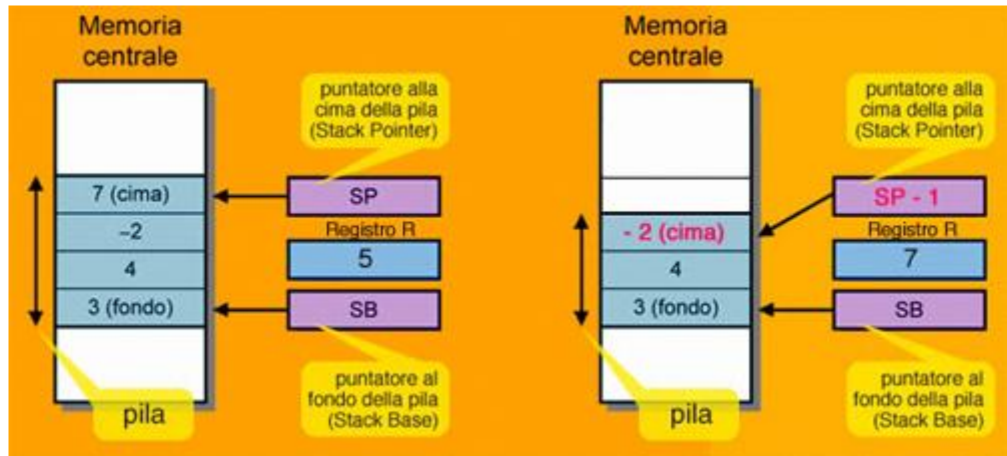


La fig. illustra la situazione iniziale con una pila di quattro elementi, correntemente in cima alla pila vi è il numero 7; il registro R il cui contenuto va scritto in cima allo stack contiene il numero 5.
La fig. a destra mostra che l'elemento 5 è stato aggiunto in cima allo stack, il quale a questo punto contiene cinque elementi, e il registro SP (puntatore alla cima della pila) è stato opportunamente incrementato fino a puntare all'elemento che ora si trova in cima allo stack. Questo completa l'operazione.

L'operazione POP è l'inversa di PUSH
Si supponga di voler togliere il 1° elemento (si trova in cima) dello stack, trasferendolo nel registro R (il contenuto precedente verrà cancellato). L'operazione in esame è POP R.
Si prende il 1° elemento della pila, che si trova nella cella di memoria puntata dal registro SP, e copiarla nel registro R.
Si decrementa di 1 il registro puntatore SP alla cima della pila.

Funzionamento dell'istruzione POP

La configurazione iniziale è mostrata in figura a sinistra. In memoria abbiamo una pila di quattro elementi, correntemente l'elemento 7 è quello che si trova in cima alla pila. Questo elemento andrà copiato nel registro R, il quale correntemente contiene il valore 5 e questo dopo l'operazione di copia andrà nella terza posizione.



La configurazione in figura a destra mostra che il registro R contiene il valore 7 (precedente cima dello stack).
L'elemento -2 è la cima corrente della pila, la quale ha perso un elemento, è passata a tre elementi.
Il registro SP è stato aggiornato in modo da puntare a quella che è correntemente la cima dello stack. Questo conclude l'operazione POP.
Possiamo riassumere in modo sintetico le operazioni PUSH e POP ricorrendo a un minimo di simbolismo.
Indicando con M(SP) la cella di memoria centrale correntemente puntata dal registro SP (suggerisce la notazione che serve per indicare gli elementi di un array, si pensa che la memoria sia un vettore di parole che il registro SP correntemente ne indica uno). Si ha:
L'operazione PUSH R si risolve in questi due passaggi:

SP + 1 → SP // incrementa SP di 1
R → M(SP) // scrive il contenuto del registro R nella cella di memoria puntata da SP

Viceversa per eseguire l'operazione POP R:

M(SP) → R // si legge il contenuto della cella di memoria puntata da SP e si copia in R, mentre il //precedente contenuto di R va preso
SP - 1 → SP // si decrementa il registro SP di 1

Esempio di costante

Sulla pila si può "impilare" (PUSH) anche una costante PUSH 5 vuol dire "metti sulla cima della pila il valore 5".
Non si può invece "spilare" (POP) una costante, non avrebbe senso.
L'operazione POP 5 non è una operazione definita. Sulla cima dello stack potrebbe esserci un valore diverso da 5.

Osservazione

Un valore corrente della cima della pila non si può memorizzare in una costante.

Per utilizzare la pila correntemente occorre verificare alcune precondizioni:

- Prima di iniziare a usare la pila, occorre fissare il fondo della pila, inizializzando il registro SB (Stack base) che punta al fondo della pila
- Se si verifica la condizione SP = SB – 1 la pila è vuota
- A pila vuota è vietato spilare (POP)

La pila mostrata in precedenza cresceva verso l'alto. Questa è una convenzione. Si può ribaltare il funzionamento della pila, facendola crescere verso il basso. In caso di pila ribaltata le operazioni di PUSH R e POP R sono così descritte in simboli:

SP - 1 → SP // si decrementa il registro SP di 1
R → M(SP) // scrive il contenuto del registro R nella cella di memoria puntata da SP

SM(SP) → R
SP + 1 → SP // incrementa SP di 1

La direzione di crescita è invertita, ma gli elementi si tolgono e si aggiungono allo stesso modo nella pila.

Approfondimenti

Uso della pila per il calcolo di espressioni (altra modalità)

- La pila può essere usata per calcolare espressioni logiche-matematiche, contenenti cioè operazioni logiche oppure aritmetico-algebriche.
- Il calcolo di un'espressione può essere sempre ricondotto a una successione di operazioni PUSH e POP, intercalate da operazioni logiche o matematiche.

Esempio
Calcolare 7+(1+2)*x*(5-3). Occorre prima riscrivere l'espressione in forma postfissa (postfix). Creiamo un algoritmo postfix: si parte dagli operatori di 1° livello (gli ultimi da calcolare) e successivamente si eseguono i passi 1, 2 e 3 eventualmente ripetendoli:
1° passo: si sposta l'operatore a destra del 2° operando
2° passo: si cancellano le eventuali parentesi attorno gli operandi
3° passo: si passa agli operatori di livello immediatamente superiore, tornando al 1° passo.
Quando non ci sono più operatori, si termina.

Analisi dell'espressione nella forma postfix
Ecco l'intera distribuzione di operatori e relativi operandi: in essa vi è l'analisi preliminare per illustrare la trasformazione dell'espressione in forma postfix:
L'operatore di 1° livello è l'operatore + ed è l'ultimo operatore ad essere calcolato.
L'operatore di 2° livello è l'operatore *x*
Vi sono anche degli operatori di 3° livello dentro le parentesi, che vengono calcolati simultaneamente.

Trasformazione nella forma postfix
Poniamo in evidenza quali sono gli operandi associati a ciascuno degli operatori; per es. l'operando 7 è il primo di questo operatore +; il 2° operando costituito da (1+2)*x*(5-3).
Si procede in modo simile con gli altri operatori contenuti nell'espressione:
il 1° operando dell'operatore *x* che è l'intera sottoespressione (1+2);
il 2° operando dell'operatore *x* che è l'intera sottoespressione (5-3);
Adesso si può procedere più velocemente:
1° operando dell'operatore +;
2° operando dell'operatore -.

Calcolo tramite la pila
Simulazione del procedimento di calcolo. All'inizio la pila è vuota, leggiamo l'espressione in forma postfix eseguendo la scansione da sinistra verso destra.
Sono presenti:

- L'operando 7, eseguiamo l'operazione PUSH (inserisco il valore 7 in pila) → la pila contiene un elemento.
- Il valore 1, eseguendo l'operazione PUSH di 1) → che 1 viene impilato nella pila sopra il valore 7.
- L'operando 2, eseguiamo PUSH 2 → che 2 è stato impilato nella pila sopra il valore 1
- L'operatore +, prelevo gli elementi in cima alla pila e scriviamo al loro posto il risultato della somma:
  7 1 2 … POP; POP; PUSH 1 + 2
  7 1 2 … PUSH 5 *(in pila troviamo 3 e 7)*
- L'operando 5, eseguendo l'operazione PUSH 5 → che la pila contiene 5 e 7
- L'operando 3, eseguiamo PUSH 3 → che la pila contiene 3 5 3 7, quindi:
  7 1 2 + 5 3 … POP; POP; PUSH 5 – 3
- L'operatore -, vuol dire eseguire la sottrazione dei due elementi in cima alla pila: leggiamo i due elementi: POP; POP;
- Mettiamo sulla cima della pila, il risultato della sottrazione 5 – 3:
  PUSH 5 – 3. Allora sulla cima della pila ho 2 e poi 3 e 7.
- L'operatore *x*, vuol dire eseguire il prodotto in cima alla pila dei due elementi:
  vengono letti POP; POP;
- Mettiamo sulla cima della pila, il risultato della moltiplicazione 3*x*2, quindi: PUSH 3*x*2.
  7 1 2 + 5 3 - … POP; POP; PUSH 3*x*2, ottenendo la nuova configurazione della cima della pila: 6 e 7.
- L'operatore +, leggiamo i due elementi, poi eseguo la somma, infine con l'operazione di PUSH ottengo il risultato in cima alla pila:
  7 1 2 + 5 3 – *x* … POP; POP; PUSH 7 + 6
- Il risultato 13 è in pila.

Siamo arrivati in fondo all’espressione, idealmente dobbiamo eseguire un’ultima operazione POP che legge l’unico elemento contenuto nella pila:

7 1 2 + 5 3 – x +… POP

La pila torna ad essere vuota come inizialmente lo era.

Riassunto della sequenza di operazioni PUSH/POP che operano sulla pila, secondo l’ordine di esecuzione delle varie operazioni. La fig. mostra l’ordine di esecuzione con l’uso di registri temporanei R1, R2, R3 per memorizzare temporaneamente i valori che vengono letti dalla pila tramite delle operazioni POP e anche il risultato dei calcoli aritmetici intermedi, per es. R2 + R1 e temporaneamente scritto in R3, per poi essere subito descritto in pila con PUSH R3.

Si usano i registri temporanei R1, R2 e R3

• PUSH 7	• PUSH 5	• POP R2
• PUSH 1	• PUSH 3	• R3 = R2 × R1
• PUSH 2	• POP R1	• PUSH R3
• POP R1	• POP R2	• POP R1
• POP R2	• R3 = R2 – R1	• POP R2
• R3 = R2 + R1	• PUSH R3	• R3 = R2 + R1
• PUSH R3	• POP R1	• PUSH R3

In pila resta un solo elemento, il risultato

Scrittura di un programma eseguibile dal processore IJVM

Si introducono le istruzioni seguenti:

- IADD (Integer ADD – somma di due numeri): togli i due numeri in cima allo stack, addizionali e scrivi il risultato della somma in cima alla pila
- ISUB: togli i due numeri in cima allo stack, esegui la sottrazione (per hp. il primo numero tolto sia il sottraendo, il secondo il minuendo) e scrivi la differenza in cima alla pila
- IMUL: togli i due numeri in cima allo stack, moltiplicali e scrivi il prodotto del risultato in cima alla pila

Queste istruzioni fanno parte dell’insieme delle istruzioni del processore IJVM. La lista delle operazioni esaminata prima, alla luce di queste nuove istruzioni può essere riscritta secondo l’ordine mostrato in figura. Ciò riproduce il programma di calcolo in linguaggio macchina IJVM della simulazione precedente.

Simulazione dell’esecuzione del programma IJVM per il calcolo dell’espressione

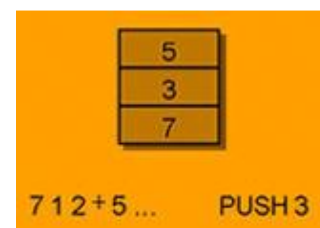
Abbiamo l’espressione nella forma postfix: 7 1 2 + 5 3 – x +. All’inizio la cima è vuota.

Eseguiamo:

- l’istruzione PUSH 7, la pila contiene l’elemento 7.
- l’istruzione PUSH 1, che per essa, sopra l’elemento 7 viene impilato 1.
- l’istruzione PUSH 2, che per essa, sopra l’elemento 1 viene impilato 2.

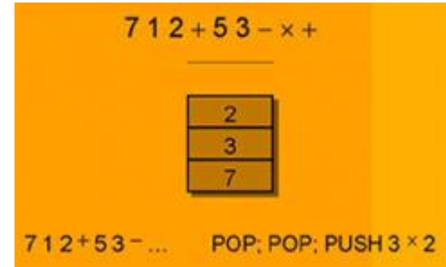
- l’istruzione IADD, preleva i due elementi in cima alla pila, toglie questi, e mette in cima allo stack il risultato della somma eseguita, cioè 3.

- Le istruzioni PUSH 5 e PUSH 3, avendo nella pila 3 5 3 7.

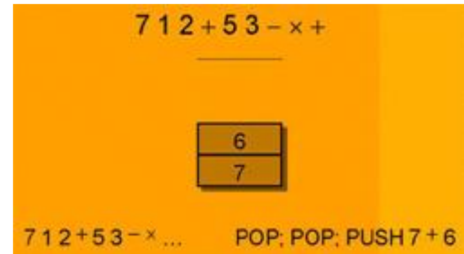


- L’istruzione ISUB, essa toglie i due elementi in cima alla pila, sottraendoli e mette al loro posto il risultato 2, avendo sulla pila 2 3 7.

- L'istruzione IMUL, essa toglie i due elementi in cima alla pila, calcola il prodotto e scrive il risultato 6.

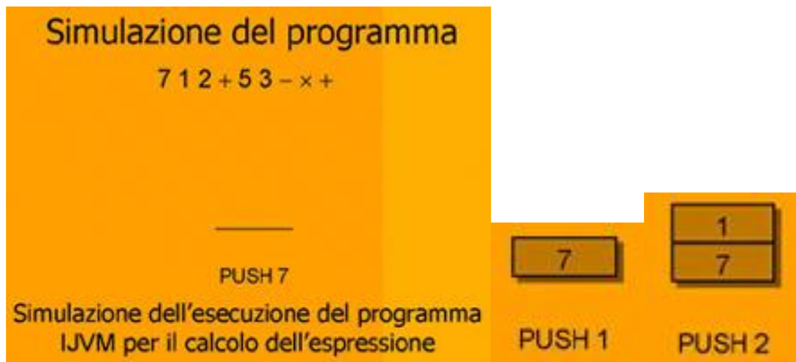


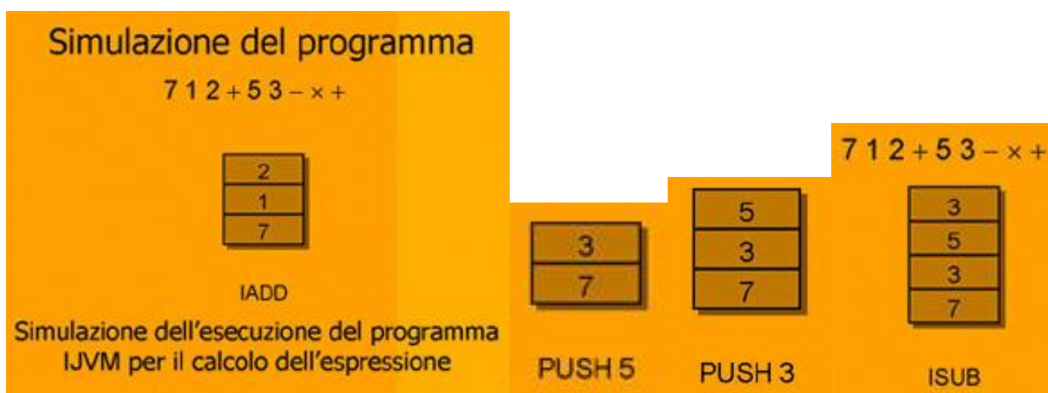
- L'istruzione IADD dei due elementi presenti nella pila, ed il risultato è 13, che è l'ultimo elemento contenuto nella pila.



Il programma di calcolo per il IJVM

- PUSH 7 // scrivi 7 sulla cima della pila
- PUSH 1 // scrivi 1 sulla cima della pila
- PUSH 2 // scrivi 2 sulla cima della pila
- IADD // addiziona i 2 numeri in cima alla pila
- PUSH 5 // scrivi 5 sulla cima della pila
- PUSH 3 // scrivi 3 sulla cima della pila
- ISUB // sottrai i 2 numeri in cima alla pila
- IMUL // moltiplica i 2 numeri in cima alla pila
- IADD // addiziona i 2 numeri in cima alla pila





Considerazioni
Il calcolo delle espressioni si generalizza anche ad:
- ✓ Espressioni contenenti variabili o costanti
- ✓ Espressioni contenenti operatori booleani (logici)
- ✓ Espressioni contenenti operatori relazionali (=>, <=, >, <, …)
- ✓ Espressioni contenenti operatori algebrici
- ✓ Espressioni con numeri reali, complessi (divisioni, radici, …)

Il compilatore analizza l'espressione ne calcola la forma postfix con l'algoritmo visto e genera il programma di calcolo dell'espressione.

Approfondimenti: Esempio di ISA: IJVM

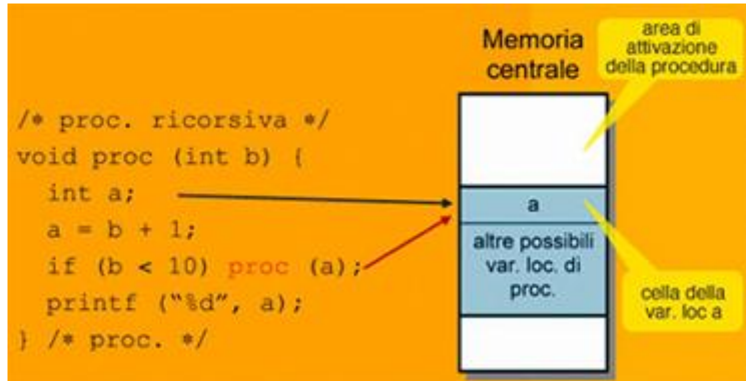
Uso della pila per l'allocazione di variabili: (Stack).
Introduciamo il livello ISA della macchina che sarà interpretato dal microprogramma eseguito nella microarchitettura Mic-1. Le procedure sono dotate di un insieme di variabili locali. Si accede a queste variabili dall'interno della procedura, ma ciò diventa impossibile una volta che la procedura termina.
In quale parte di memoria bisogna memorizzare queste variabili? La soluzione più semplice, che consiste nell'assegnare a ciascuna variabile un indirizzo assoluto della memoria non funziona. Quando la procedura viene invocata due volte, è impossibile memorizzare le sue variabili in locazioni assolute della memoria perché la seconda invocazione può interferire con la prima. Allora utilizziamo un'area di memoria, detta stack nella quale non vi sono indirizzi assoluti per le singole variabili.
Abbiamo visto la struttura è il funzionamento di base della pila, cioè le operazioni PUSH e POP.
Vediamo alcuni usi di questa struttura dati. la pila ha un ruolo molto importante nel linguaggio di programmazione per almeno due motivi:
il linguaggio di programmazione come il C, Java ed altri, permettono di avere:
- o Procedure e funzioni dotate di variabili locali
- o Procedure e funzioni ricorsive

La pila ha un ruolo essenziale nella gestione dell'uso corretto delle variabili locali e sul meccanismo di ricorsione.

La problematica delle variabili locali e della ricorsione
Ogni procedura o funzione ha il proprio insieme di variabili locali; si potrebbe supporre di assegnare alla procedura (o funzione) un'area diversa di memoria centrale, in cui la procedura tiene le proprie variabili locali. Si presenta però un problema: se la procedura è ricorsiva, la seconda chiamata della procedura si troverebbe ad operare sulle stesse variabili locali della prima (che per la programmazione è un errore – le chiamate ricorsive ad una procedura operano su variabili locali che in generale sono distinte).

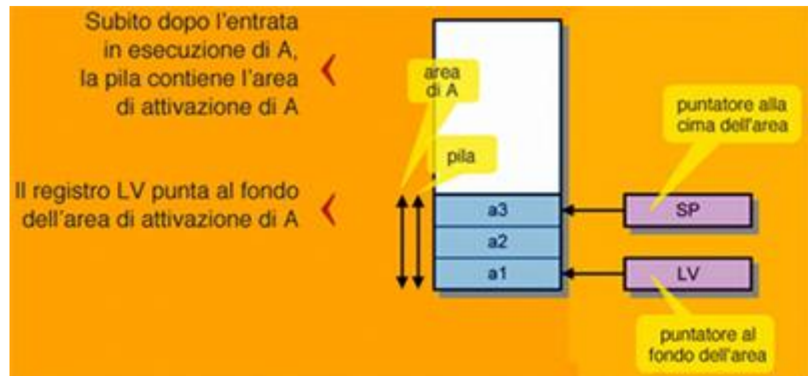
Esempio
Abbiamo una procedura del tipo 'intero' che non restituisce alcun valore (tipo void) ed effettua qualche calcolo. La procedura è una variabile locale di tipo intero chiamata 'a'. Per operare correttamente la procedura deve avere assegnata una cella di memoria su cui tenere memorizzata la variabile intera 'a'. pertanto nel momento in cui la procedura entra in attività essa si servirà della variabile 'a' memorizzata nella cella di memoria indicata.

Come si vede dall'elenco delle istruzioni (fig.) della procedura, tale procedura è ricorsiva perché richiama se stessa in un certo punto del proprio codice. Questa seconda chiamata si troverà ad operare sulla stessa variabile locale della prima. Ciò non è un procedimento corretto nel caso della ricorsione. Non è dunque possibile allocare le variabili locali di procedure (o funzioni) in celle di memoria di indirizzo assoluto (cioè un indirizzo fissato una volta per tutte durante la compilazione del programma). Occorre utilizzare una strategia diversa che si può realizzare mediante la pila di memoria.

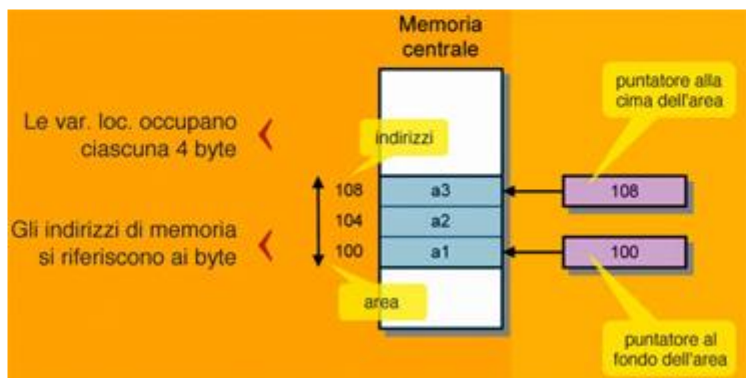


Funzionamento dell'assegnazione di variabili locali alle procedure (o funzioni)
Ogni procedura possiede le proprie variabili locali, le quali vengono allocate in successione in un'area di memoria chiamata "area di attivazione" (o frame) della procedura.

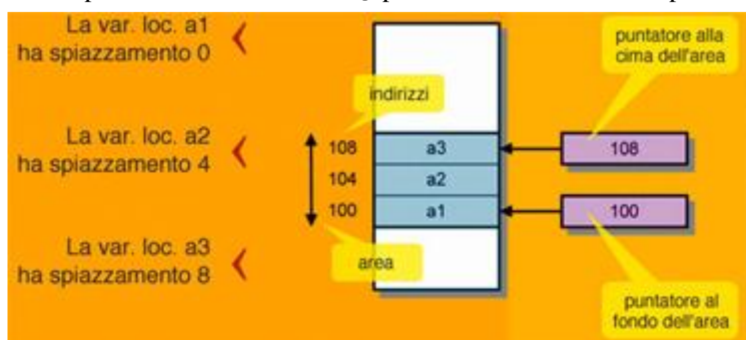
Le aree di attivazione delle procedure vengono impilate (quando la procedura viene attivata – operazione di PUSH) oppure spilate (quando la procedura termina – operazione POP eseguita sulla pila). La fig. mostra la struttura di una generica area di attivazione. Vi è una procedura dotata di tre variabili locali a_1, a_2 e a_3 le quali sono allocate in celle di memoria contigue che si trovano nella pila. Il registro SP (**S**tack **P**ointer) punta alla cima dell'area di attivazione (a_3), della procedura che correntemente è in esecuzione. Inoltre abbiamo un altro registro LV (**L**ocal **V**ariable) che punta al fondo (attuale) dell'area di attivazione (a_1) – LV punta alla procedura corrente.



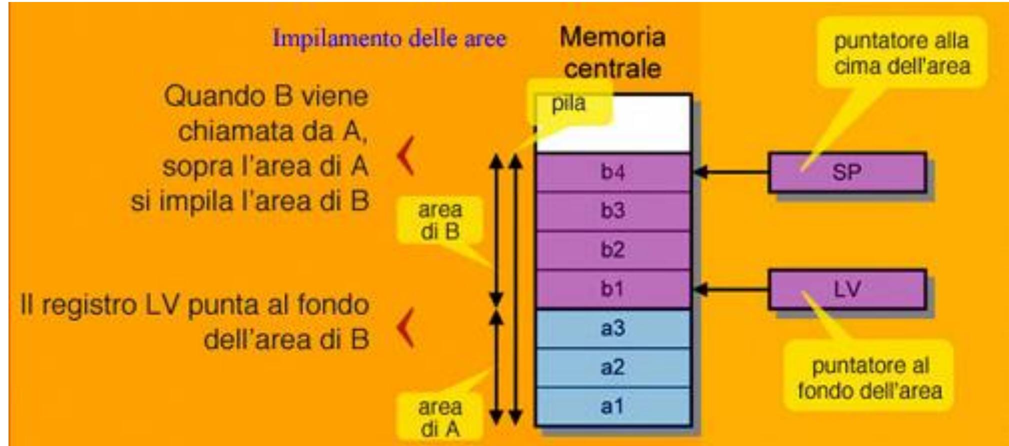
Osservazione
Le variabili locali di tipo intero occupano ciascuna 4 Byte. Gli indirizzi di memoria si riferiscono invece ai singoli Byte. Quindi passando da una parola di memoria, a quella successiva nella pila, gli indirizzi hanno un incremento di 4 unità come mostrato in figura.
Ogni variabile locale è collocata con un certo spiazzamento (offset) all'interno dell'area di attivazione. La variabile locale al fondo dell'area di attivazione ha spiazzamento 0.



La fig. seguente mostra alcuni spiazzamenti, per esempio la variabile locale a_1 presente sul fondo della pila ha spiazzamento 0.
La variabile locale a_2 ha spiazzamento 4.
La variabile locale a_3 ha spiazzamento 8.
Le aree di attivazione di procedure diverse vengono impilate l'una sopra l'altra nella pila di memoria a seconda dell'ordine di attivazione delle procedure.



Esempio
La procedura A ha un'area di attivazione contenente tre variabili: a_1, a_2 e a_3.
La procedura B ha un'area di attivazione contenente quattro variabili: b_1, b_2, b_3 e b_4.
La procedura A, ad un certo punto della sua esecuzione, chiama la procedura B.
La fig. mostra la config. iniziale, nella pila è presente l'area di attivazione di A contenente le tre variabili a_1, a_2 e a_3.
Il registro SP punta alla cima dell'area di attivazione, LV punta al fondo dell'area di attivazione.
Subito dopo, che la procedura A ha chiamato B, quindi B è stata attivata, sulla pila sono stati inseriti (impilata) l'area di attivazione di B (colore viola).

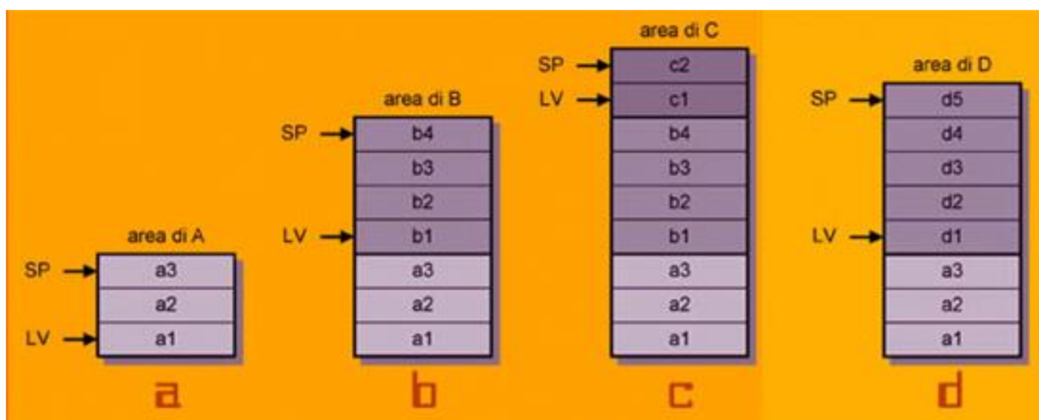


In questa area vi sono quattro variabili di B; il registro SP punta alla cima dello stack e anche la cima dell'area di attivazione di B, mentre LV punta al fondo dell'area di attivazione di B.
Quando si impila l'area di attivazione della procedura B, il registro LV viene incrementato in modo da farlo puntare al fondo dell'area di attivazione di B, e non più di A.
In ogni istante, le variabili locali appartenenti alla procedura correntemente in esecuzione, che si trovano nella sua area di attivazione sono sempre indicate tramite i rispettivi spiazzamenti rispetto al registro LV.

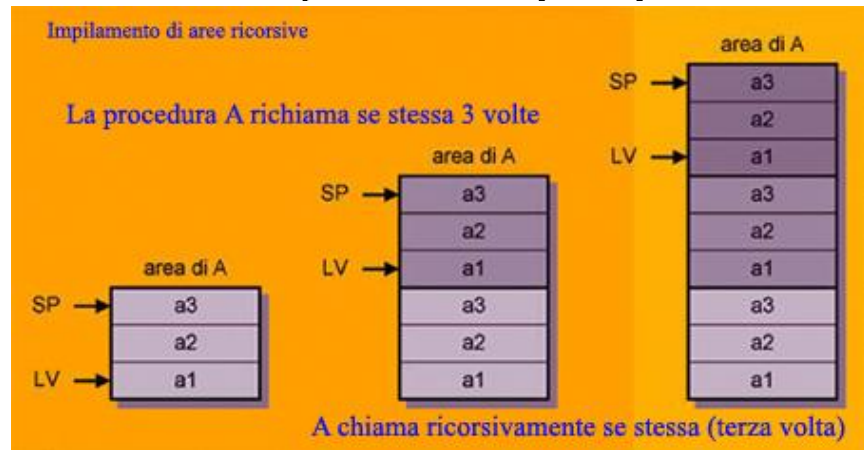
Esempio
Si consideri la sequenza multipla di chiamate a procedure; distinguiamo quattro procedure e la successione di eventi è la seguente:
- ✓ La procedura A entra in azione
- ✓ A ad un certo istante chiama B
- ✓ B a sua volta chiama C
- ✓ C ad un certo istante conclude senza chiamare altre procedure e anche B termina
- ✓ A riprende l'esecuzione, proseguendo, è chiama la procedura D

Sulla figura vediamo la successione di impilamenti e sfilamenti di aree di attivazione è determinata da questa successione di chiamate a procedura:
- ➢ Entra in esecuzione la procedura A, così viene impilata la relativa area di attivazione
- ➢ Entra in esecuzione la procedura B, e viene impilata la relativa area di attivazione sistemata sopra quella di A
- ➢ Entra in esecuzione la procedura C, e viene impilata la relativa area di attivazione

In questa situazione il contenuto della pila indica che A è sospesa, in attesa della terminazione di B, la quale a sua volta è sospesa in attesa della terminazione di C.
Le procedure C e B terminano e le relative aree di attivazione vengono spilate; entra in esecuzione D.
L'area di attivazione della procedura D si trova ora impilata direttamente sopra l'area di attivazione della procedura A.
Le aree di attivazione di B e C non ci sono più perché queste procedure sono definitivamente terminate.



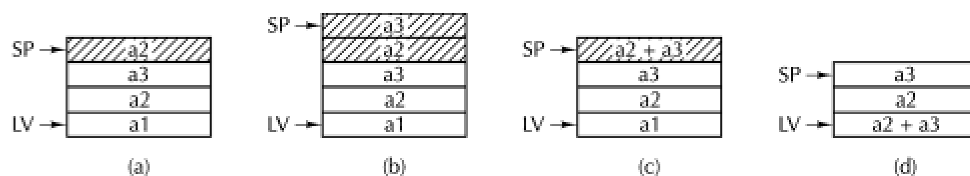
Il *meccanismo di impilamento* delle aree di attivazione delle varie procedure è anche in grado di gestire correntemente le procedure ricorsive. Infatti in questo modo le variabili locali di chiamate successive e ricorsive di una medesima procedura rimarranno separati in pila.



Ad ogni chiamata ricorsiva, si impila una nuova area di attivazione, che è identica come struttura a quella precedente, ma in una posizione diversa all'interno della pila.
Possiamo vedere tre chiamate ricorsive della procedura A con una simulazione:

- *A* viene attivata nella relativa area di attivazione, impilata,
- e ad un certo istante *A* chiama ricorsivamente se stessa, in una nuova area di attivazione di A, viene impilata, che come struttura è identica alla precedente con una posizione diversa.

Questa 2ª chiamata ricorsiva richiama nuovamente A, e viene quindi impilata una 3ª area di attivazione di A identica alle due precedenti.
Si potrebbe proseguire in questo modo, fin quando ad un certo punto le chiamate ricorsive inizieranno a rientrare, e le aree di attivazione verranno spilate progressivamente.
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 di affrontare questa somma consiste nel porre *a2* in cima allo stack, come mostrato nella fig. (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 1° operando. In seguito a3 viene posto in cima allo stack, come mostra la fig. (b).
il calcolo effettivo può a questo punto essere realizzato eseguendo un'istruzione che preleva due parole dallo stack, la somma e inserisce il risultato nuovamente nello stack, come mostra la fig. (c). Infine, la parola che si trova in cima allo stack può essere rimossa e memorizzata nella variabile locale *a1*, fig. (d). Il blocco delle variabili locali e lo stack degli operandi possono essere messi insieme tra loro. Per esempio quando si calcola un'espressione come $x^2 + f(x)$, una parte dell'espressione (per es. 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.
Da 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.

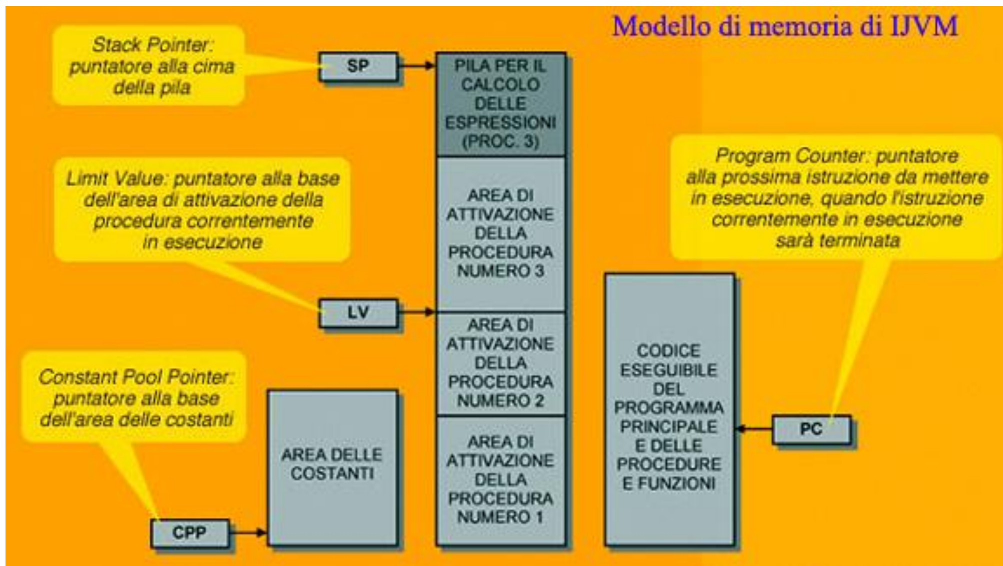


Modello di memoria e sottoprogramma (di IJVM)
Introduzione
Il programma in generale è suddiviso in parti che hanno funzioni differenti, per es. il codice eseguibile e le variabili. Nei prossimi paragrafi vedremo il modello di memoria del processore IJVM, ovvero come questo processore struttura la memoria e la divide in aree dedicando ciascuna area a una funzione particolare.

Strutturazione della memoria
La memoria centrale nel processore IJVM è strutturata in aree (suddivisione non necessariamente contigue) aventi usi diversi:
Un'area a seconda dell'uso che le stato assegnato può contenere:
Codice eseguibile, costanti (dati fissi), dati (variabili), riferimenti ad altre aree (puntatori) o altri tipi di informazioni.
Il processore IJVM dispone di uno spazio di memoria centrale di 4 GB, cioè 4G x 8 bit (parole da 8 bit). È anche possibile vedere la memoria come un vettore di 1G, cioè 1G x 32 bit, raggruppando i Byte a gruppetti di quattro, da formare parole da 32 bit.

Le istruzioni macchina del processore IJVM non vedono direttamente gli indirizzi della memoria centrale. Esse operano in modo implicito sulla memoria, utilizzando degli appositi registri come puntatori (PC, SP, LV, CPP e altri registri).
Per es. le operazioni logico-matematiche vengono calcolate tramite la pila delle espressioni (SP, puntatore alla cima della pila). Come già detto, la RAM del processore IJVM è suddivisa in quattro aree principali:

a) Area delle costanti (Constant Pool Area) *Porzione costante di memoria*
b) Pila delle aree di attivazione (Frame Stack Area)*Blocco delle variabili locali*
c) La pila per il calcolo delle espressioni (Operand Stack Area) *Stack degli operandi*
d) L'area del codice eseguibile (Excutable Code Area) *Area dei metodi*
La schematizzazione seguente mostra le quattro aree indicate sopra. Vi sono inoltre i registri la cui funzione è quella di manipolare e gestire queste differenti aree.



L'area delle costanti contiene delle costanti numeriche e anche di altro tipo, come per es. stringhe fisse di caratteri che servono nel corso dell'esecuzione del programma e puntatori ad altre aree di memoria. L'area viene inizializzata quando il programma viene caricato in memoria. Il contenuto dell'area delle costanti non può essere modificato.

Il programma che deve utilizzare delle costanti, accede all'area ad esse dedicate tramite il registro puntatore CPP.
L'area più fondamentale è ***l'area di attivazione*** del programma principale della procedura e della funzione.

- Essa contiene alcuni puntatori di servizio, usati nelle gestione del meccanismo di chiamata-ritorno di sottoprogramma
- Essa contiene i parametri passati alla procedura o funzione all'atto della sua chiamata
- Essa contiene lo spazio riservato per le variabili locali della procedura
- Ogni qual volta che una procedura o funzione associata si attiva, la rispettiva area di attivazione viene allocata sulla cima delle pila
- La procedura correntemente in esecuzione usa le variabili locali e i parametri contenuti nella propria area di attivazione
- Quando la procedura correntemente in esecuzione termina, l'area di attivazione in cima alla pila viene eliminata
- Sempre in pila, ma sopra le aree di attivazione del programma principale dell'eventuale procedura o funzione correntemente in esecuzione (sopra quest'area di attivazione c'è sempre spazio libero per fare crescere ulteriormente la pila), viene costruita la ***pila per il calcolo delle espressioni*** negli istanti in cui il programma deve calcolare un'espressione logico-matematica di qualche tipo.
  Il registro LV, contenuto nel processore IJVM, viene utilizzato per puntare alla base dell'area di attivazione della procedura correntemente in esecuzione.
  Il registro SP, contenuto nel processore IJVM, punta in ogni istante alla cima della pila, ovvero:

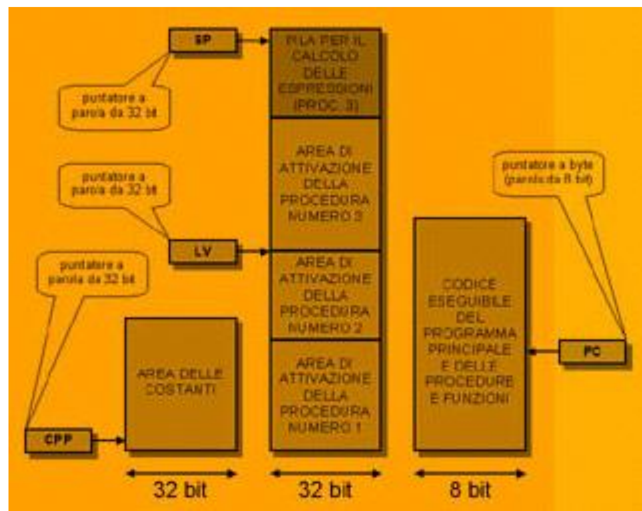
- Alla cima dell'area di attivazione corrente, cioè quella della procedura o funzione che è correntemente in esecuzione
- Oppure alla cima della pila delle espressioni, nei momenti in cui questa non è vuota, cioè quando è in corso il calcolo di un'espressione.
- La pila per il calcolo delle espressioni è una zona di spazio libero, di dimensioni prefissate, riservata per fare crescere la pila per il calcolo delle espressioni logico-matematiche.
  Quando il compilatore genera le sequenze di istruzioni che calcolano le espressioni usando la pila, si assicura che la pila non possa crescere in modo eccessivo rispetto allo spazio di memoria lasciato libero.
- ***L'area del codice eseguibile*** del programma principale e delle procedure e funzioni contiene il codice eseguibile del programma e delle eventuali procedure facenti parte del programma complessivo da eseguire; almeno il programma principale è sempre presente, mentre le procedure e/o funzioni sono facoltative.
- Il contenuto del codice eseguibile viene caricato quando l'intero programma entra in esecuzione. Quando il programma principale esegue l'istruzione di terminazione in linea di principio il contenuto dell'area di codice eseguibile diventa inutile e potrebbe essere cancellato.

- Il registro PC (interno al processore IJVM) serve per gestire l'area del codice eseguibile, in ogni istante questo registro serve per puntare all'interno dell'area di codice eseguibile alla prossima istruzione macchina da mandare in esecuzione, quando l'istruzione macchina correntemente in esecuzione sarà terminata (supponendo che l'istruzione macchina corrente non sia un'istruzione di salto altrimenti la destinazione può essere diversa).

Precisazioni sui registri che servono per gestire le aree – esempio di allocazioni di area

- I registri CPP, LV e SP, contenenti nel processore IJVM, sono puntatori a parole da 32 bit, non a singoli Byte, cioè:
- Leggere alla locazione puntata per esempio dal registro SP significa leggere la parola da 32 bit (non il Byte) situato in cima alla pila.
- Leggere alla locazione puntata per esempio da LV + 1 significa leggere la parola da 32 bit (non il Byte) situata dopo la parola da 32 bit puntata direttamente da LV.
- Il registro PC è un puntatore a Byte, non a parole da 32 bit, cioè:
- Leggere alla locazione puntata da PC significa leggere un Byte (non una parola da 32 bit) facente parte dell'istruzione da mettere in esecuzione.
- Leggere alla locazione puntata da PC incrementato di 1 (PC + 1), significa leggere il Byte successivo all'ultimo Byte letto (non la parola da 32 bit successiva all'ultima parola letta), facente parte dell'istruzione da mettere in esecuzione.

Nella figura seguente sono mostrati i registri più significativi SP, LV, CPP e PC. Per ciascuno di questi è indicato se si tratta di un puntatore ad una parola da 32 bit, oppure un puntatore a singolo Byte, cioè ad una parola da 8 bit.
L'area del codice eseguibile è pensata come una struttura di Byte (successione di Byte). Mentre le rimanenti aree (delle costanti, di attivazione delle procedure o funzioni, della pila delle espressioni sono pensate strutturate con parole da 32 bit successivi).



Insieme delle istruzioni IJVM

Illustriamo la struttura del linguaggio del processore IJVM. Contiene diverse istruzioni macchina, esamineremo:

- ✓ La classe generale di queste istruzioni, cioè come sono ripartite in classi di istruzioni;
- ✓ Il componente di istruzioni di ciascuna classe

Ogni istruzione viene spiegata fornendo la struttura e accennando brevemente al funzionamento.

La figura sottostante mostra dell'insieme delle istruzioni IJVM. Ogni istruzione mostra 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 es. 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, 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 (GO TO) e tre condizionali (IFEQ, IFTL, IF_ICMPEQ). Tutte queste istruzioni, se accettate, modificano il valore del PC in base alla grandezza del loro spiazzamento (16 bit con segno), che si trova nell'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).

L'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.

Esa Codice Mnemonico	Significato
0x10 BIPUSH *byte*	Push del *byte* specificato sullo stack
0x59 DUP	Duplica la prima parola sullo stack (push della copia)
0xA7 GOTO *offset*	Salto non condizionato alla locazione corrente + offset
0x60 IADD	Pop delle due parole dalla cima dello stack, somma e push del risultato
0x7E IAND	Pop delle due parole dalla cima dello stack, AND logico e push del risultato
0x99 IFEQ *offset*	Pop di una parola e salto a locazione corrente + *offset* se la parola è 0
0x9B IFTL *offset*	Pop di una parola e salto a locazione corrente + *offset* se la parola è negativa
0x9F IF_ICMPEQ *offset*	Pop di due parole dallo stack e salto a locazione corrente + *offset* se sono uguali (estrae le due parole in cima allo stack ed effettua una diramazione se sono uguali)
0x84 IINC *varnum const*	Somma la costante const alla variabile locale indirizzata da *varnum*
0x15 ILOAD *varnum*	Push sullo stack della variabile il cui indirizzo è specificato dall'indice *varnum* (che è di 8 bit a meno che non sia stato utilizzato il prefisso WIDE)
0xB6 INVOKEVIRTUAL *disp*	Invoca un metodo; *disp* è l'offset della constant area dove reperire informazioni circa il metodo chiamato
0x80 IOR	Pop delle due parole dalla cima dello stack, OR logico e push del risultato
0xAC IRETURN	Ritorna da un metodo con un valore intero
0x36 ISTORE *varnum*	Pop di una parola dallo stack (Preleva una parola dalla cima dello stack) e la memorizza come variabile locale, di cui è l'indice.
0x64 ISUB	Pop delle due parole dalla cima dello stack, sottrazione e push del risultato
0x13 LDC_W *index*	Push della costante dalla constant area sullo stack. Index è l'indirizzo della costante nella costant area
0x00 NOP	Nessuna operazione
0x57 POP	Cancella una parola dalla cima dello stack
0x5F SWAP	Scambia le due parole in cima allo stack.
0xC4 WIDE	Prefisso; l'istruzione ILOAD o ISTORE successiva ha un indice *varnum* a 16 bit che gli consente di indirizzare più di 256 variabili ...

Generalità sul linguaggio macchina

Il linguaggio macchina IJVM (Integer JVM) è un sottoinsieme del linguaggio JVM completo.

- ➢ IJVM contiene le istruzioni macchina di JVM specializzate per l'elaborazione dei numeri interi (naturali e relativi)
- ➢ IJVM (come anche JVM) ha un modello di esecuzione basato sul processore JVM e sulla pila di memoria (la realizzazione della macchina è basata sull'architettura Mic–1).

Le istruzioni si possono suddividere in classi di istruzioni, che raggruppano quelle che sono logicamente simili, cioè che svolgono funzioni affini.

Le classi di istruzioni del linguaggio IJVM sono:

- Aritmetico-logica. Essa contiene istruzioni che eseguono le principali operazioni aritmetiche e logiche bit per bit (bitwise) su numeri interi.
- Caricamento-memorizzazione. Esse servono per leggere e scrivere le informazioni in memoria
- Salto (condizionato e non). Questa classe serve per controllare il flusso di esecuzione del programma
- Manipolazione della pila. Serve per scrivere e leggere in pila di memoria

- Controllo. Serve per la gestione del processore

In seguito utilizzeremo numeri interi rappresentati in notazioni esadecimale. La rappresentazione esadecimale (hex) è di tipo posizionale, in base 16:

Tipo	Cifre elementari															
hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

$$C4 = 12 * 16^1 + 4 * 16^0 = 196$$
$$1A7E = 1 * 16^3 + 10 * 16^2 + 7 * 16^1 + 14 = 68222$$

In Java i numeri esadecimale si scrivono con il prefisso 0x; per es. 0xC4, 0x1A7E.
Per trasformare un *numero esadecimale in binario* è necessario espandere ogni cifra hex in un gruppo di 4 bit:
1A7E hex = 1/A/7/E = $(0001\ 1010\ 0111\ 1110)_2$
Da *binario a esadecimale,* si compatta a gruppo di 4 bit in una sola cifra hex.

Esempio
11 0000 1110 1010 = $(3\ 0\ E\ A)_{16}$
Importante
1 Byte = 2 Byte *hex*
1 parola da 32 bit = 8 cifre *hex*

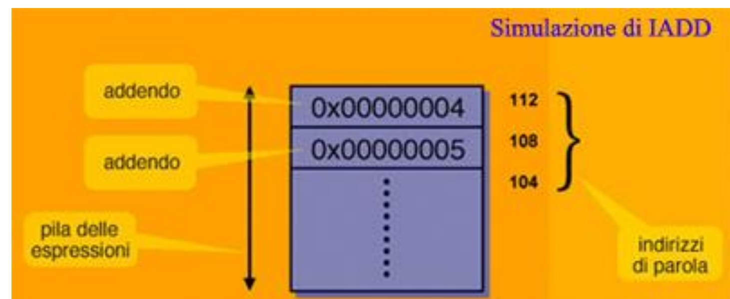
<u>Classi di istruzioni del linguaggio macchina</u>
La classe delle istruzioni aritmetiche-logiche contiene istruzioni per il calcolo di operazioni. Le istruzioni che consideriamo sono:

Hex	Mnemonico	Funzionamento: operazione e risultato
0x60	IADD	Addiziona i 2 elementi in cima alla pila
0x64	ISUB	Sottrae i due elementi in cima alla pila
0x7E	IAND	AND bit-per-bit dei 2 elementi in cima alla pila
0x80	IOR	OR bit-per-bit dei 2 elementi in cima alla pila
0x84	INC numvar valore	Addiziona "valore" alla variabile locale "numvar"

- **IADD**, **ISUB**, **IAND** e **IOR** che operano sulla pila per il calcolo delle espressioni, con parole da 32 bit.
- L'istruzione **INC** opera sulla variabili locali da 32 bit, ma con valore da 1 Byte

- Simulazione di **IADD**

Si addizionano due parole da 32 bit; gli addendi vengono tolti dalla pila, e al loro posto è scritto il risultato.
Abbiamo due addendi, due parole da 32 bit in cima alla pila, espressi in formato esadecimale, quindi abbiamo una successione di 8 cifre esadecimale.

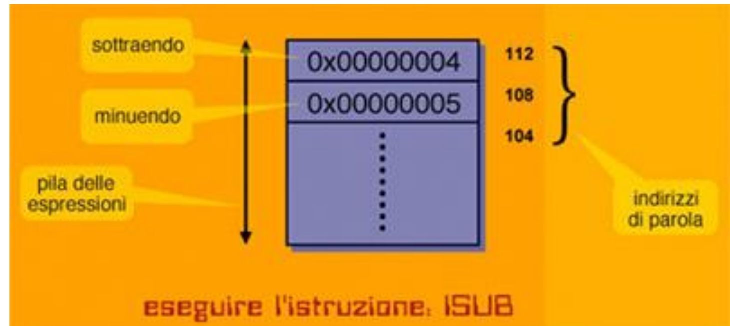


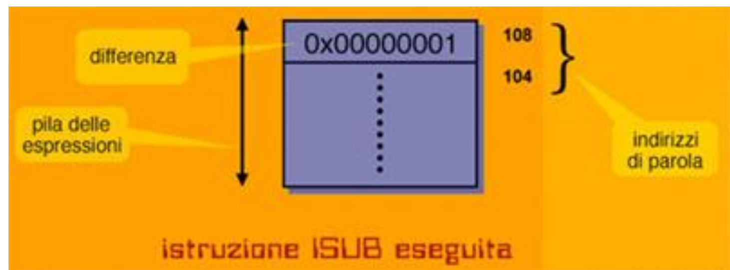
Eseguendo l'addizione dei due addendi, otteniamo un risultato che viene collocato in cima alla pila, in questo caso 9.

- Simulazione di **ISUB**

Si sottraggono due parole da 32 bit, il minuendo e sottraendo vengono tolti dalla pila, scrivendo in cima alla pila il risultato.

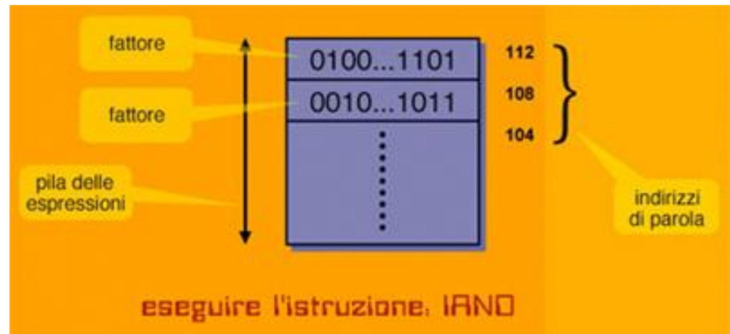
Lo stato della pila prima dell'esecuzione dell'istruzione ISUB, subito dopo aver eseguito la sottrazione abbiamo 1.

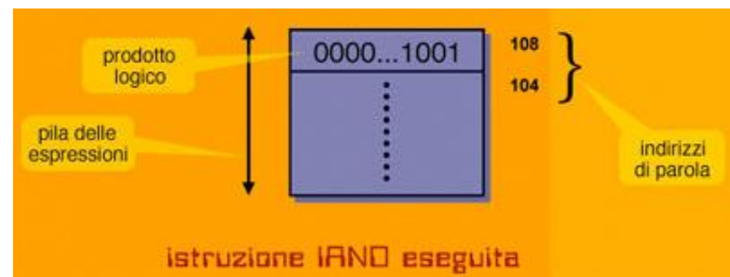




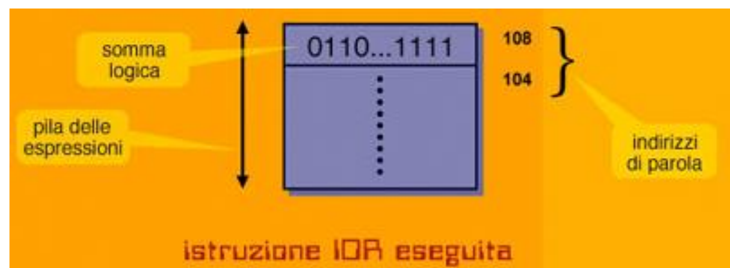
- Simulazione di **IAND**

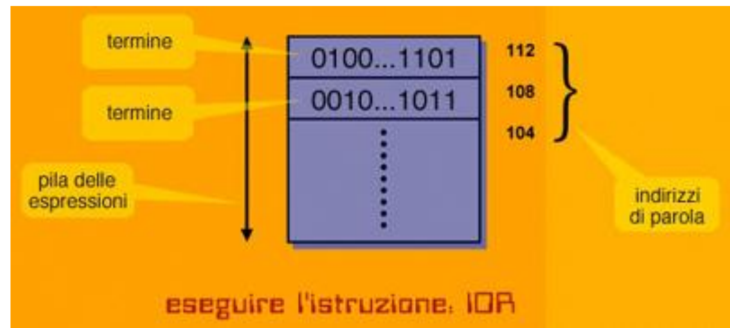
Si calcola il prodotto logico bit per bit di due parole (32 bit); i fattori vengono tolti dalla pila, e viene scritto in cima alla pila il risultato





- Simulazione di **IOR**. Si calcola la somma logica bit per bit di due parole da 32 bit; i termini vengono tolti dalla pila. Dopo aver effettuato la somma, avremo sulla pila della pila il risultato delle due parole.



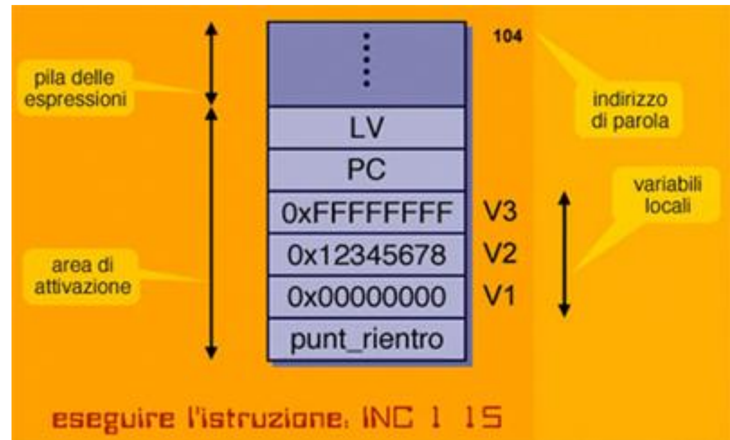


Simulazione di **INC**. effettuare la somma di una costante da 8 bit addizionata ad una variabile locale da 32 bit.

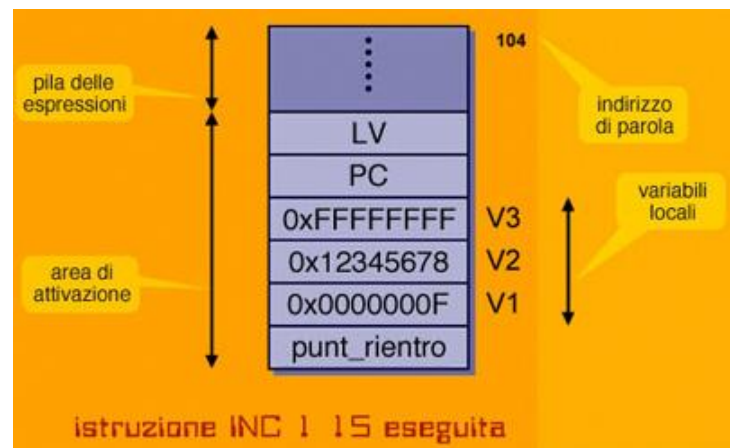
Lo stato della pila prima di eseguire l'istruzione: INC 1 15.

Significato dell'espressione INC 1 15:
alla variabile locale come spiazzamento 1 verrà addizionata la costante 15.

In questo caso hanno importanza gli elementi del contenuto dell'area di attivazione della procedura che esegue l'istruzione INC.



La variabile locale verrà sommata con la costante 15. La variabile locale v_1 è stata incrementata di 15 (notazione decimale la cifra meno significativa vale 15). La pila per il calcolo delle espressioni, nel caso dell'istruzione INC non è coinvolta.



Istruzioni di caricamento-memorizzazione

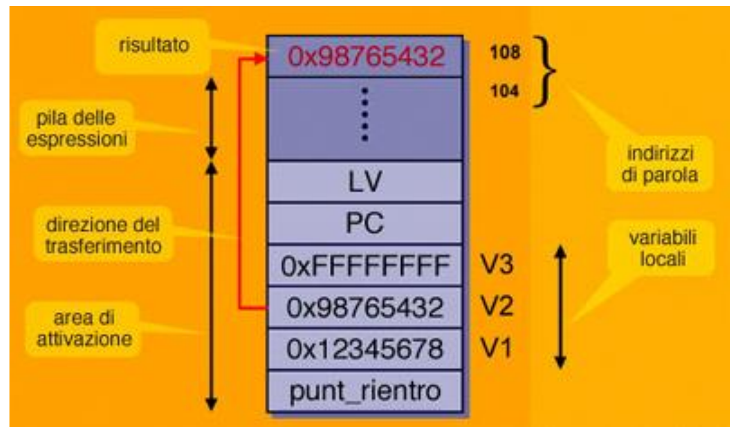
Le istruzioni sono due:

Hex	Mnemonico	Funzionamento: operazione e risultato
0x15	ILOAD numvar	Push della variabile locale "numvar" sulla pila
0x36	ISTORE numvar	Pop dell'elemento sulla pila nella var. loc. "numvar"

Istruzione **ILOAD** numvar ha la funzione di PUSH della variabile locale “numvar” sulla pila.

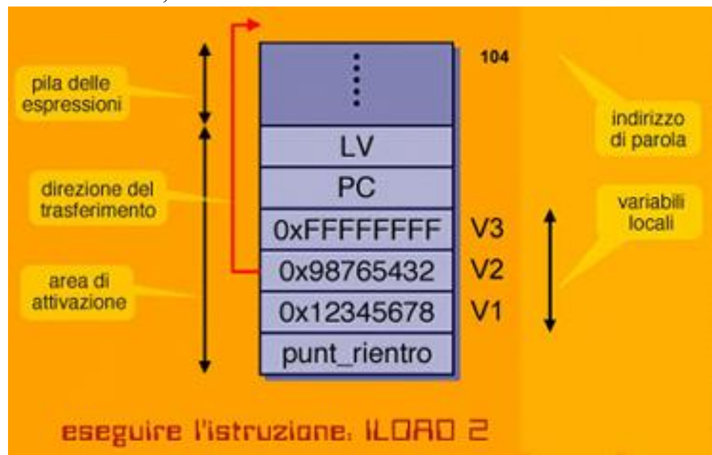
Istruzione **ISTORE** numvar ha la funzione di POP dell'elemento sulla pila nella variabile locale indicata dallo spiazzamento“numvar”.

Le istruzioni citate operano sia sulla pila per il calcolo delle espressioni sia sulle variabili locali. In ogni caso esse operano su parole da 32 bit.



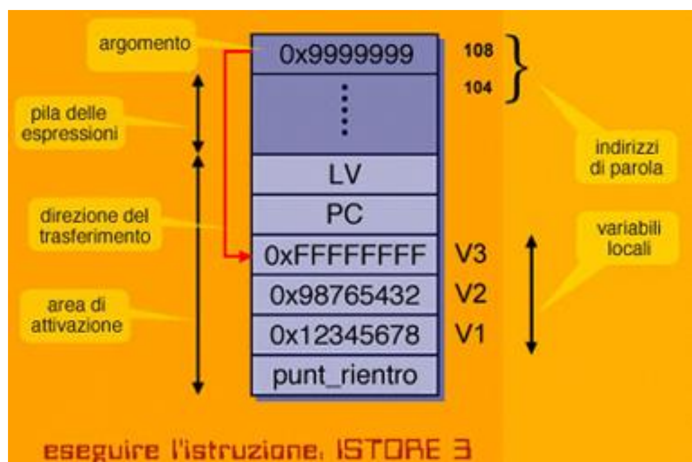
- Simulazione **ILOAD**

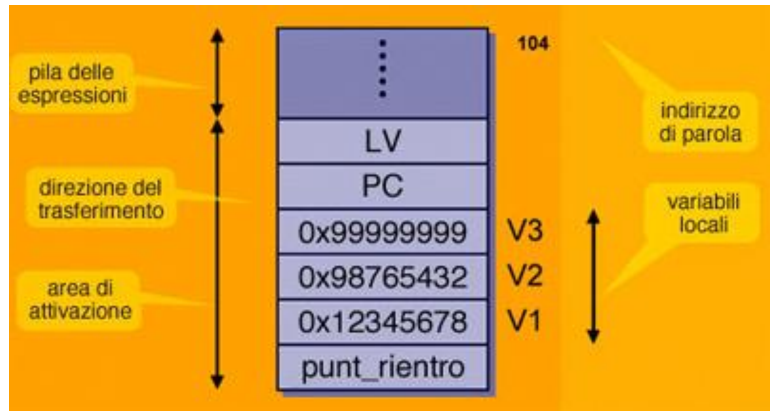
Mostriamo lo stato della pila prima di eseguire l'istruzione ILOAD 2, e lo stato successivo all'istruzione ILOAD 2.
Si nota che la variabile locale indicata con lo spiazzamento 2, cioè la variabile v_2 il cui valore 0x98765432 è stata scritta sulla cima della pila.



- Simulazione **ISTORE**

Mostriamo lo stato della pila prima di eseguire l'istruzione ISTORE 3, e lo stato successivo della pila dopo l'esecuzione dell'istruzione.
Si nota che l'elemento sulla cima della pila è stato tolto e sovrascritto al valore precedente alla variabile indicata dallo spiazzamento 3.





Istruzioni di caricamento-memorizzazione

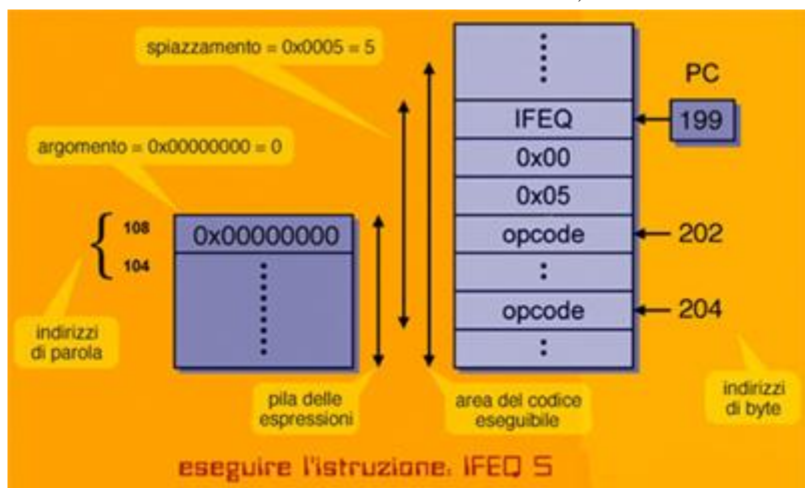
Classi di "istruzioni di salto" (di controllo del flusso di esecuzione del programma). Lo schema qui riportato, sintetizza le istruzioni facenti parte di questa classe, e il loro funzionamento.

Hex	Mnemonico	Funzionamento: operazione e risultato
0x99	IFEQ spiazzamento	Salta se l'elemento in cima alla pila è = 0
0x9B	IFLT spiazzamento	Salta se l'elemento in cima alla pila è < 0
0x9F	IFCMPEQ spiazzamento	Salta se i 2 elementi in cima alla pila sono =
0xA7	GOTO spiazzamento	Salta in modo incondizionato
0xAC	IRETURN	Rientra da sottoprogramma
0xB6	INVOKEVIRTUAL spiazzamento	Salta a sottoprogramma

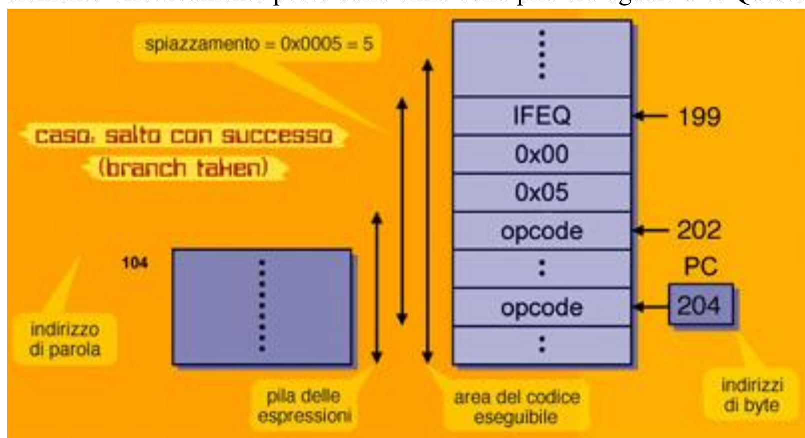
Le istruzioni **IF** e **GOTO**, operano sulla pila delle espressioni, con parole da 32 bit e sul contatore di programma PC.

- Simulazione **IFEQ** (estrae una parola sulla cima dello stack ed effettua una diramazione se ha valore 0).

Mostriamo lo stato della memoria prima dell'istruzione IFEQ con spiazzamento 5. Sulla cima pila è presente una parola di memoria che vale 0, rappresentata in notazione esadecimale.
Il contatore di programma correntemente punta all'istruzione IFEQ, la quale è contenuta nell'area del codice eseguibile. Questa istruzione è formata da 3 Byte (1 Byte che indica l'istruzione IFEQ stessa, e poi vi è una coppia di Byte che indicano lo spiazzamento del salto.

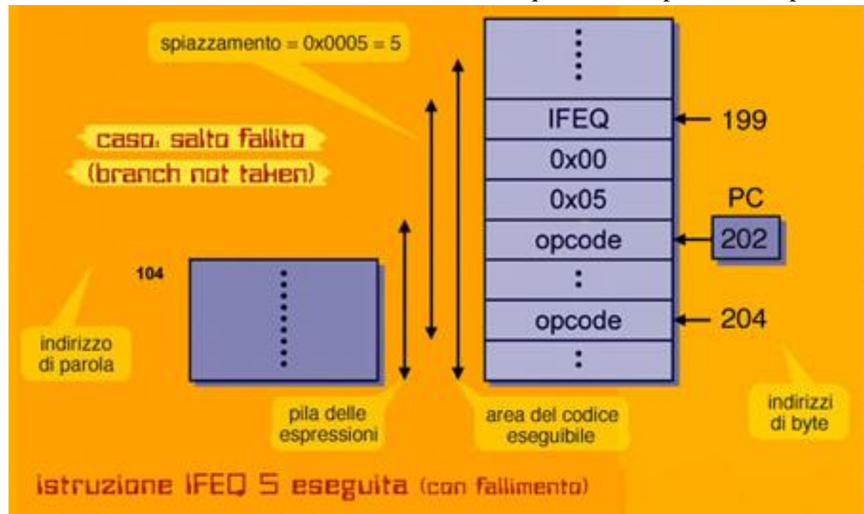


L'operazione mostrata in figura indica che l'elemento effettivamente posto sulla cima della pila era uguale a 0. Questo elemento è stato tolto e il salto viene eseguito, cioè il contatore di programma viene spostato in avanti secondo quanto indicato dallo spiazzamento 5, cioè di cinque posizioni. Qui si troverà la prossima istruzione da eseguire.
Si dice anche: il salto è stato eseguito con successo.



Altro caso di esecuzione

Se sulla cima della pila avessimo trovato un elemento diverso da 0 come valore, in questo caso può essere pure il numero 1, il salto fallisce, si dice pure che l'istruzione IFEQ 5 viene eseguita, ma avremo un fallimento, l'elemento sulla cima della pila viene comunque eliminato, mentre non ha luogo il salto, quindi si passa all'esecuzione dell'istruzione immediatamente successiva a IFEQ stessa.



- Istruzione **IFLT** e simulazione

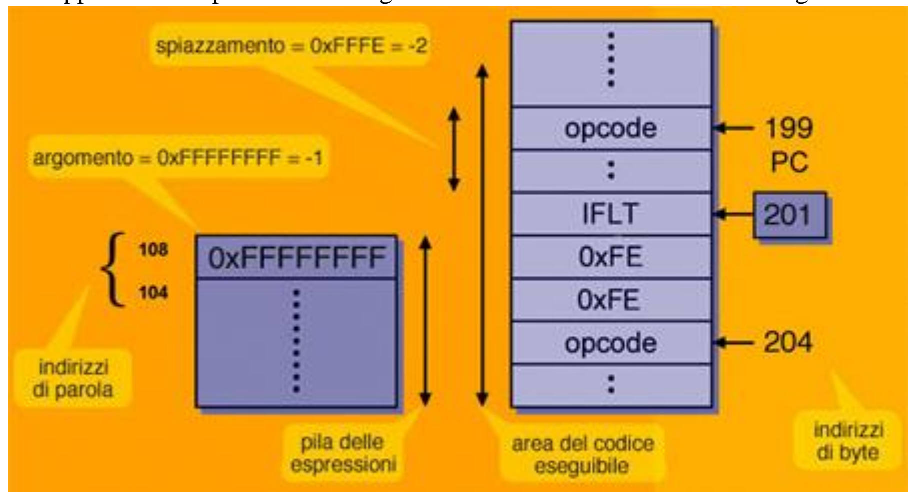
Questa istruzione esegue un salto secondo:

- lo spiazzamento indicato
- l'elemento sulla cima della pila sia < 0 oppure no

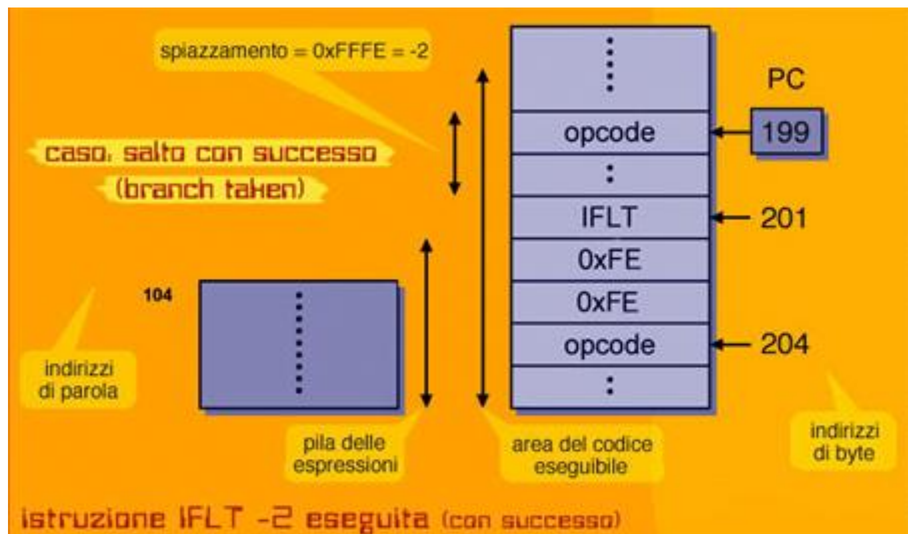
Il funzionamento di IFLT è analogo all'istruzione IFEQ.

Esempio

Eseguire l'istruzione IFLT -2, dove -2 rappresenta lo spiazzamento negativo. Vuol dire che il salto viene eseguito sulla condizione verificata all'indietro se è un'istruzione precedente. Lo schema mostra lo strato della memoria a monte dell'esecuzione dell'istruzione. Sulla cima dello stack è presente un elemento, ed esso se interpretato in completamento a due indica un numero negativo (-1).



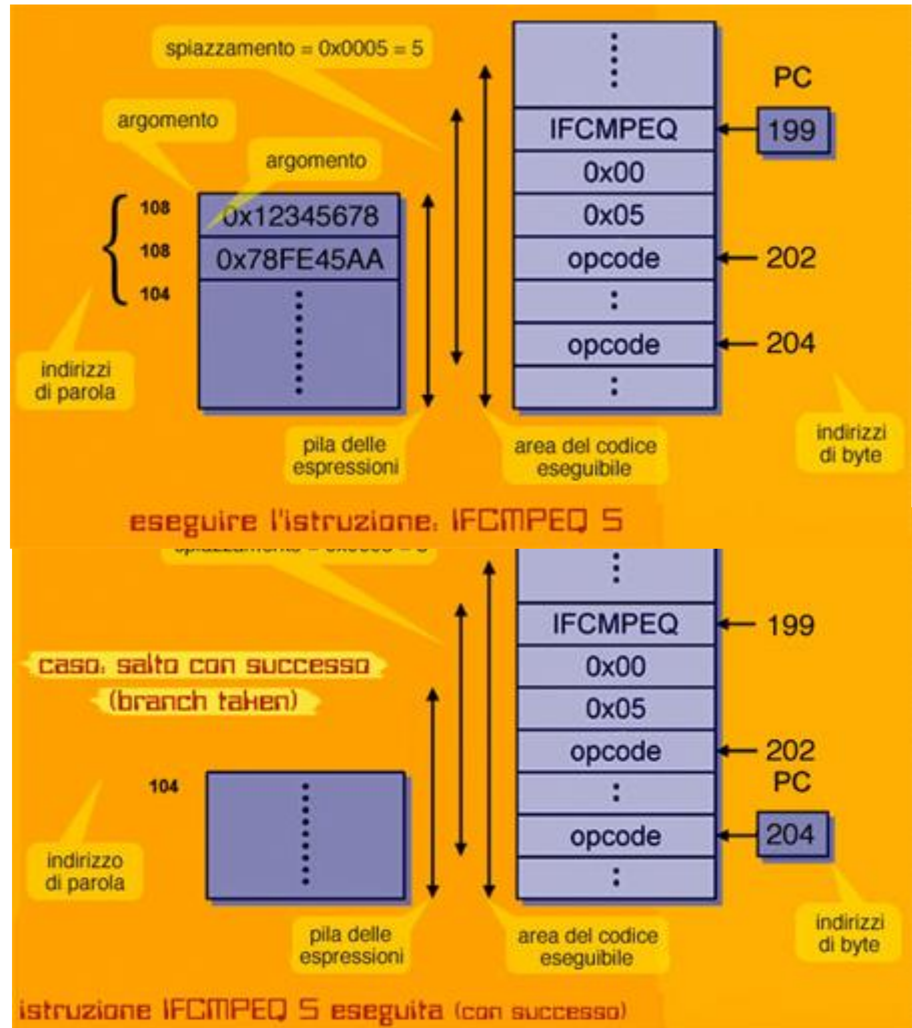
Il successivo schema, mostra che l'istruzione IFLT -2 è stata eseguita con successo. Il PC viene opportunamente incrementato di due unità in modo da puntare ad una istruzione precedente (salto all'indietro).



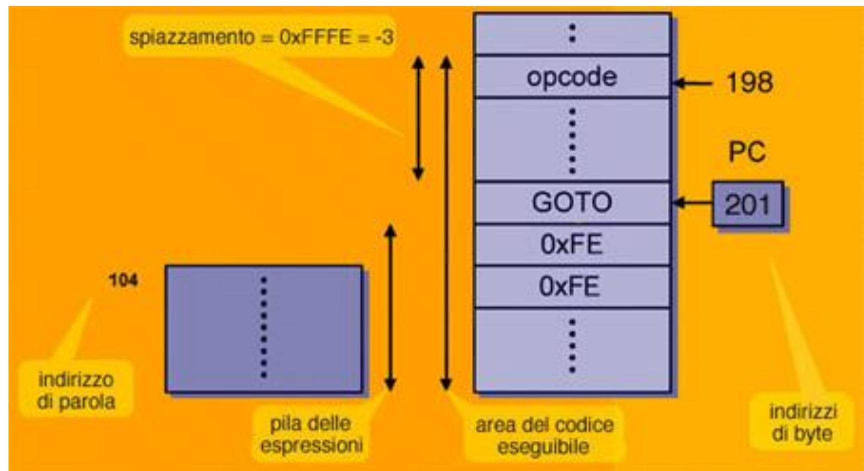
Un caso di fallimento del salto
Qualora sulla cima dello stack è presente un numero > 0, l'istruzione IFLT -2 fallisce, l'elemento sulla cima della pila dello stack viene eliminato, ma il salto non ha luogo, si passa dunque all'esecuzione dell'istruzione successiva.

- Simulazione **IFCMPEQ**

Tale istruzione confronta i due elementi della cima della pila. Il salto viene eseguito con successo, questi due elementi sono coincidenti, altrimenti fallisce.
Lo schema fa vedere che sulla cima dello stack sono presenti i due elementi coincidenti con uno spiazzamento di salto 5.
A valle dell'esecuzione dell'istruzione i due elementi in cima alla pila sono stati tolti e il salto eseguito con successo.



Altro caso
Lo schema mostra un esecuzione dell'istruzione IFCMPEQ 5 che si concluderà con un fallimento. Notiamo che i due elementi in cima alla pila sono diversi. A valle dell'esecuzione dell'istruzione sono stati comunque tolti, ma il salto è fallita (poiché gli elementi sono diversi). Pertanto si passa all'esecuzione dell'istruzione successiva.



o Simulazione di **GO TO** (Salto incondizionato)
Mostriamo la situazione della memoria prima dell'esecuzione dell'istruzione GO TO -3. Lo spiazzamento è negativo, il salto avviene in modo incondizionato all'indietro. Dopo l'esecuzione del salto, il PC è stato spostato all'indietro di tre unità, portandolo a puntare all'istruzione precedente a questa. Il salto avviene con successo.
L'istruzione di manipolazione della pila forma con la classe (eterogenea), che serve per eseguire varie operazioni di gestione dello stack. Per la maggior parte si tratta di istruzioni che operano sulla pila e sull'area delle costanti, con parole da 32 bit.

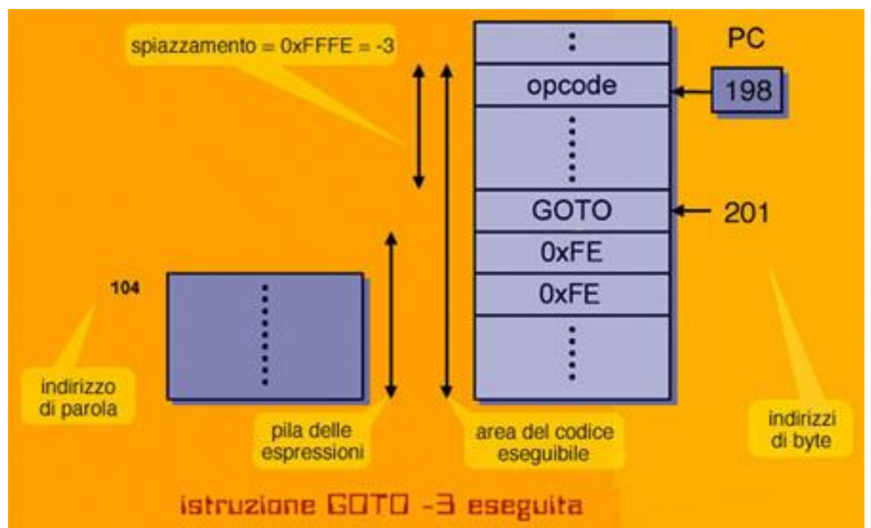
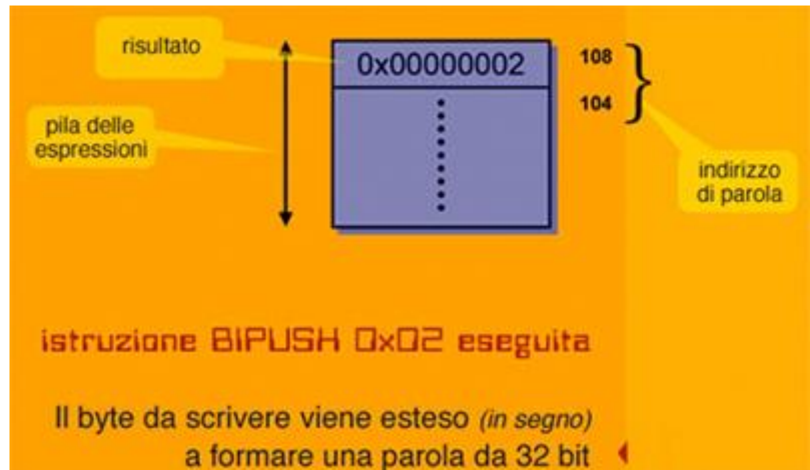


Tabella delle istruzioni

Hex	Mnemonico	Funzionamento: operazione e risultato
0x10	BIPUSH byte	Push del valore "byte" (8 bit) sulla cima della pila
0x13	LDWC numcost	Push della costante "numcost" sulla pila
0x57	POP	Elimina la parola (32 bit) in cima alla pila
0x59	DUP	Duplica la parola (32 bit) in cima alla pila
0x5F	SWAP	Scambia le 2 parole (32 bit) in cima alla pila

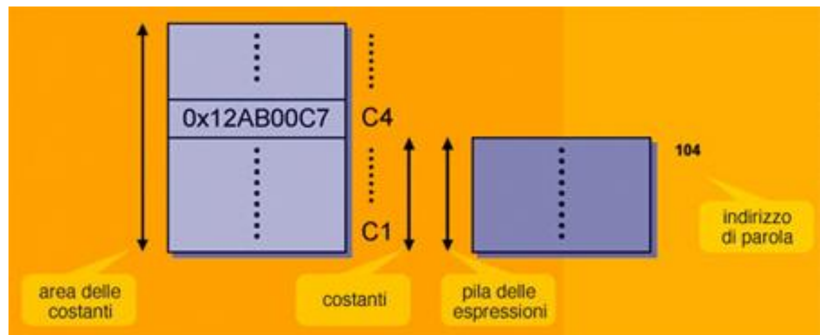
- Simulazione di **BIPUSH**

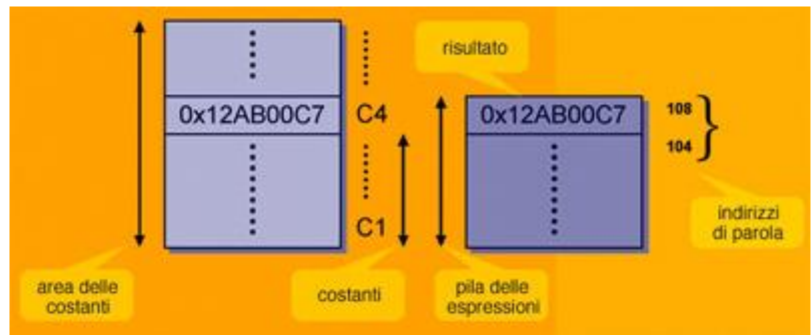
Questa istruzione ha un argomento da 1 Byte, che viene esteso a formare una parola da 32 bit (vedere l'estensione di segno in complemento a 2).
Mostriamo la situazione della memoria prima dell'esecuzione dell'istruzione BIPUSH con argomento indicato in esadecimale 0x02, esteso a 32 bit e messo sulla cima dello stack, e successivamente dopo l'esecuzione dell'istruzione viene messo sulla cima.

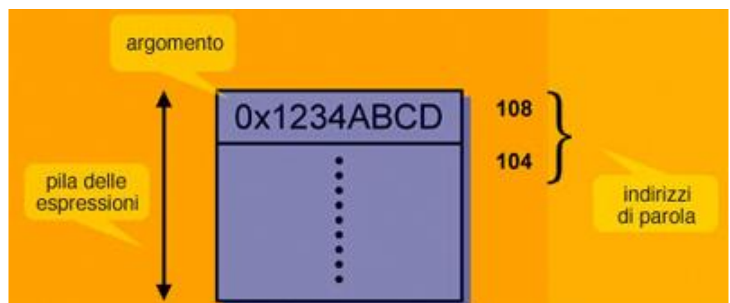


- Simulazione di **LDWC**

Per eseguire l'istruzione LDWC 4, dove 4 è uno spiazzamento che all'interno dell'area delle costanti, indica una certa costante a 32 bit, il cui valore in notazione esadecimale è visualizzato in figura, nella quale deve essere svolta l'istruzione LDWC 4.
Subito dopo l'esecuzione dell'istruzione LDWC 4, la costante indicata con lo spiazzamento 4 è stata scritta sulla cima dello stack.





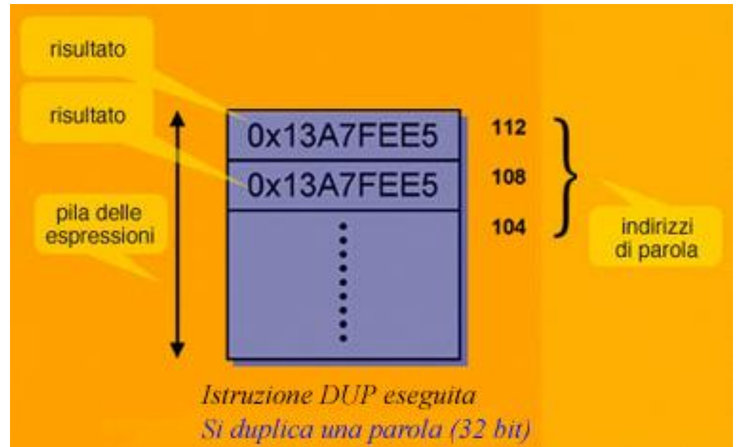


- Simulazione di **POP**

L'istruzione consente di togliere una parola sulla cima dello stack (da 32 bit). Mostriamo la situazione della pila prima dell'esecuzione di POP e dopo all'esecuzione di POP. L'elemento in cima alla pila è stato eliminato.

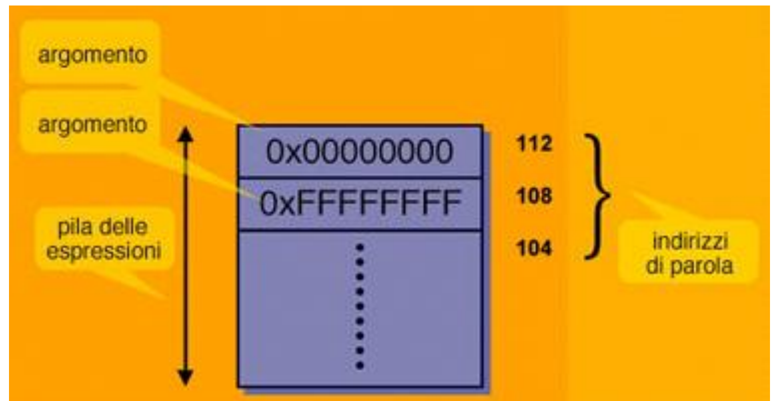
- Simulazione di **DUP**

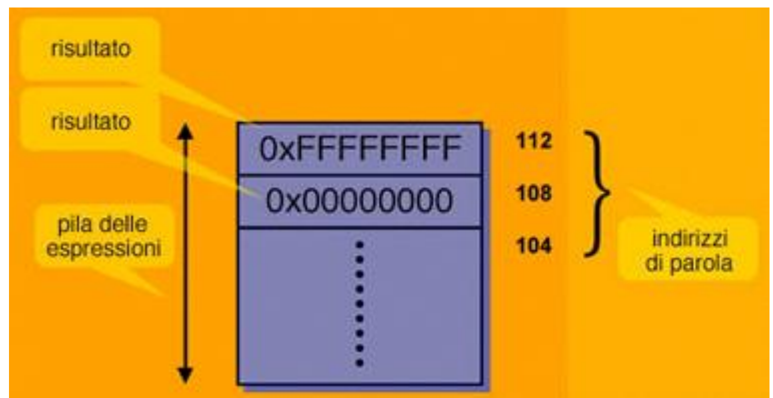
L'istruzione duplica l'elemento che si trova in cima sulla pila. La figura visualizza l'esecuzione di DUP.



- Simulazione di **SWAP**

L'istruzione consente di scambiare la posizione degli elementi in cima allo stack.
In figura è visualizzata la situazione prima e dopo l'esecuzione dell'istruzione.
La classe delle istruzioni di controllo è piuttosto disomogenea, legate al modello di processore.





Analizziamo due istruzioni di controllo come esempio, riportati nella seguente tabella:

Hex	Mnemonico	Funzionamento: operazione e risultato
0x00	NOP	Nessuna operazione
0xC4	WIDE	L'istruzione seguente ha un argomento di 16 bit

L’istruzione **WIDE** indica una modifica nel modo di interpretare l’istruzione successiva, la quale deve avere un argomento da 16 bit.

Altri possibili istruzioni di controllo:

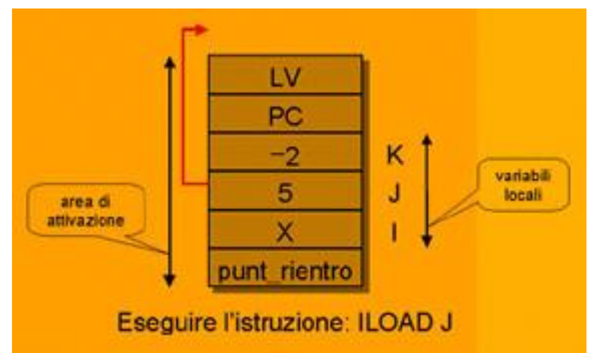
- **HALT** per arrestare il processore, e le istruzioni per il controllo delle interruzioni. Il processore IJVM non dispone di istruzioni macchina I/O.

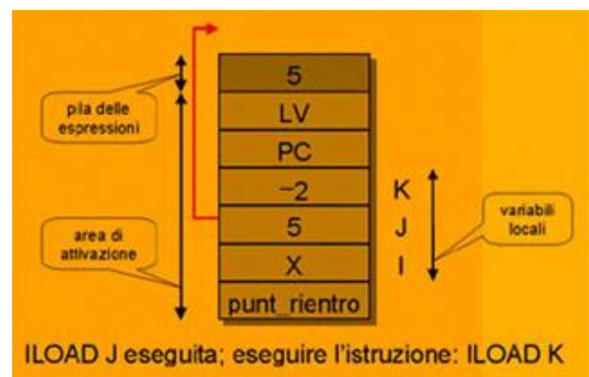
Per comunicare con le unità funzionali di I/O, il processore IJVM si serve di una libreria di funzioni Java predefinite, che si usano come dei normali sottoprogrammi.
Altri tipi di processore, comunque dispongono di istruzioni I/O, facenti parte del linguaggio macchina.

Simulazione dell’esecuzione del programma (fa comprendere la corrispondenza tra il codice java e la traduzione in linguaggio macchina).

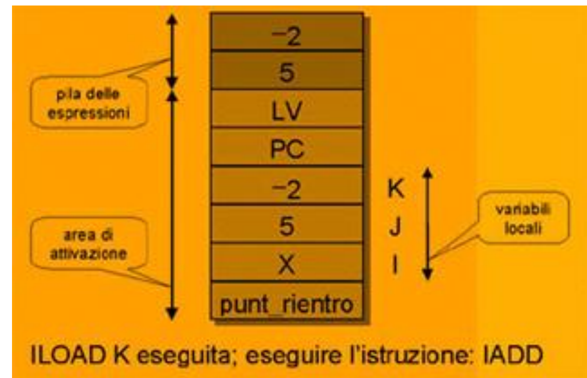
Abbiamo J = 5, K = -2, pertanto:
l’assegnamento I = J + K vale 5 – 2 → 3.
La pila delle espressioni è vuota; la figura visualizza lo stato di partenza dell’area di attivazione.

Adesso eseguiamo l’istruzione ILOAD J.
Mostriamo lo stato della memoria prima dell’esecuzione e dopo l’esecuzione della stessa. Notiamo che il valore della variabile locale J è stato scritto sulla cima dello stack dell’espressione.

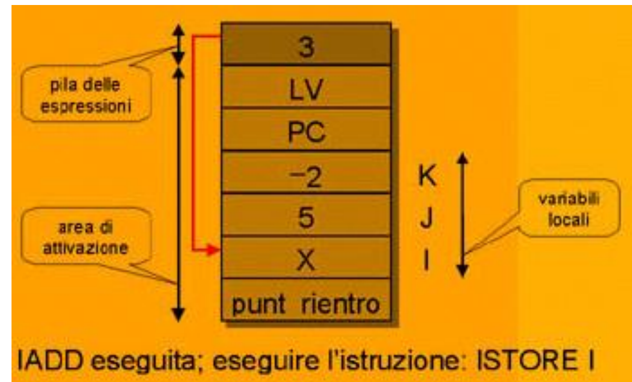




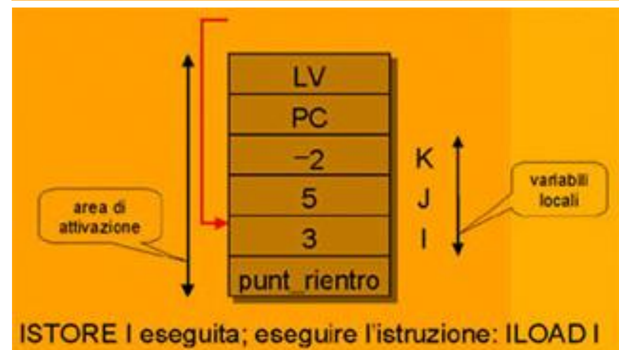
Eseguiamo l'istruzione ILOAD K.
Mostriamo lo stato di esecuzione di ILOAD K.
Sulla cima dello stack è stato scritto il valore della variabile locale K: -2.



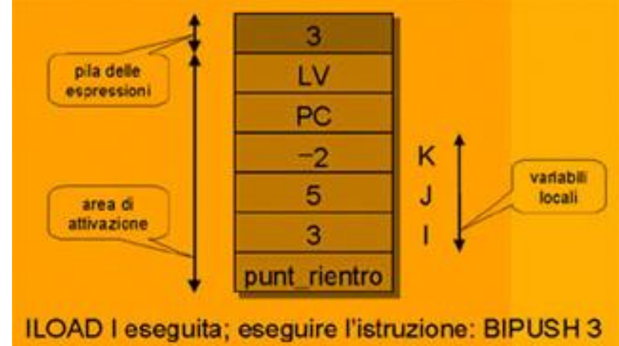
Eseguiamo l'istruzione IADD.
Sulla cima della pila i valori di J e K sono sommati, è in loro luogo viene scritto il risultato.



Eseguiamo l'istruzione ISTORE I.
Sulla cima della pila il valore 3 viene tolto e viene scritto nella cella della variabile locale I. notiamo che la pila dell'espressione è vuota.

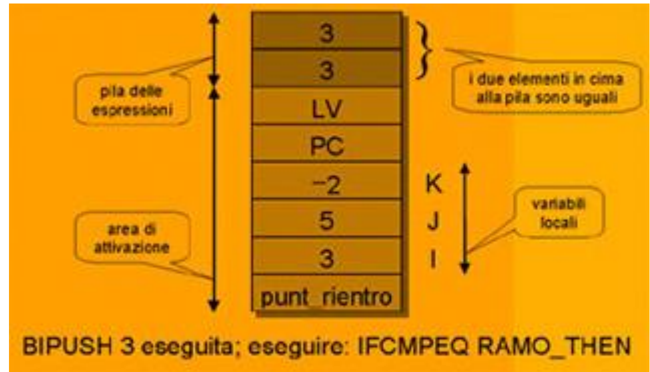


Eseguiamo l'istruzione ILOAD I.
Sulla cima dello stack viene scritto il valore 3.

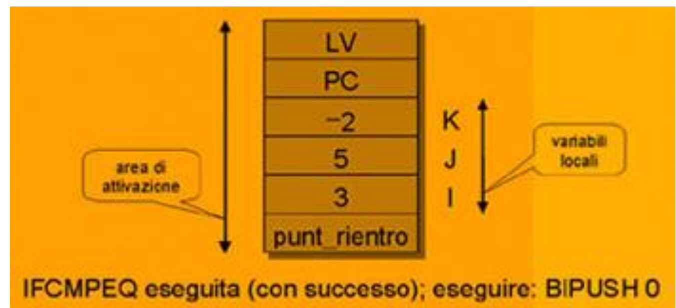


Eseguiamo l'istruzione BIPUSH 3.

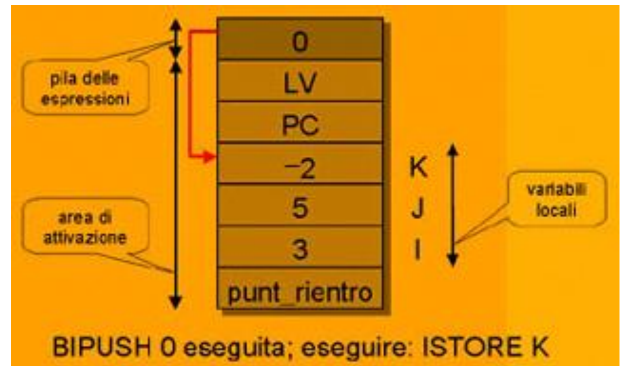
Eseguendo l'istruzione sulla cima dello stack viene scritto il valore 3.



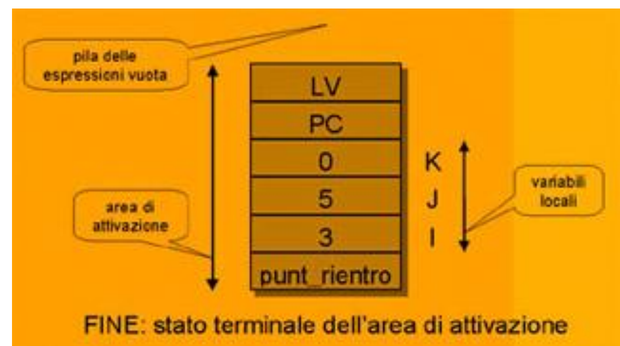
Eseguire l'istruzione IFCMPEQ RAMO_THEN (salto alla destinazione indicata dall'etichetta RAMO_THEN).
Eseguendo il salto con successo, il valore 3 (duplicato) viene eliminato, mentre il punto di esecuzione del programma viene trasferito all'etichetta RAMO_THEN. In questa posizione può rieseguire l'istruzione BIPUSH 0.



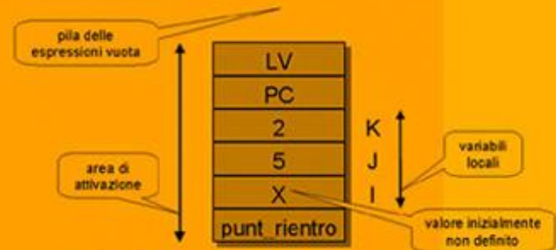
Eseguiamo l'istruzione BIPUSH 0.
Sulla cima della pila viene scritto il valore 0.

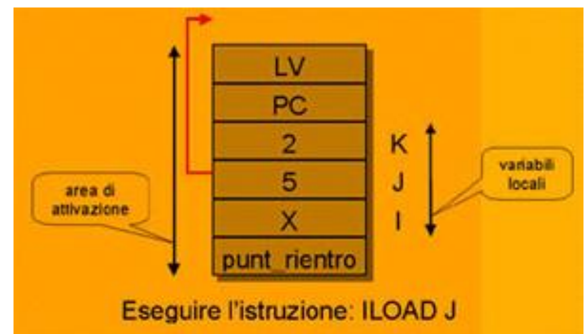


Eseguiamo l'istruzione K.
La variabile locale K assume valore 0. Con questa operazione si raggiunge lo stato terminale finale dell'area di attivazione.

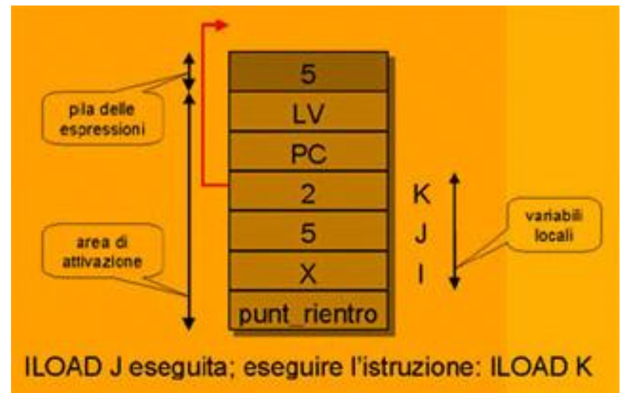


Altro caso
J = 5, K = 2, pertanto:
l'assegnamento I = J + K vale 5 + 2 → 7.

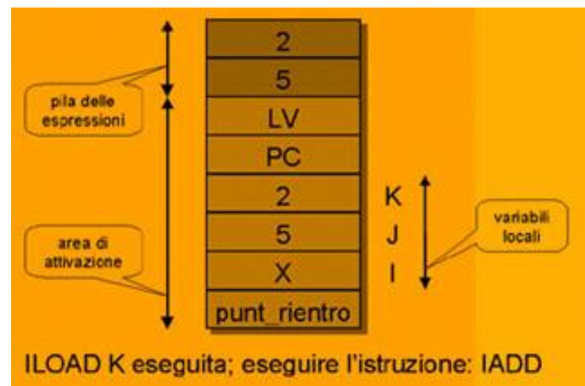




Mostriamo lo stato iniziale dell'area di attivazione in cui si vedono i valori delle variabili già definiti.
Eseguiamo l'istruzione ILOAD J.
Essa rappresenta la 1ª istruzione del programma; il valore della variabile J viene copiato sulla cima dello stack, quindi ho 5.

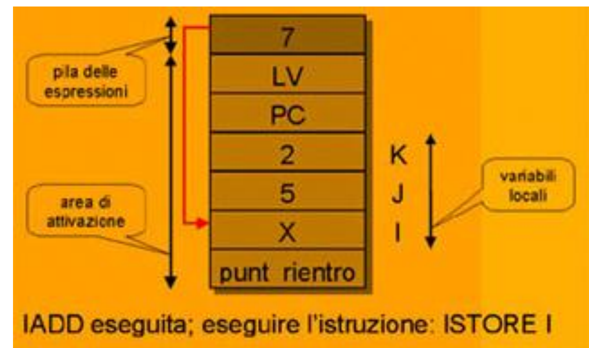


Eseguiamo l'istruzione K, il valore 2 viene scritto sulla cima della pila.

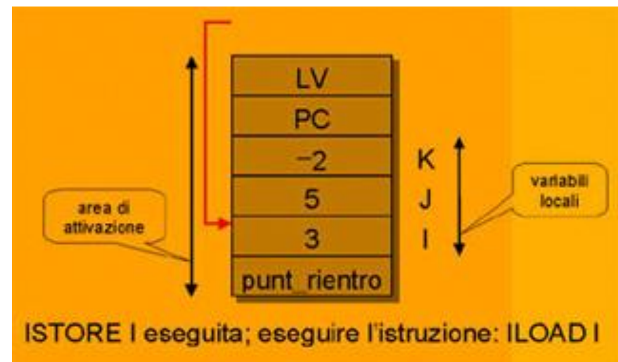


Eseguiamo l'istruzione IADD
Sulla cima dello stack scriviamo 7, mentre gli elementi della somma sono eliminati.

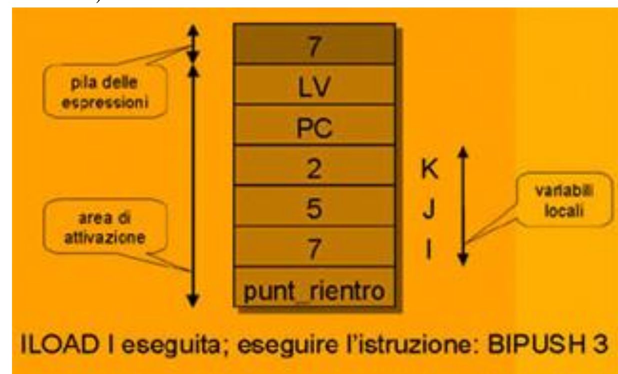
Eseguiamo l'istruzione ISTORE I.
Il valore in cima verrà scritto nella cella della variabile locale I.



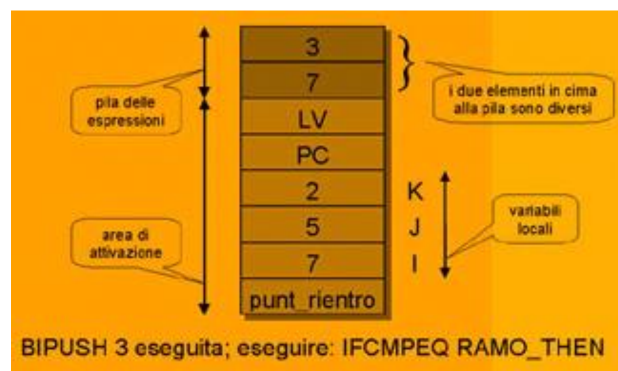
Eseguiamo l'istruzione ILOAD I.
Il valore della variabile locale I viene scritto sulla cima dello stack.



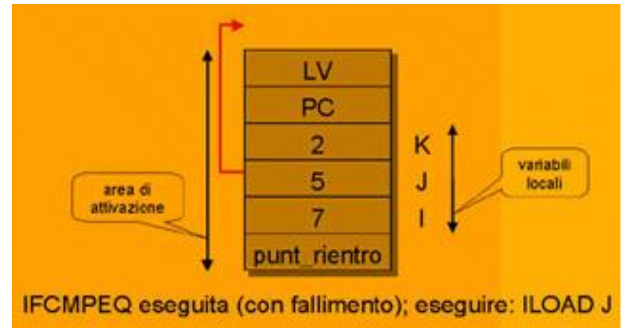
Eseguiamo l'istruzione BIPUSH 3.
Viene scritta la costante 3 sulla cima dello stack (estendendola a 32 bit).



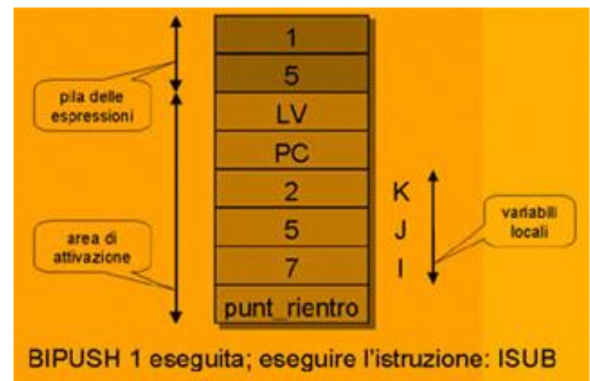
Eseguiamo l'istruzione IFCMPEQ RAMO_THEN; questa istruzione di salto condizionato fallisce perché i due elementi vicino alla pila sono diversi, vengono rimossi e confrontati, ma il salto non ha luogo.



Eseguiamo l'istruzione ILOAD J.
Viene scritto sulla cima dello stack il valore di J, cioe 5.

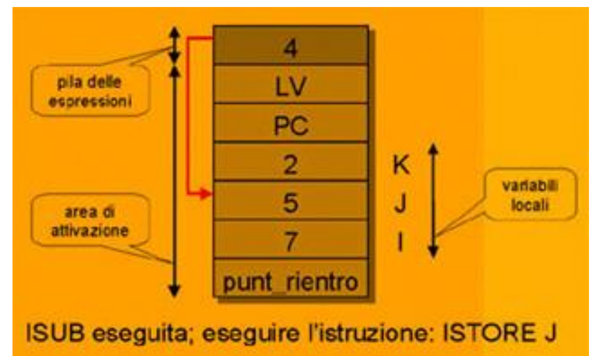


Eseguiamo l'istruzione BIPUSH 1.
La costante 1 viene scritta sulla cima della pila.



Eseguiamo l'istruzione ISUB.
Viene scritto sulla cima dello stack il valore 4.

Eseguiamo l'istruzione ISTORE J.
Il valore in cima alla pila viene scritto nella cella della variabile J.



Eseguiamo l'istruzione di salto incondizionato GO TO FINE (dove FINE indica l'etichetta di destinazione del salto). Eseguendo l'istruzione FINE: stato terminale dell'area di attivazione.

IMPLEMENTAZIONE DI ESEMPIO

Dopo aver specificato la microarchitettura e la macroarchitettura, rimane il problema dell'implementazione. In altri termini, 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.

Microistruzioni e notazione

In teoria potremmo descrivere la memoria di controllo in binario, 36 bit per parola. Ma è vantaggioso introdurre una notazione che esprime l'essenza dei problemi da trattare. 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. 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. L'organizzazione del controllo dipende dalla capacità di specificare gli indirizzi in modo efficiente. La nostra notazione specifica su una singola linea tutte le attività che si svolgono durante un unico ciclo di clock. È necessario poter analizzare i cicli in modo da comprendere lo svolgimento e verificare le operazioni. 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 un ciclo di clock. Supponiamo che in ciclo di clock vogliamo incrementare il valore di SP, oltre a dare inizio a un'operazione di lettura, vogliamo anche che l'istruzione successiva sia quella che risiede nella locazione 122 della memoria di controllo.
Prima di scrivere la sintassi, è opportuno dire che:

Mic-1: Micro Assembly Language

Prima di studiare come scrivere microprogrammi per codificare le istruzioni ISA **IJVM** in termini di microistruzioni, introduciamo una notazione semplificativa denominata **MAL** (**M**icro **A**ssembly **L**anguage) per indicare cosa ogni microistruzione deve fare.
Scrivere microprogrammi usando un numero binario di 36 bit per ogni microistruzione risulterebbe infatti piuttosto difficile e soprattutto completamente incomprensibile!
La traduzione da MAL a codici binari 36 bit (**microassembling di MAL**) è un compito noioso ma molto semplice per un calcolatore.
Nella nostra notazione MAL tutto ciò che viene eseguito in un singolo ciclo deve essere scritto in un'unica riga.
Ritornando all'esempio, se in un ciclo volessimo leggere SP sul bus B, eseguirne l'incremento di 1 tramite la ALU, memorizzare il risultato in SP, avviare una lettura da memoria e indicare 122 come prossimo indirizzo di microistruzione, potremmo scrivere:
ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
dove WSP significa "scrivere il registro SP".
Questo rappresenta una notazione completa, ma di difficile lettura.
È meglio utilizzare semplicemente:
SP = SP + 1; rd
Chiamiamo il nostro linguaggio ad alto livello MAL. È definito per riflettere le caratteristiche della micro architettura. Durante ogni ciclo di clock è possibile scrivere qualsiasi registro, anche se in genere ne viene scritto uno. Soltanto uno può essere collegato al lato B dell'ALU, mentre sul lato A (canale sinistro) 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 MAR possiamo per esempio dire:
MDR = SP
Copiare un registro sull'altro (es. SP = MDR) equivale semplicemente a utilizzare 0 sul canale sinistro dell'ALU.
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 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 tabella accanto, in cui SOURCE può essere MDR, PC, MBR, MBRU, SP, LV, CPP, TOS oppure (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 possono essere illegali. Per esempio l'assegnamento:

MDR = SP + MDR
sembra ragionevole, ma non può essere eseguito in un unico ciclo sul percorso dati della tabella. 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. 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 1 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 alla memoria 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 procedure 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:
MDR = SP; rd
MDR = H // entrambe assegnano un valore a MDR al termine del secondo ciclo, sequenza illegale
La prima microistruzione assegna alla fine della seconda microistruzione un valore della memoria 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 sopra di essa.

Direttive di salto
Per facilitare il lavoro del programmatore normalmente il microassemblatore assegna un indirizzo a ciascuna microistruzione e completa il campo NEXT_ADDRESS in modo che le microistruzioni scritte sulle linee consecutive vengano eseguite in sequenza.
Tuttavia in alcuni casi il microprogrammatore 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.

In altri termini: Per convenzione ogni riga di MAL che non contiene un salto esplicito, viene implicitamente tradotta impostando NEXT_ADDRESS all'indirizzo della microistruzione della riga seguente.
Per eseguire un salto incondizionato è sufficiente indicare al termine della riga: goto label forzando in questo modo il valore di NEXT_ADDRESS.

Implementazione di IJVM con Mic-1
Adesso è possibile costruire ordinatamente il microprogramma eseguito su Mic-1 che interpreta IJVM. Il microprogramma è costituito da 112 istruzioni. Per ciascuna microistruzione ci sono tre colonne:
- *l'etichetta simbolica*
- *il microcodice effettivo*
- *un commento*

Da rilevare che microistruzioni consecutive non sono necessariamente localizzate in indirizzi consecutivi all'interno della memoria di controllo.

L'interprete IJVM per Mic-1 realizzato dal microprogramma MAL prevede come tutti gli interpreti un ciclo principale infinito che legge, decodifica ed esegue le istruzioni.
Il **ciclo principale** è costituito dalla sola riga **Main1** che:
- incrementa il Program Counter (PC = PC +1)
- inizia il fetch del prossimo byte (op-code successivo o operando istruzione corrente).
- salta all'indirizzo dell'istruzione presente in MBR. Si assume che quando ci si trova in Main1 l'op-code dell'istruzione sia già stato caricato in MBR; sarà dunque compito del microprogramma di ogni istruzione IJVM provvedere a ciò.

Etichetta	Microistruzione	Commenti
Main1	PC=PC+1; fetch; goto (MBR)	MBR contiene già l'op-code dell'istruzione corrente

Si assume inoltre che il registro TOS rimanga sempre aggiornato al contenuto della parola in cima allo stack. Ciò consentirà di risparmiare preziosi accessi alla memoria. Anche in questo caso ogni microprogramma di istruzione IJVC deve rispettare questa assunzione. Gli indirizzi delle microistruzioni all'interno del control store non vengono qui riportati; al loro posto vengono utilizzate delle più pratiche etichette:
L'etichetta di ogni microistruzione è composta dal nome dell'istruzione IJVM cui il microprogramma si riferisce ed è affiancata da un intero crescente all'interno del microprogramma di quell'istruzione. Le etichette vengono (automaticamente) tradotte in indirizzi dal microassemblatore ...
Nel seguito sono riportati i frammenti di microprogramma dell'Interprete IJVM relativi ad alcune istruzioni.

Etichetta	Microistruzione	Commenti
Nop1	goto Main1	Nop non esegue nessuna istruzione; è sufficiente saltare al ciclo principale
iadd1 Iadd2 iadd3	MAR=SP=SP-1; rd H =TOS MDR=TOS=MDR+H; wr; goto Main1	La prima delle 2 parole è già in TOS; avvia la lettura della seconda che si trova a SP -1 (*legge la 2ª parola in cima allo stack*) Copia in H la prima parola, la seconda sarà in MDR al termine di questo ciclo (*H = cima dello stack*) MDR viene aggiornato con la somma e si avvia la scrittura su SP -1; il multi - assegnamento consente di mantenere aggiornato anche TOS (*Somma le due parole in cima allo stack; scrive in cima allo stack*)
isub1 isub2 isub3	MAR=SP=SP-1; rd H =TOS MDR=TOS=MDR-H; wr; goto Main1	Legge la 2ª parola in cima allo stack Copia in H la prima parola, la seconda sarà in MDR al termine di questo ciclo (*H = cima dello stack*) MDR viene aggiornato con la sottrazione e si avvia la scrittura su SP - 1; il multiassegnamento consente di mantenere aggiornato anche TOS (*Sottrae le due parole in cima allo stack; scrive in cima allo stack*)
iand1 iand2 iand3	MAR=SP=SP-1; rd H =TOS MDR=TOS=MDR AND H; wr; goto Main1	Legge la 2ª parola in cima allo stack Copia in H la prima parola, la seconda sarà in MDR al termine di questo ciclo (*H = cima dello stack*) MDR viene aggiornato con l'AND e si avvia la scrittura su SP -1; il multiassegnamento consente di mantenere aggiornato anche TOS (*Esegue l'AND in cima allo stack; scrive nella nuova cima dello stack*)
ior1 ior2 ior3	MAR=SP=SP-1; rd H =TOS MDR=TOS=MDR OR H; wr; goto Main1	Legge la 2ª parola in cima allo stack Copia in H la prima parola, la seconda sarà in MDR al termine di questo ciclo (*H = cima dello stack*) MDR viene aggiornato con l'OR e si avvia la scrittura su SP -1; Il multiassegnamento consente di mantenere aggiornato anche TOS (*Esegue l'OR in cima allo stack; scrive nella nuova cima dello stack*)

dup1 dup2	MAR=SP=SP+1 MDR =TOS; wr; goto Main1	Incrementa SP e prepara MAR per la scrittura Prepara MDR per la scrittura SP+1; avvia la scrittura e salta a Main1
pop1 pop2 pop3	MAR = SP = SP - 1; rd TOS = MDR; goto Main1	Legge la parola a SP-1 per il semplice motivo di dover tenere aggiornato TOS Attende che il nuovo sia letto dalla memoria (*Deve attendere un ciclo per la lettura senza fare nulla*) Copia la nuova parola in TOS (*Può finalmente assegnare TOS e tornare al Main1*)
bipush1	SP=MAR=SP+1	MBR=Byte da inserire nello stack (*Lo stack deve crescere inserendo il nuovo byte*)

bipush2	PC=PC+1; fetch	Incrementa PC, preleva il successivo codice operativo (*Il byte operando per questa istruzione è già stato pre-caricato da Main1; devo comunque eseguire il fetch per caricare in MBR l'op-code successivo*)
bipush3	MDR=TOS=MBR; wr; goto Main1	Estende il segno della costante e la inserisce nello stack (*MBR viene esteso con segno a lunghezza parola che viene scritta a SP*)
iload1	H=LV	MBR contiene l'indice; copia LV in H (*LV è la base delle variabili locali*)
iload2	MAR=MBRU + H; rd	MAR = indirizzo della variabile locale da inserire nello stack (*L'indirizzo a cui prelevare la variabile è LV + varnum (il cui valore è già in MBR). Varnum deve essere considerato unsigned (pertanto utilizzo MBRU*)
iload3	MAR=SP=SP +1	SP punta alla nuova cima dello stack; prepara la scrittura (*La variabile sarà salvata a SP +1*)
iload4	PC=PC +1; fetch; wr	Incrementa PC; ottiene il nuovo codice operativo; scrive la cima dello stack (*Esegue il fetch del prossimo op-code; avvia la scrittura nello stack in quanto MDR ora è disponibile con il valore della variabile*)
iload5	TOS = MDR; goto Main1	Aggiorna TOS e torna al Main1
goto1	OPC=PC-1	Salva l'indirizzo del codice operativo
goto2	PC=PC+1; fetch	MBR contiene il 1° Byte dell'indice; preleva il secondo
goto3	H=MBR<<8	Trasla in H il 1° Byte con segno
goto4	H=MBRU OR H	H = spiazzamento a 16 bit della diramazione
goto5	PC=OPC+H; fetch	Aggiunge lo spiazzamento a OPC
goto6	gotoMain1	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	Inserisci in TOS la nuova cima dello stack
iflt4	N=OPC; if (Z) 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
ifeq 2	OPC=TOS	Salva temporaneamente TOS in OPC
ifeq3	TOS=MDR	Inserisci in TOS la nuova cima dello stack
ifeq4	N=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_icmpe	TOS=MDR	Inserisci in TOS la nuova cima dello stack
	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

q5 if_icmpeq6		
Invokevirtual1	PC=PC=1; fetch	MBR = Byte 1 dell'indice; incrementa PC; ottiene il secondo Byte

I registri CPP, LV e SP sono utilizzabili 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 verrà 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; ciò nonostante 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.
- Il registro OPC è un registro temporaneo e non ha un utilizzo predefinito. È usato per es. 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 per Mic-1 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 essere stato memorizzato all'interno di MBR.
- Ciò implica che prima di iniziare ogni iterazione dobbiamo assicurarci che PC sia già 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 esegue 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 tabella dell'insieme d'istruzioni IJVM vediamo quindi 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 qualche parte un documento 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 micro diramazioni esplicite (cioè microistruzioni che effettuano salti) per saltare da una cella ad un'altra della memoria, il tempo sprecato sarebbe significativo.

  Per vedere come funziona l'interprete, assumiamo che MBR contenga il 0x60, cioè il codice operativo corrispondente a IADD. 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 al 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 estraendo in questa microistruzione non ha 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 niente, essa effettua semplicemente un salto all'inizio del ciclo principale, dove viene ripetuta la sequenza utilizzando però il nuovo codice operativo memorizzato in MBR.