

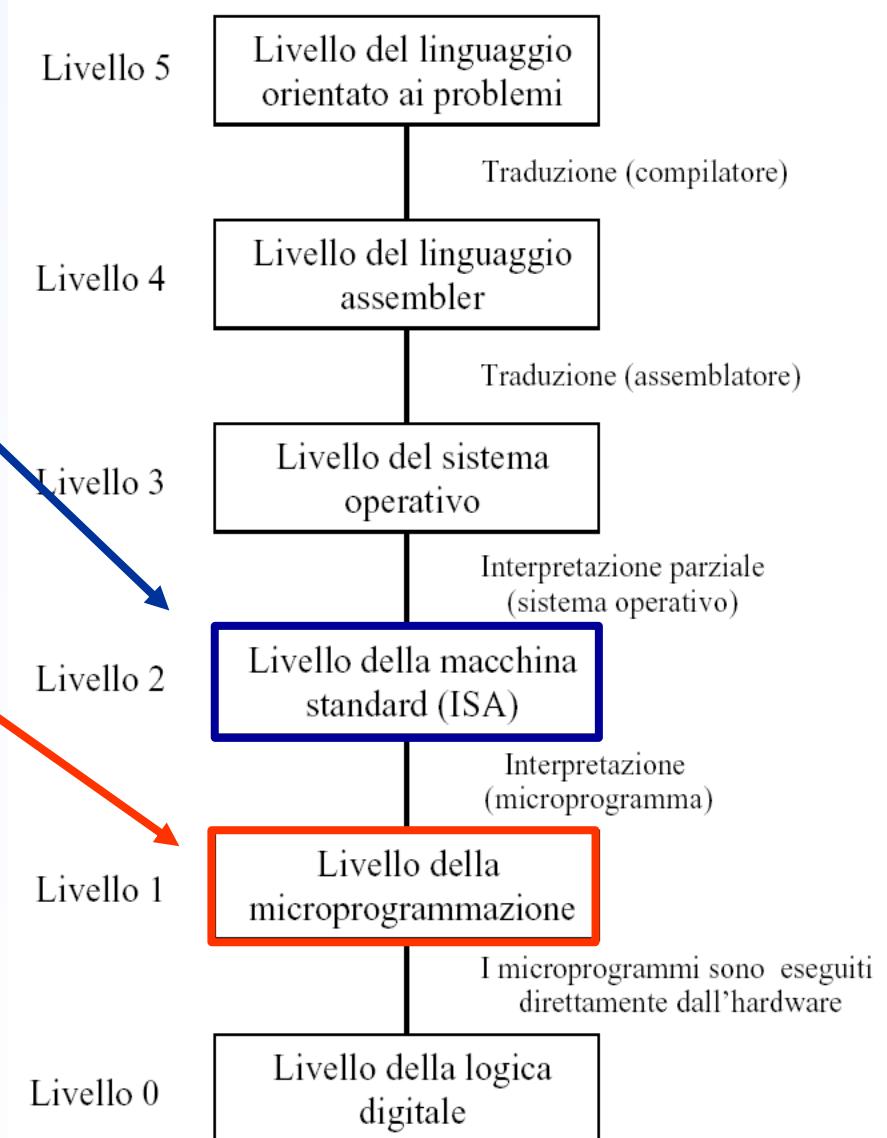
Corso di  
*Architettura degli Elaboratori*  
a.a. 2018/2019

Il Livello della microarchitettura:  
L'ISA IJVM

# Livello ISA- Livello microarchitettura

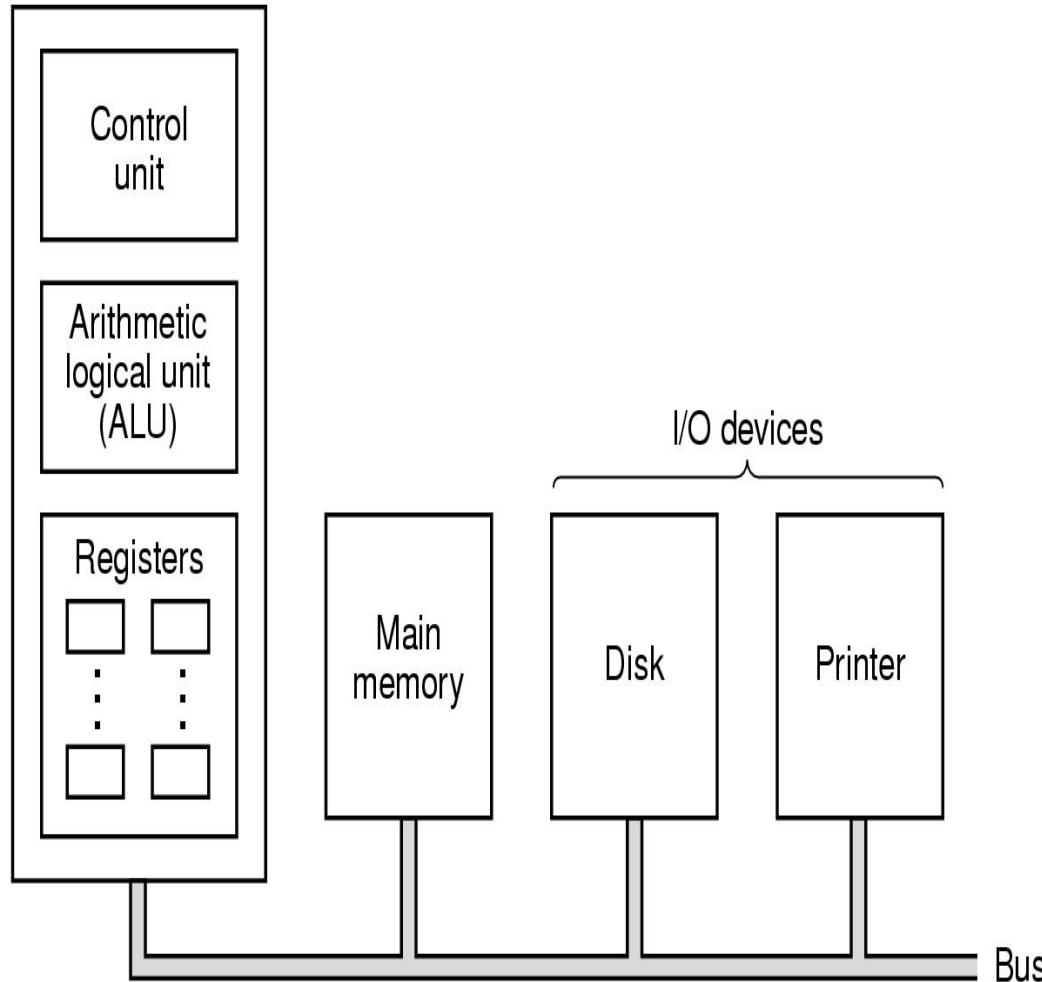
**Livello 2: ISA (Instruction Set Architecture)**

**Livello 1:** ha il compito di interpretare il livello soprastante della macchina standard



# Organizzazione della CPU in una macchina di von Neumann

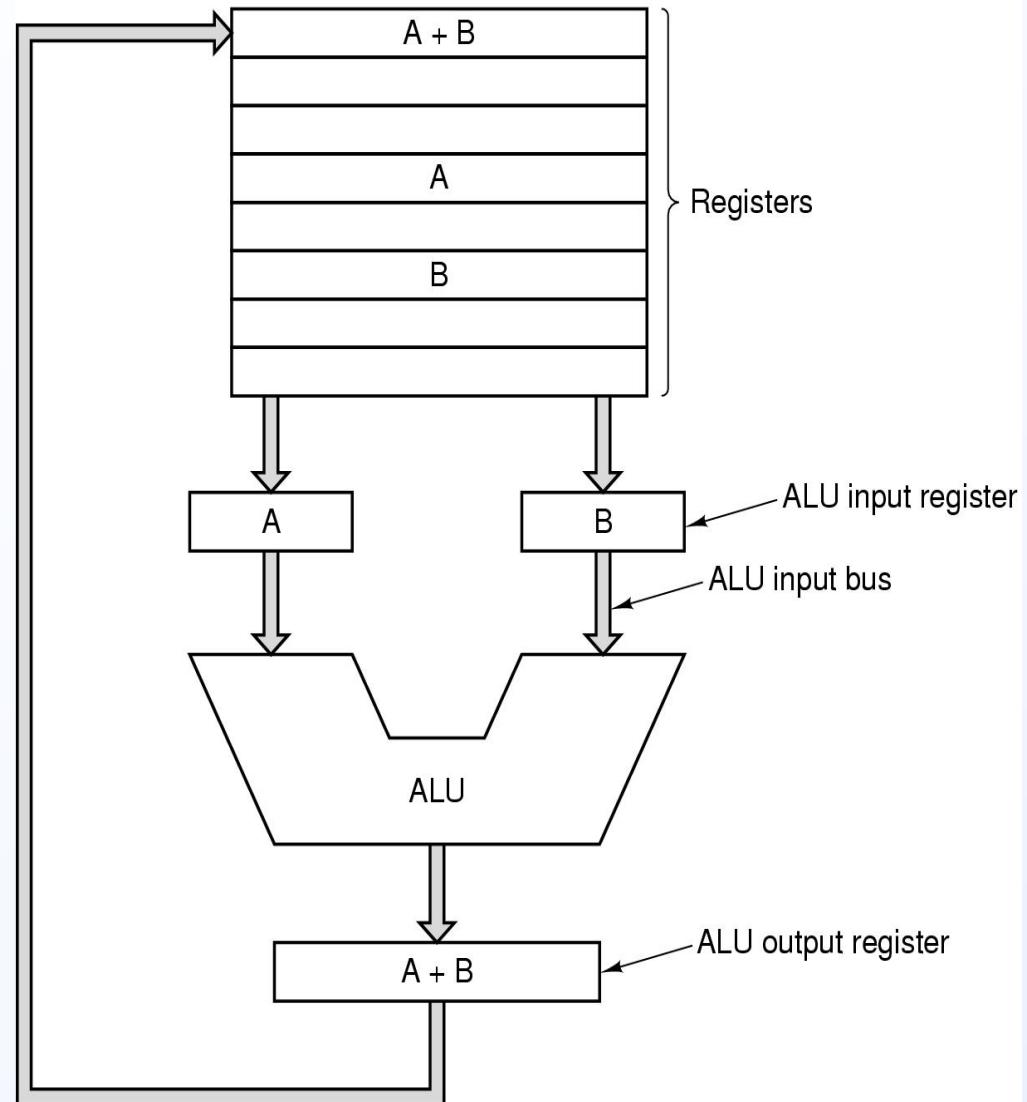
Central processing unit (CPU)



- La CPU si compone di diverse parti distinte: unità di controllo, unità aritmetico-logica, registri
- I registri, l'unità aritmetico-logica e alcuni bus che li collegano compongono il ***data path***
- Due registri importanti: ***Program Counter (PC)*** e ***Instruction Register (IR)***

# CPU di von Neumann

- **Data Path:** organizzazione interna di una CPU (registri, ALU, bus interno)
- **Registro:** memorie veloci per dati temporanei: Registri A e B, input dell'ALU, e A+B output dell'ALU
- **Istruzioni** registro-registro e registro-memoria
- **Ciclo del data path:** processo di far passare due operandi attraverso l'ALU e memorizzarne il risultato



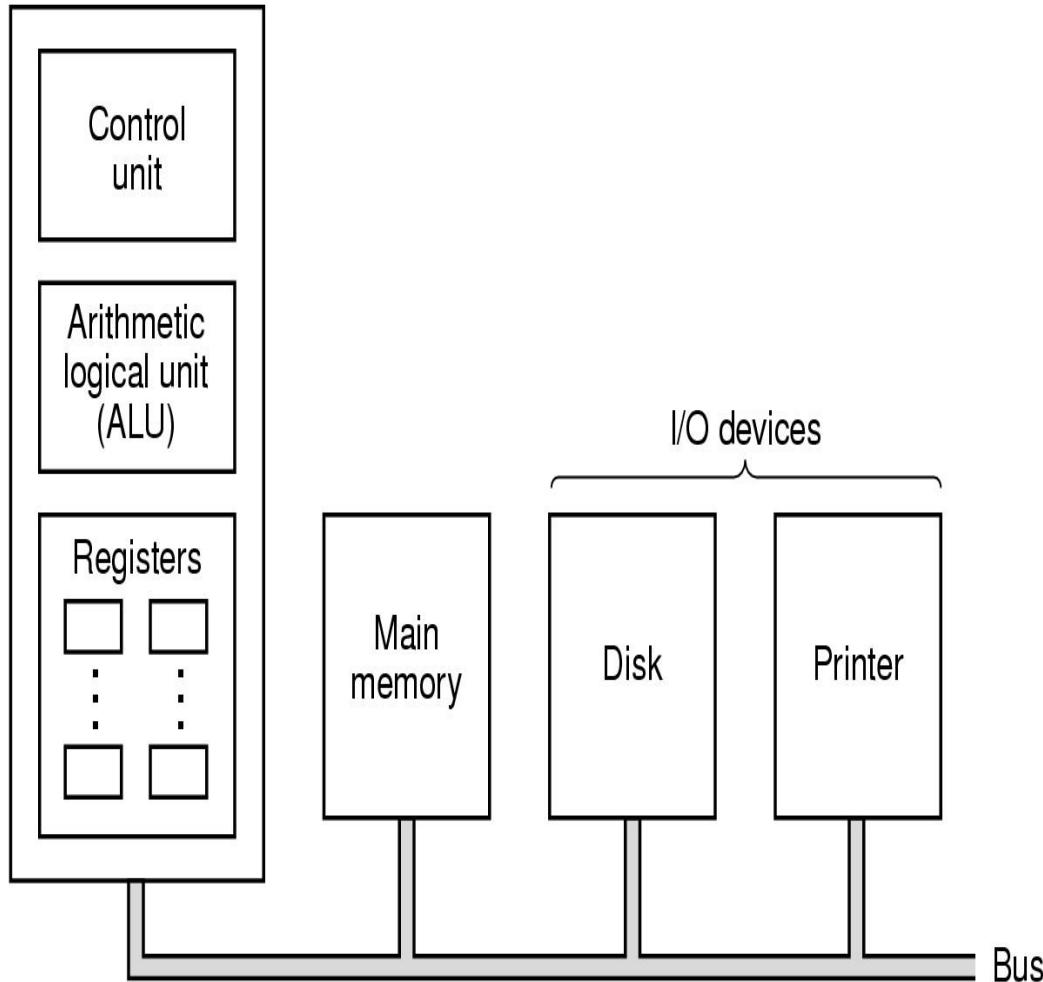
# Esecuzione delle istruzioni: ciclo di **fetch-decode-execute**

La CPU esegue ogni istruzione del livello 2 (ISA) per mezzo di una serie di *passi elementari*:

1. Prendi l'istruzione seguente dalla memoria e mettila nel **registro delle istruzioni**
2. Cambia il **program counter** per indicare l'istruzione seguente
3. Determina il **tipo** dell'istruzione appena letta
4. Se l'istruzione usa una parola in memoria, **la preleva dove**
5. **Metti la parola**, se necessario, **in un registro** della CPU
6. **Esegui** l'istruzione
7. Torna al punto 1 e inizia a eseguire l'istruzione successiva

# Organizzazione della CPU in una macchina di von Neumann

Central processing unit (CPU)



- È l'unità di controllo che esegue il ciclo di **fetch-decode-execute**: cioè legge le istruzioni dalla memoria centrale (**fetch**), ne determina il tipo (**decode**) e provvede ad eseguirle (**execute**)
- L'unità di controllo può essere vista come un **programma** che permette di **interpretare** le istruzioni ed impostare in maniera corrispondente il data path

# Microinterpretazione

**Interprete**: un programma per eseguire le istruzioni di un altro programma

*Maurice Wilkes (1951) introduce la microprogrammazione: correzione errori, estensione, test*

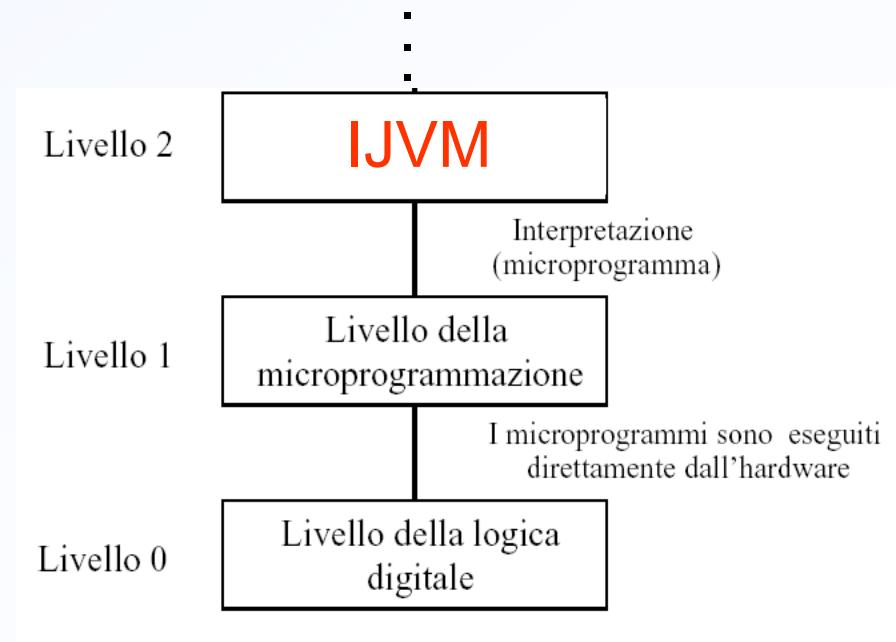
Architettura del System/360 IBM

Il linguaggio macchina è interpretato dal microinterprete che lo traduce in una sequenza di microistruzioni

***le microistruzioni controllano un ciclo di data-path***

# Una microarchitettura per IJVM

- Consideriamo come livello ISA un sottoinsieme della Java Virtual Machine che contiene solo istruzioni su numeri interi:
  - IJVM** (Integer Java Virtual Machine)
- Il microprogramma avrà il compito di leggere (fetch), decodificare ed eseguire le istruzioni IJVM mediante “cicli di data-path”
- Assumeremo che l'interprete risieda in una **ROM** dedicata



# ISA IJVM

- Insieme di 20 istruzioni
- Ciascuna costituita da un codice operazione (*opcode*) ed eventualmente un operando (*memory offset* oppure *costante*)
- Troppo semplice per scrivere programmi complicati
- Troppo complicata per essere eseguita direttamente dai circuiti

# L'insieme di istruzioni IJVM

Hex	Mnemonic	Meaning
0x10	BIPUSH <i>byte</i>	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO <i>offset</i>	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ <i>offset</i>	Pop word from stack and branch if it is zero
0x9B	IFLT <i>offset</i>	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ <i>offset</i>	Pop two words from stack; branch if equal
0x84	IINC <i>varnum const</i>	Add a constant to a local variable
0x15	ILOAD <i>varnum</i>	Push local variable onto stack
0xB6	INVOKEVIRTUAL <i>disp</i>	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE <i>varnum</i>	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W <i>index</i>	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Gli operandi *byte*, *const*, e *varnum* hanno dimensione di 1 byte, *disp*, *index*, e *offset* sono di 2 byte.

# ISA IJVM

- Tipi di istruzioni:
  - Operazioni operative
    - **aritmetiche e logiche** (IADD,ISUB,IAND, IOR)
  - Salto condizionato (IFEQ,IFLT,IF\_ICMPEQ)
  - Salto incondizionato (GOTO)
  - Dialogo con la memoria
    - **Load** di una parola da varie sorgenti sullo stack (LDC\_W, ILOAD, BIPUSH)
    - **Store** di una parola dallo stack ed eventuale suo caricamento nello local variable frame (POP, ISTORE)
  - Chiamata di procedura (INVOKEVIRTUAL, IRETURN)

# ISA IJVM

Altri tipi di istruzioni:

- Scambio delle due parole in cima allo stack (SWAP)
- Duplicazione della parola in cima allo stack (DUP)
- Conversione di formato → istruzione di prefix (WIDE)
- Chiamata di un metodo (INVOKEVIRTUAL)
- Ritorno da un metodo (IRETURN)
- NOP
- Aggiunta di una costante ad una variabile locale (INCC)

# ISA IJVM

- **BIPUSH byte**: push di un byte sullo stack
- **DUP**: copia della prima parola sullo stack e push
- **GOTO offset**: branch non condizionato
- **IADD**: pop di due parole dallo stack; push della loro somma
- **IAND**: pop di due parole dallo stack; push dell'AND logico
- **IFEQ offset**: pop di una parola e branch se e` zero
- **IFLT offset**: pop di una parola e branch se e` negativa
- **IFICMPEQ offset**: pop di due parole dallo stack; branch se sono uguali
- **IINC varnum const**: somma una costante a una variabile locale
- **ILOAD varnum**: push di una variabile locale sullo stack
- **INVOKEVIRTUAL disp**: chiama un metodo

# ISA IJVM

- ***IOR***: pop di due parole dallo stack; push dell'OR logico
- ***IRETURN***: ritorno da un metodo con un valore intero
- ***ISTORE varnum***: pop una parola dallo stack e memorizzala in una variabile locale
- ***ISUB***: pop due parole dallo stack; push la loro differenza
- ***LDCW index***: push di una costante dalla constant pool sullo stack
- ***NOP***: nessuna operazione
- ***POP***: cancella una parola
- ***SWAP***: scambia le due parole in cima allo stack
- ***WIDE***: prefisso; l'istruzione seguente ha un indirizzo di 16 bit

# Da Java a JVM

y = x + 1;



```
ILOAD x  
BIPUSH 1  
IADD  
ISTORE y
```

# Non dare niente per scontato

- $y = x + 1;$
- Dove metto  $x$  e  $1$  prima di sommarli?
- Dopo la somma, dove metto il risultato di  $x + 1$  prima di assegnarlo a  $y$ ?
- Solo apparentemente semplice. Cf:  
 $x = (x + 1) * (y + 2) * ((y + 3) * x)....$
- Come faccio ad avere una istruzione di somma che non deve gestire la differenza fra una variabile ed un valore come operando?
- ...e poi: dove sono le parentesi in JVM?

# Non dare niente per scontato

- $x = x + 1;$
- Dove metto  $x$  e  $1$  prima di sommarli?
- “Li metto nei registri!”
- Quanti ce ne vogliono per  
 $x = (x + 1) * (y + 2) * ((y + 3) * x) \dots ?$
- Troppi ... (apparentemente)

# Soluzione: uso uno stack

- La JVM per fare operazioni matematiche e logiche su dei valori e per restituire un risultato (e per gestire l'assenza delle parentesi) usa una struttura dati: lo **stack**
- Cos'è lo stack?
- Metto operandi su stack
- Operazione in linguaggio macchina mette il risultato su stack
- Il risultato può diventare operando di altre operazioni
- Operazione eseguita “dopo” gli operandi (notazione polacca inversa RPN: *Reverse Polish Notation*)
- Stack facile da implementare!

# Stack per operandi

- Metto due operandi sullo stack, eseguo l'operazione, e li sostituisco sullo stack con il risultato
- $(2 + 2) * 3$  diventa: 2 2 + 3 \*

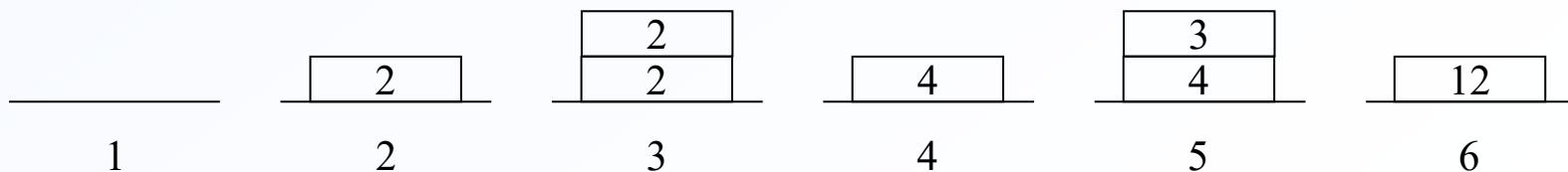
1: Metti 2 sullo stack,

2: Metti 2 sullo stack,

3: Somma 2 e 2 e sostituiscili con 4

4: Metti 3 sullo stack

5: Esegui il prodotto e sostituisci 4 e 3 con 12 sullo stack



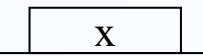
# Traduzione

$z = (x + y) * ((z - y) * x);$

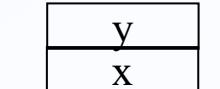
diventa in RPN

$x\ y\ +\ z\ y\ -\ x\ * \ *$

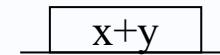
6: iload\_1



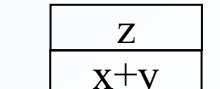
7: iload\_2



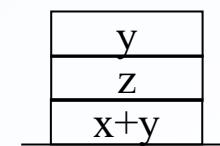
8: iadd



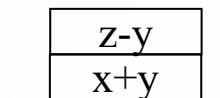
9: iload\_3



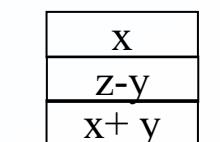
10: iload\_2



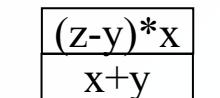
11: isub



12: iload\_1



13: imul



14: imul

$(x+y)*((z-y)x)$

# Traduzione in ISA IJVM

1	ILOAD j
2	ILOAD k
3	IADD
4	ISTORE i

```
i = j + k;  
if (i == 3)  
    k = 0;  
else  
    j = j - 1;
```

Per l'esecuzione dell'espressione aritmetica di sfrutta l'*operand stack*

$i = j + k$  viene tradotto in:

- carica il valore di  $j$  nello stack
- carica il valore di  $k$  nello stack
- somma i due valori nello stack
- memorizza il risultato (che sta sul top dello stack) nella variabile  $i$

# Traduzione in ISA IJVM

i = j + k;	1	ILOAD j
if (i == 3)	2	ILOAD k
k = 0;	3	IADD
else	4	ISTORE i
j = j - 1;	5	ILOAD i
	6	BIPUSH 3
	7	IFICMPEQ L1
	8	GOTO L2
	9	L1: ...

uso dello stack anche per l'espressione booleana del test:

- . carico il valore di i
- . carico la costante 3
- . mi chiedo se questi sono uguali e se sì prevedo di saltare “due istruzioni avanti” altrimenti prevedo di saltare “in un qualche punto molto più avanti”

# Traduzione in ISA IJVM

i = j + k;	1	ILOAD j
if (i == 3)	2	ILOAD k
k = 0;	3	IADD
else	4	ISTORE i
j = j - 1;	5	ILOAD i
	6	BIPUSH 3
	7	IFICMPEQ L1
	8	GOTO L2
	9	L1: BIPUSH 0
	10	ISTORE k
	11	GOTO L3
	12	L2: ...

Traduco il contenuto del primo ramo dell'istruzione if

- Alla conclusione delle istruzioni nel primo ramo devo assicurarmi che il flusso di esecuzione non proseguia attraverso il secondo ramo dell'if!!
- Ora conosco dove è collocata l'istruzione L2

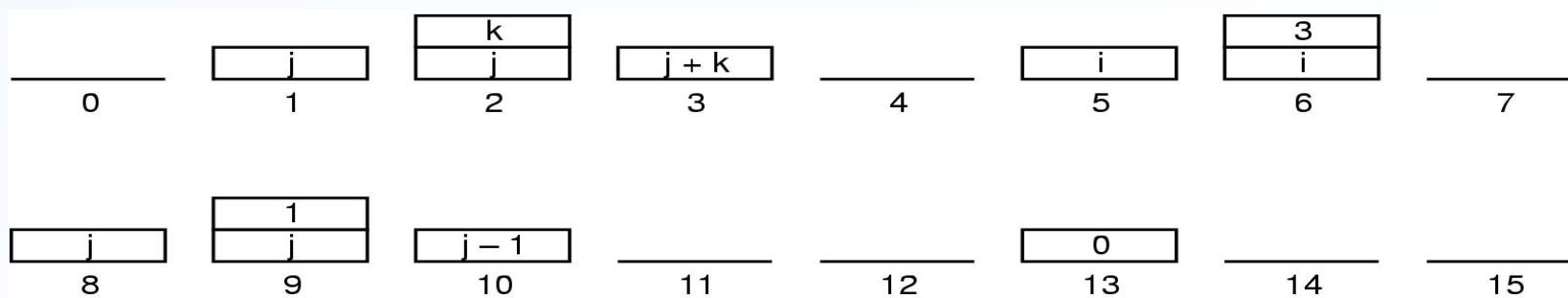
# Traduzione in ISA IJVM

i = j + k;	1	ILOAD j	
if (i == 3)	2	ILOAD k	
k = 0;	3	IADD	
else	4	ISTORE i	
j = j - 1;	5	ILOAD i	
	6	BIPUSH 3	
	7	IFICMPEQ L1	
	8	GOTO L2	
	9	L1: BIPUSH 0	
	10	ISTORE k	
	11	GOTO L3	
	12	L2: ILOAD j	
	13	BIPUSH 1	
	14	ISUB	
	15	ISTORE j	
	16	L3: ...	

- Viene tradotto di seguito anche il secondo ramo dell'istruzione if
- Al termine si conosce anche la posizione dell'istruzione L3

# Traduzione in ISA IJVM

i = j + k;	1	ILOAD j
if (i == 3)	2	ILOAD k
k = 0;	3	IADD
else	4	ISTORE i
j = j - 1;	5	ILOAD i
	6	BIPUSH 3
	7	IFICMPEQ L1
	8	ILOAD j
	9	BIPUSH 1
	10	ISUB
	11	ISTORE j
	12	GOTO L2
	13 L1:	BIPUSH 0
	14	ISTORE k
	15 L2:	...



# Traduzione in ISA IJVM

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

(a)

(b)

(c)

(b) Un codice JAVA per il frammento (a).

(c) Il programma IJVM in esadecimale.

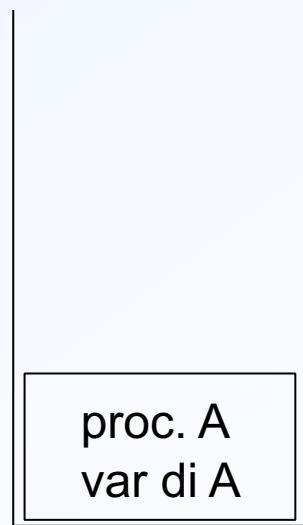
# L'insieme di istruzioni IJVM

Hex	Mnemonic	Meaning
0x10	BIPUSH <i>byte</i>	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO <i>offset</i>	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ <i>offset</i>	Pop word from stack and branch if it is zero
0x9B	IFLT <i>offset</i>	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ <i>offset</i>	Pop two words from stack; branch if equal
0x84	IINC <i>varnum const</i>	Add a constant to a local variable
0x15	ILOAD <i>varnum</i>	Push local variable onto stack
0xB6	INVOKEVIRTUAL <i>disp</i>	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE <i>varnum</i>	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W <i>index</i>	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Gli operandi *byte*, *const*, e *varnum* hanno dimensione di 1 byte, *disp*, *index*, e *offset* sono di 2 byte.

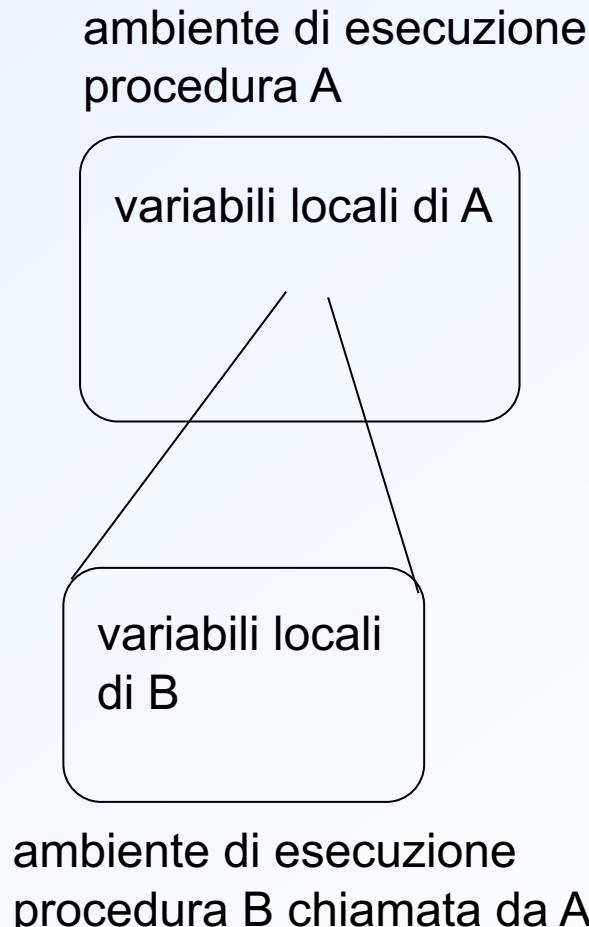
# Il modello di memoria dell'ISA IJVM

ambiente di esecuzione  
procedura A



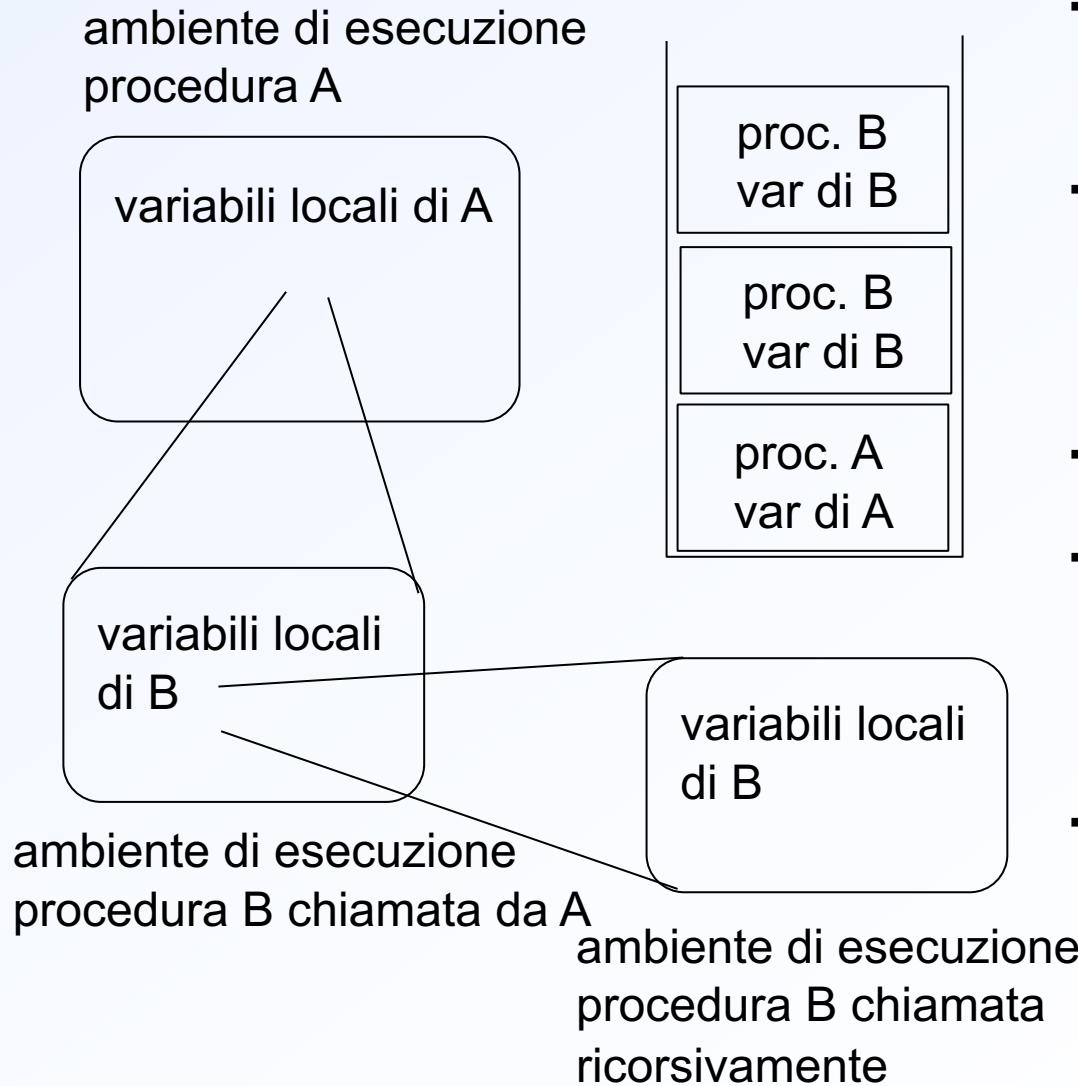
- Procedure (metodi) e variabili locali
- Chiamata di una procedura come apertura di un nuovo ambiente
- Chiamate ricorsive
- Le variabili locali esistono fintanto che la procedure e` in esecuzione
- Usare uno stack di ambienti!

# Procedure e variabili locali



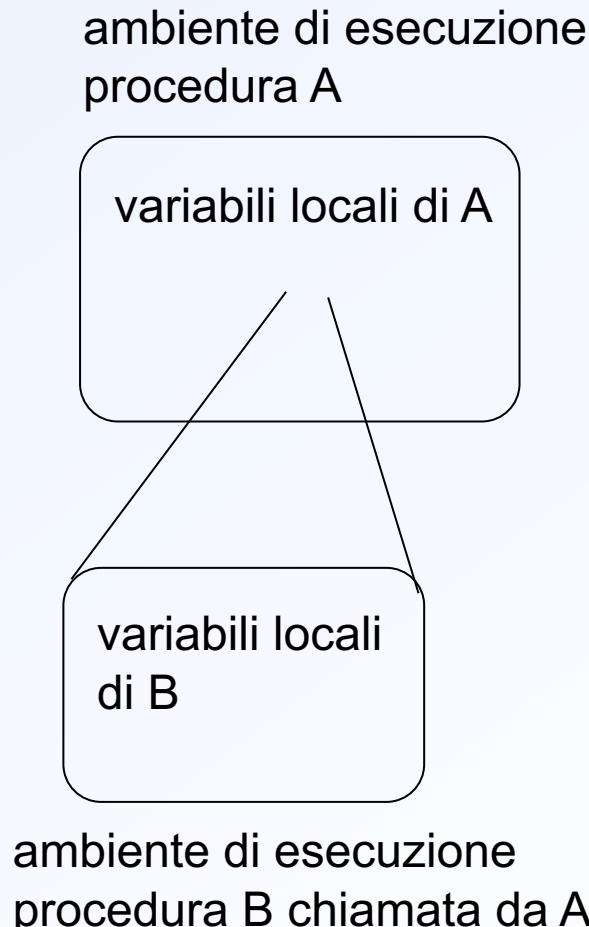
- Procedure (metodi) e variabili locali
- Chiamata di una procedura come apertura di un nuovo ambiente
- Chiamate ricorsive
- Le variabile locali esistono fintanto che la procedure e` in esecuzione
- Usare uno stack di ambienti!

# Procedure e variabili locali



- Procedure (metodi) e variabili locali
- Chiamata di una procedura come apertura di un nuovo ambiente
- Chiamate ricorsive
- Le variabile locali esistono fintanto che la procedure e` in esecuzione
- Usare uno stack di ambienti!

# Procedure e variabili locali

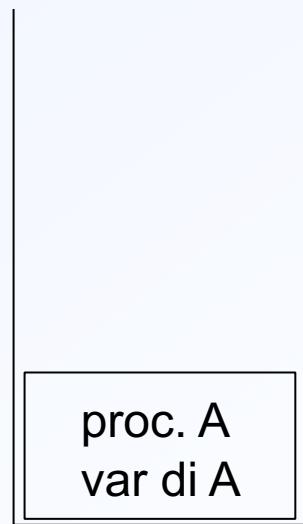


- Procedure (metodi) e variabili locali
- Chiamata di una procedura come apertura di un nuovo ambiente
- Chiamate ricorsive
- Le variabile locali esistono fintanto che la procedure e` in esecuzione
- Usare uno stack di ambienti!

# Procedure e variabili locali

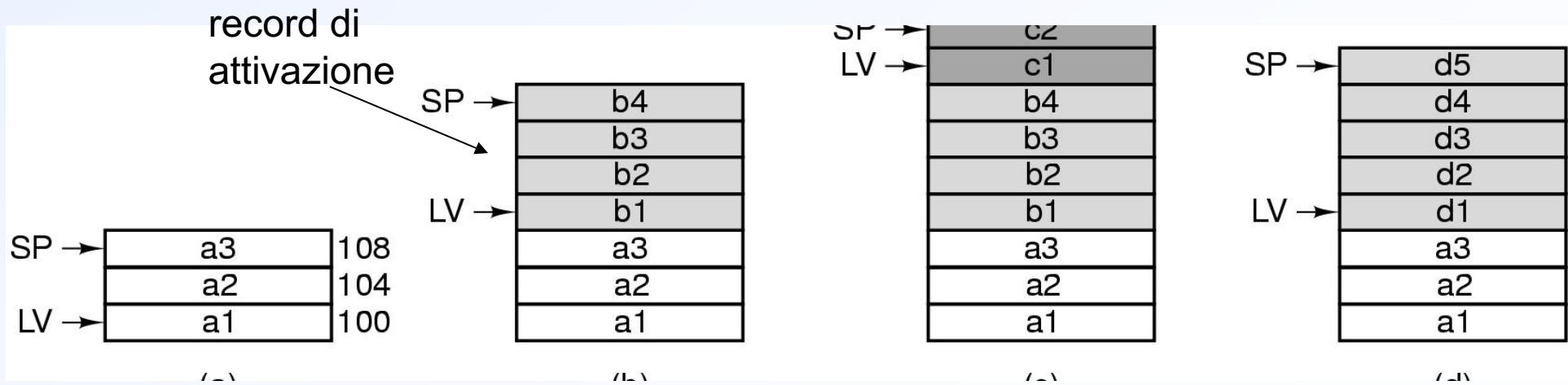
ambiente di esecuzione  
procedura A

variabili locali di A



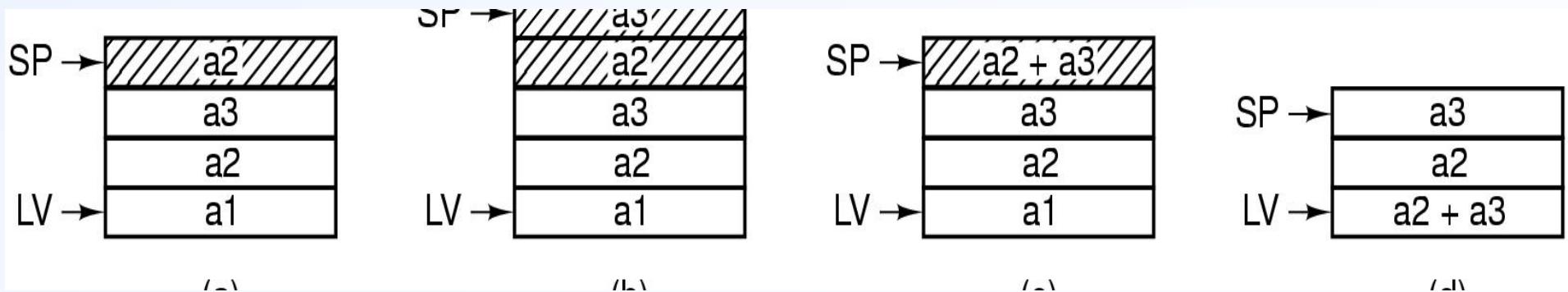
- Procedure (metodi) e variabili locali
- Chiamata di una procedura come apertura di un nuovo ambiente
- Chiamate ricorsive
- Le variabile locali esistono fintanto che la procedure è in esecuzione
- Usare uno stack di ambienti!

# Lo stack di esecuzione



- Non vengono assegnati indirizzi assoluti alle variabili locali ma solo relativi (**offset**) alla base del **record di attivazione**
- Un registro (LV) punterà sempre alla base del record di attivazione della procedura (metodo) in esecuzione**
- Un registro (SP) punterà sempre al top dello stack**

# Uno stack anche per i calcoli



- La struttura a stack può essere utilizzata in modo proficuo anche per tenere gli operandi durante il calcolo di una espressione aritmetica (***operand stack***)
- Lo stack di esecuzione e gli operand stack possono coesistere
- Macchina a zero indirizzi

# Memoria Centrale di IJVM

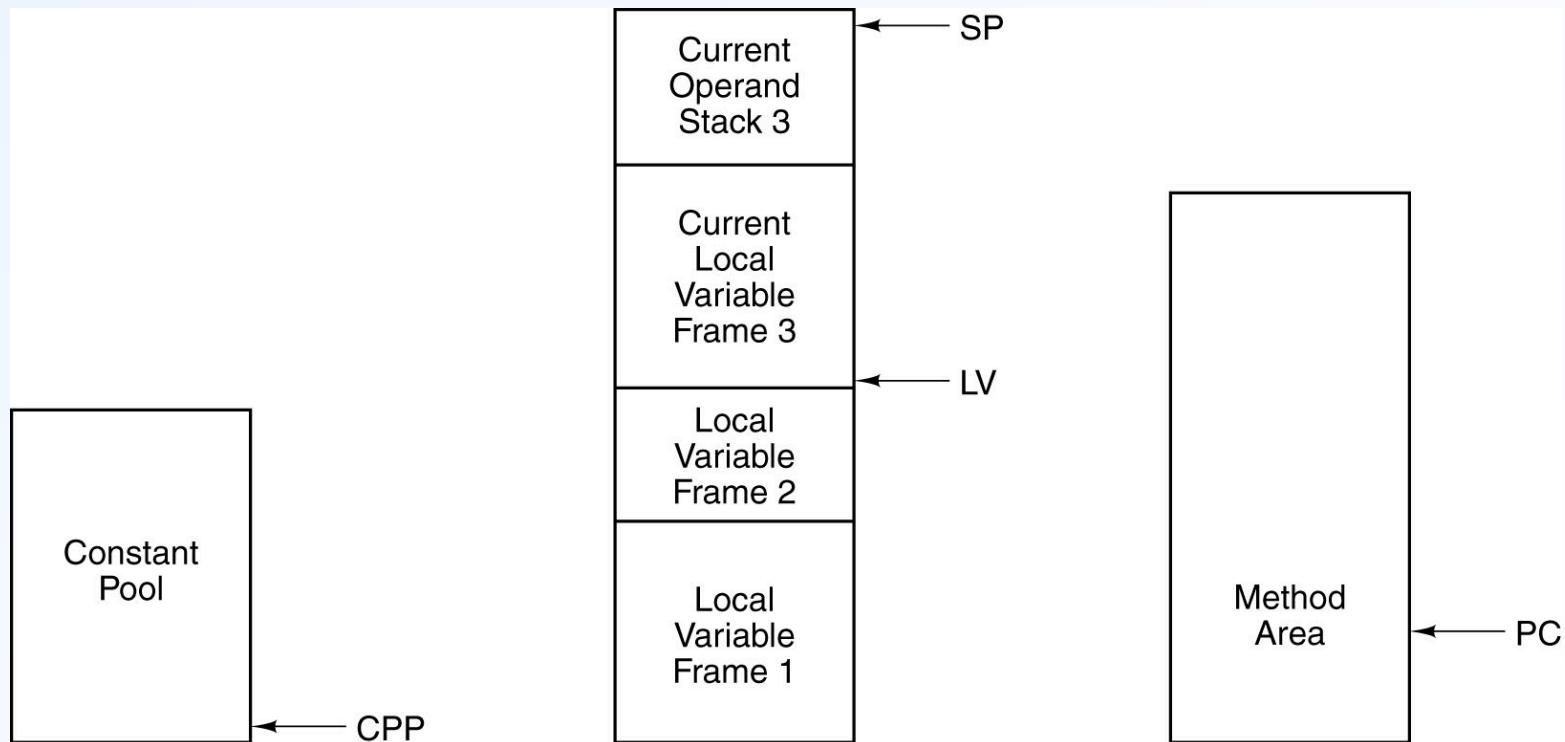
- La macchina dell' ISA IJVM dispone di uno spazio di memoria centrale di **4GB**, cioè 4G di *celle di 1B* che si possono anche interpretare come una RAM di 1 G di *parole di 4B*
- Le istruzioni ISA non usano direttamente gli indirizzi di RAM ma operano sulla memoria utilizzando degli appositi registri come **puntatori** (PC, SP, LV, ecc..); ad esempio le operazioni logico matematiche usano il puntatore al top dello stack (registro. SP)

# Memoria Centrale di IJVM

La memoria centrale del processore IJVM è suddivisa in 4 aree principali (non necessariamente tutte contigue):

- **area delle costanti** (*Constant Pool Area*);
- **stack di esecuzione** (*Frame Stack Area*);
- **stack degli operandi** (*Operand Stack Area*);
- **area del codice eseguibile** (*Executable Code Area*)

# IJVM: organizzazione della memoria



**Costant pool**: caricato quando un programma viene portato in memoria, read-only, contiene costanti, stringhe, puntatori ad altre aree, registro **CPP**

**Stack di esecuzione**: registro **LV**

**Operand stack**: registro **SP**

**Area dei metodi** (codice): registro **PC**

CPP, LV, SP sono puntatori a parole (di 4 byte), PC a byte

# Memoria Centrale di IJVM

## Area delle Costanti

- Contiene *costanti* numeriche e di altro tipo, sequenze fisse di caratteri e puntatori ad altre aree di memoria
- L' area viene inizializzata quando il programma viene *caricato*
- Il programma *non può modificare* il contenuto dell'area
- Il programma accede all'area tramite il *registro puntatore CPP*

# Memoria Centrale di IJVM

## Stack di esecuzione

Contiene:

- *puntatori di servizio*, necessari per il meccanismo di chiamata/ritorno di metodi (sottoprogrammi);
- i *parametri* passati al programma (o al sotto programma) all'atto della sua chiamata;
- lo spazio riservato per le *variabili locali*.
- Il registro **LV**, contenuto nella CPU, viene usato per puntare alla base dello stack di esecuzione della procedura correntemente in esecuzione

# Memoria Centrale di IJVM

## Stack di esecuzione

- Lo stack di esecuzione viene allocato sul top dello stack quando la *procedura associata entra in esecuzione*.
- La procedura correntemente in esecuzione *usa le variabili locali e i parametri* contenuti nel proprio stack di esecuzione.
- Quando la procedura termina, lo stack di esecuzione viene *eliminato*.

# Memoria Centrale di IJVM

## Stack degli operandi

- Sopra lo stack di esecuzione viene costruito lo stack degli operandi, nei momenti in cui il programma deve calcolare un'espressione logico-matematica.
- Il registro **SP**, contenuto nella CPU, punta alla cima dello stack, quindi:
  - alla cima dello stack di esecuzione
  - oppure alla cima dello stack degli operandi, quando questa non è vuota

# Memoria Centrale di IJVM

## Area del codice eseguibile

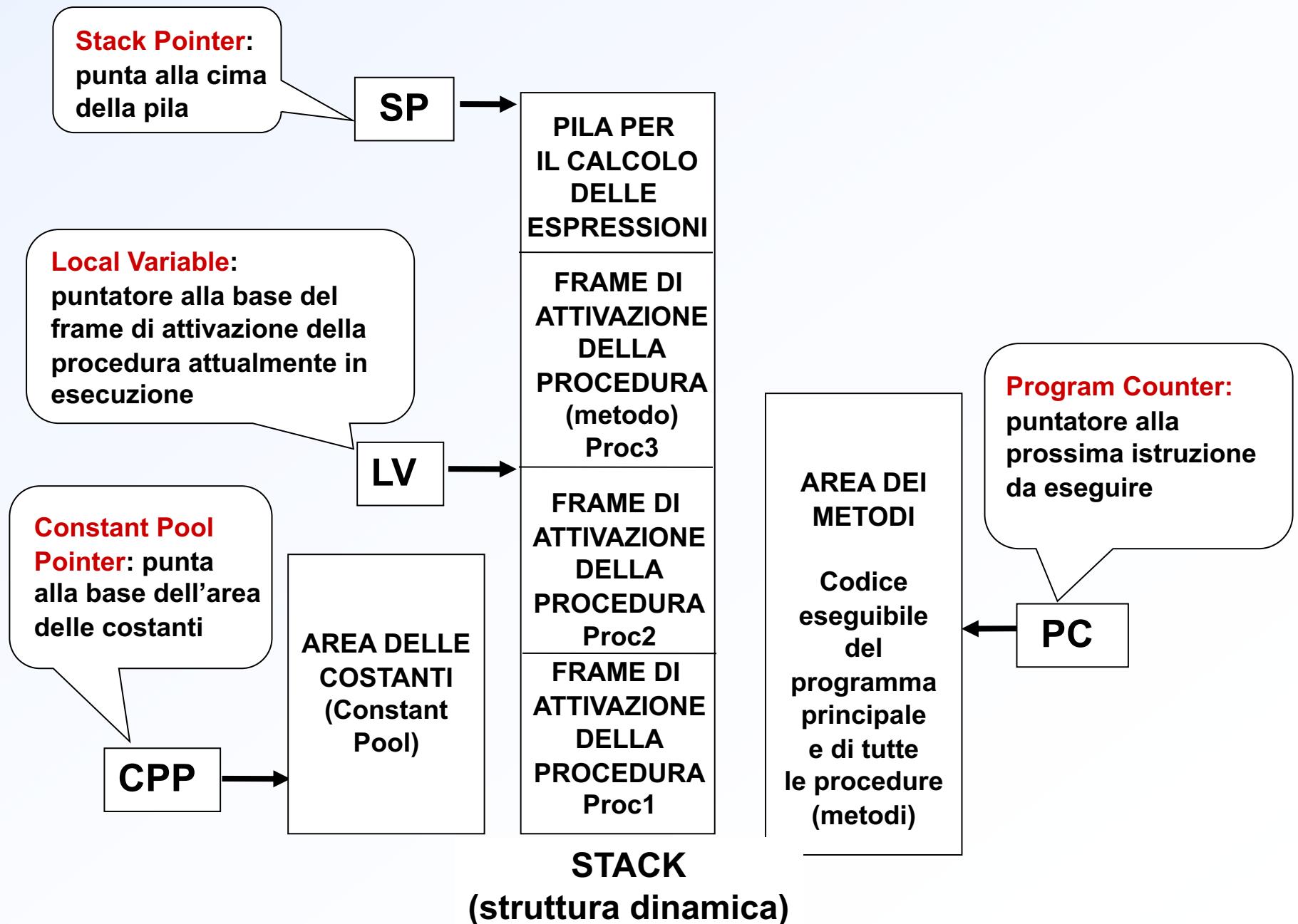
- Contiene il **codice eseguibile** del programma principale e delle eventuali procedure e/o funzioni facenti parte del programma complessivo da eseguire
- Viene **caricato in memoria** quando il programma entra in esecuzione
- Quando il programma **termina** diventa inutile e può essere **cancellato**
- Il registro **PC**, contenuto nella CPU, viene usato per puntare alla prossima istruzione macchina da eseguire

# Memoria Centrale di IJVM

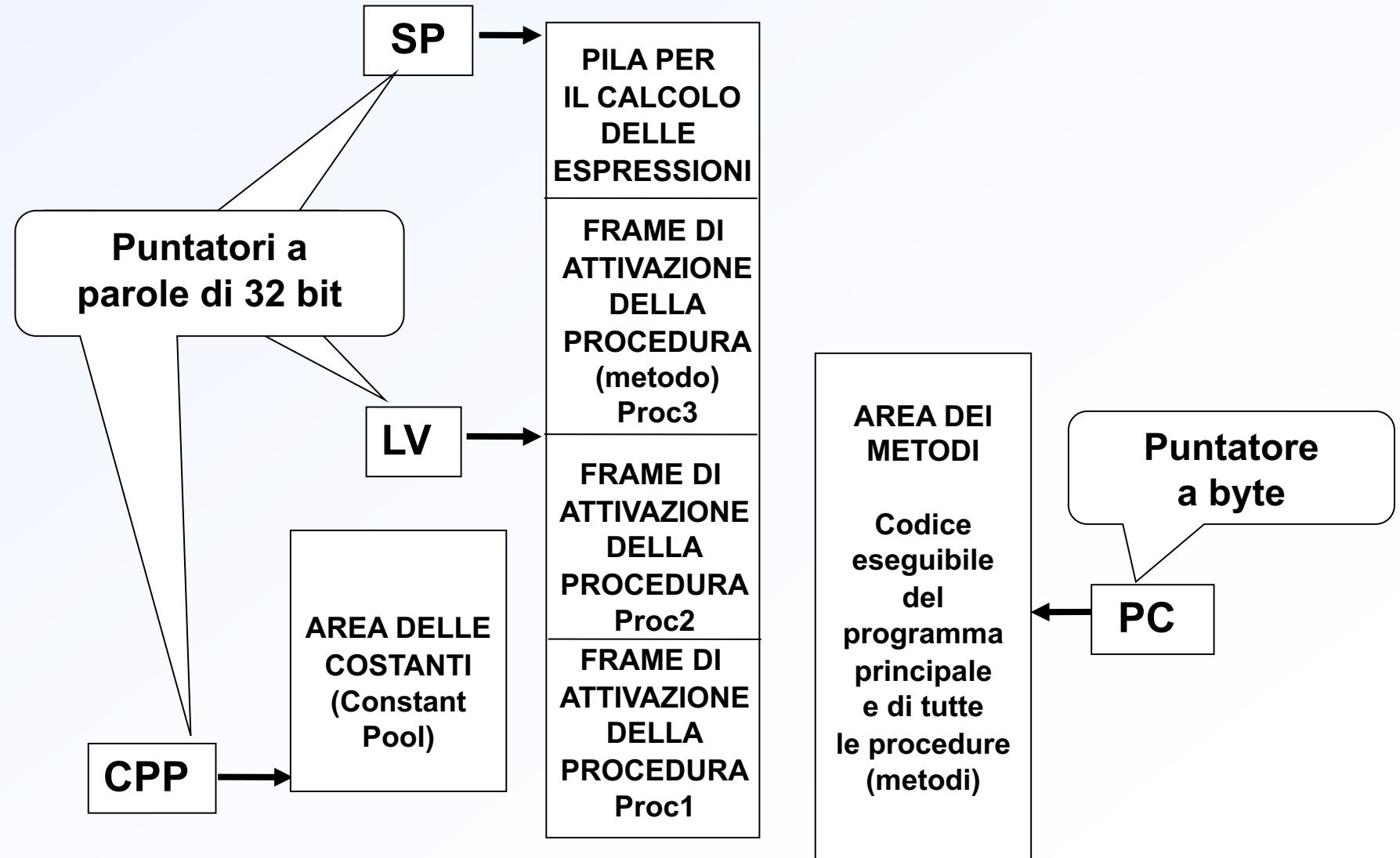
## Precisazione sui registri

- I registri *CPP*, *LV* e *SP* contengono indirizzi a *parole di 4B*, quindi:
  - leggere la parola il cui indirizzo è contenuto in SP significa leggere la parola di 32 bit, posta sulla cima dello stack
- Il registro *PC* contiene l'indirizzo di 1 Byte, quindi
  - leggere la parola il cui indirizzo è contenuto in PC, significa leggere il byte facente parte dell'istruzione da mandare in esecuzione.

# IJVM: organizzazione della memoria



# IJVM: organizzazione della memoria



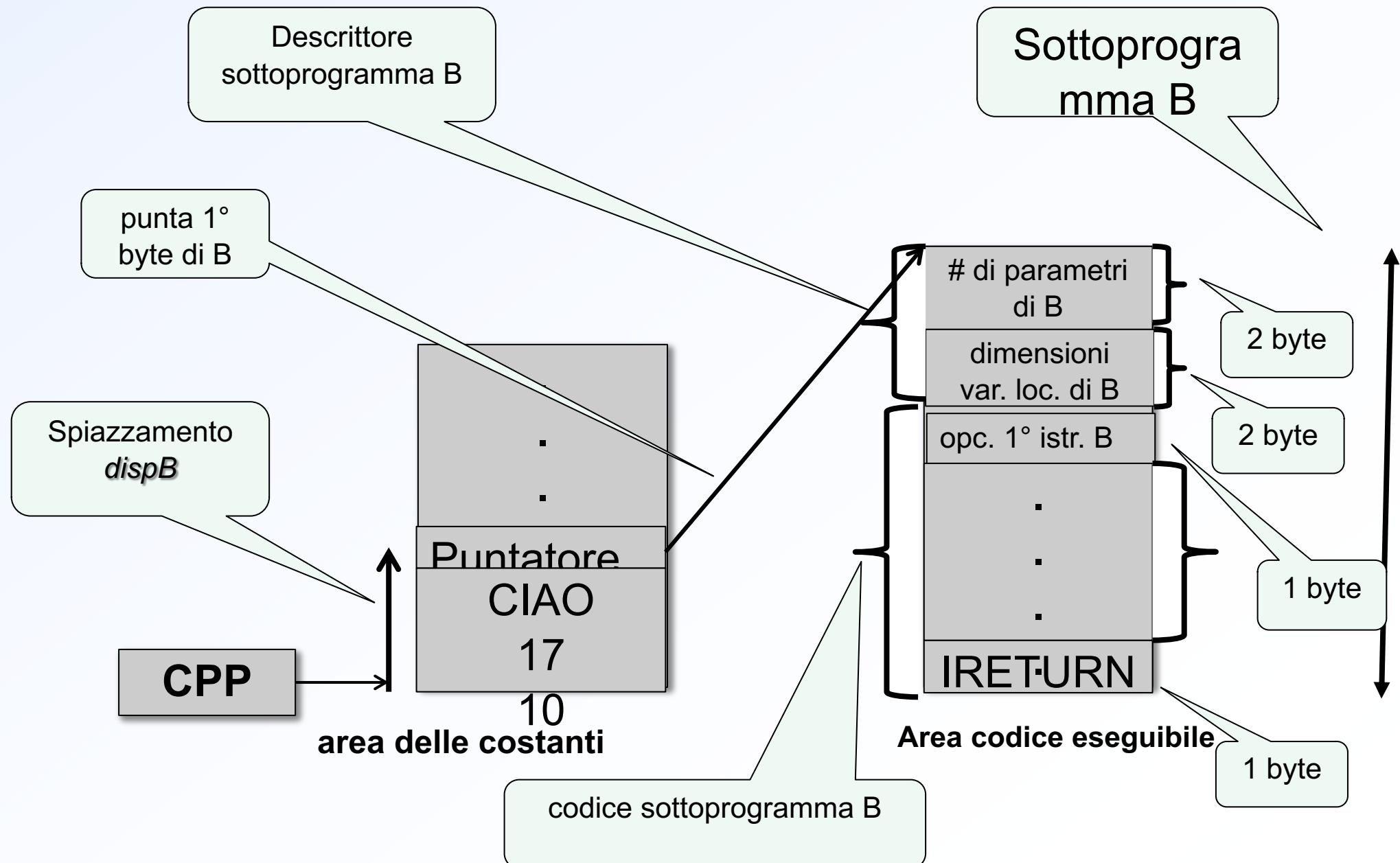
# Sottoprogrammi

- Il linguaggio macchina ISA IJVM ammette come sottoprogrammi solo *funzioni* (un valore di ritorno)
- Una funzione IJVM restituisce sempre *un numero intero come valore di output*
- Nell' ISA IJVM esistono *due istruzioni macchina* per gestire la chiamata ad un sottoprogramma:
  - **INVOKEVIRTUAL** *disp*: il programma chiamante attiva il sottoprogramma specificato da *disp*
  - **IRETURN**: il sottoprogramma chiamato ritorna al chiamante restituendogli un numero intero

# Specifiche del chiamato

- Per specificare la collocazione e le caratteristiche del sottoprogramma chiamato, l'istruzione **INVOKEVIRTUAL** adotta un apposito formato di specifica
- Tale formato si basa su un «**descrittore**» del sottoprogramma da chiamare, contenente le informazioni necessarie e inserito in cima al codice del sottoprogramma
- Nell' **area delle costanti** viene mantenuto il puntatore al codice di B

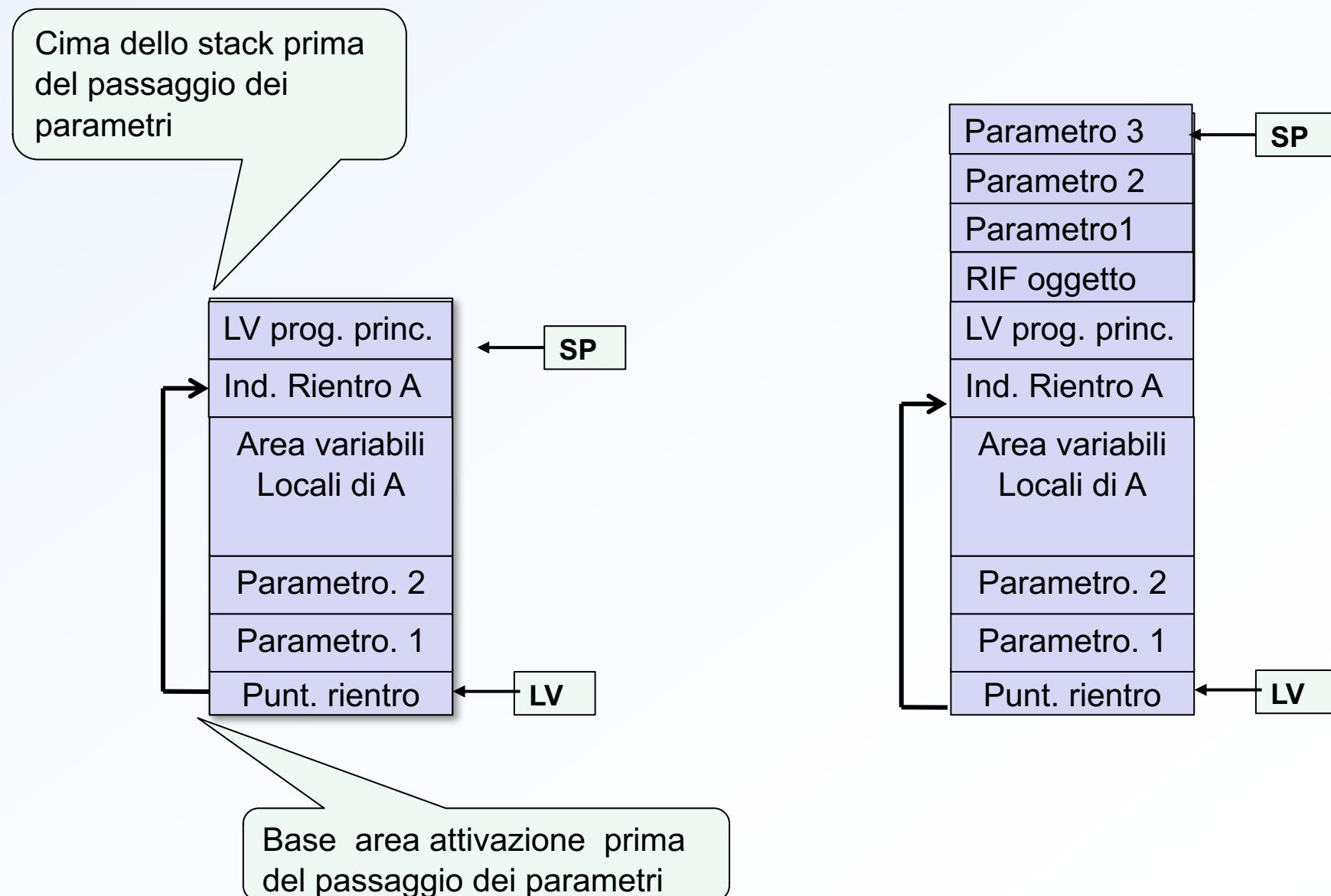
# Esempio: INVOKEVIRTUAL *dispB*



# Funzionamento della chiamata

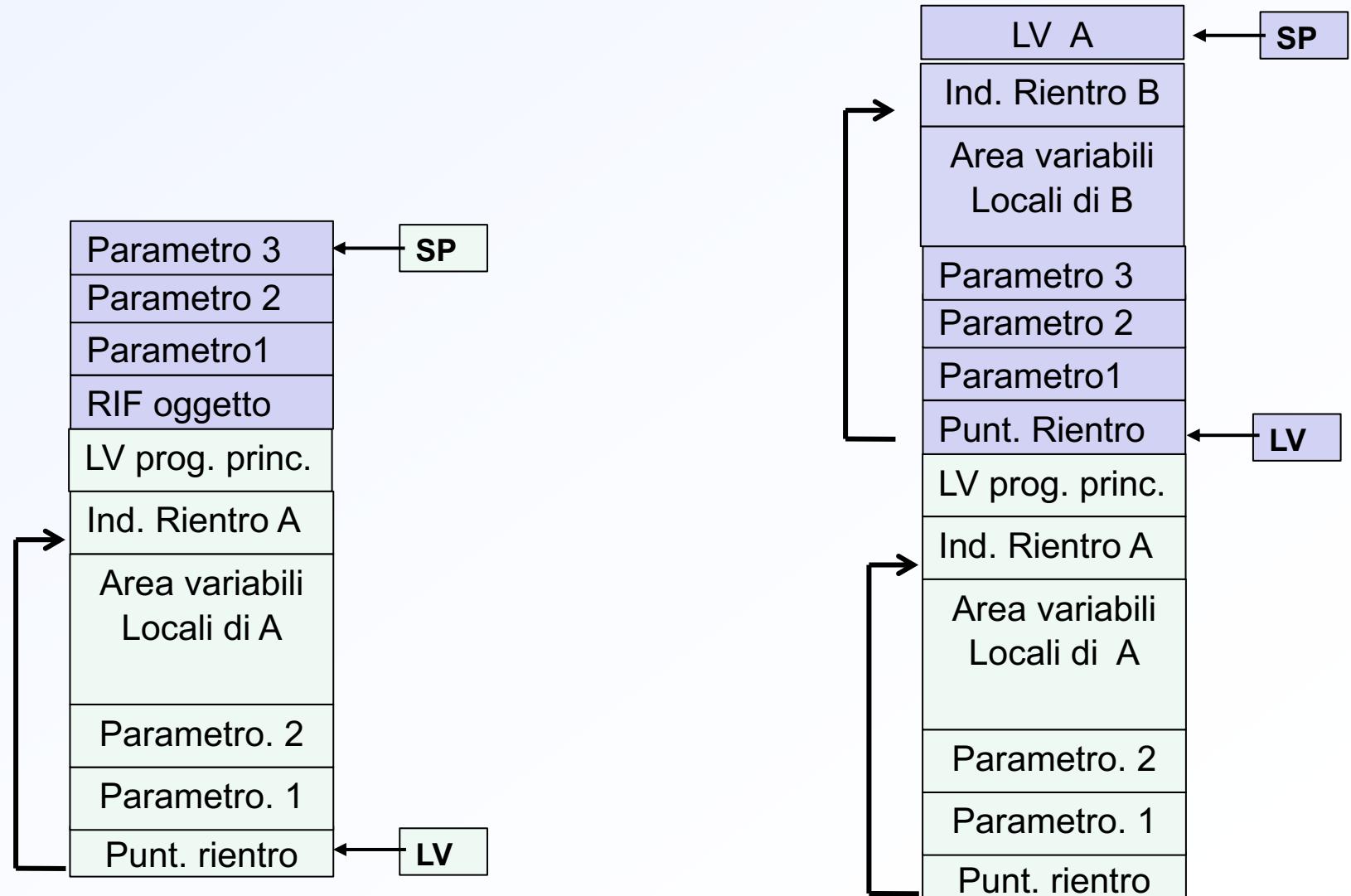
- Il programma chiamante scrive sulla cima dello stack il riferimento all'oggetto OBJREF (pleonastico) e i parametri da passare al sottoprogramma chiamato
- Il chiamante esegue l'istruzione ***INVOKEVIRTUAL disp*** dove ***disp*** indica il sottoprogramma cui passare l'esecuzione
- L'istruzione ***INVOKEVIRTUAL*** predisponde l'ambiente per il passaggio al sottoprogramma:
  - Crea l'area di attivazione del chiamato sul top dello stack
  - Inserisce in quest'area le informazioni necessarie per il ritorno
  - Aggiorna i registri ***LV***, ***SP*** e ***PC***
  - Lancia l'esecuzione del sottoprogramma

# Chiamata di un sottoprogramma(1) preparazione dei parametri di input



# Chiamata di un sottoprogramma(2)

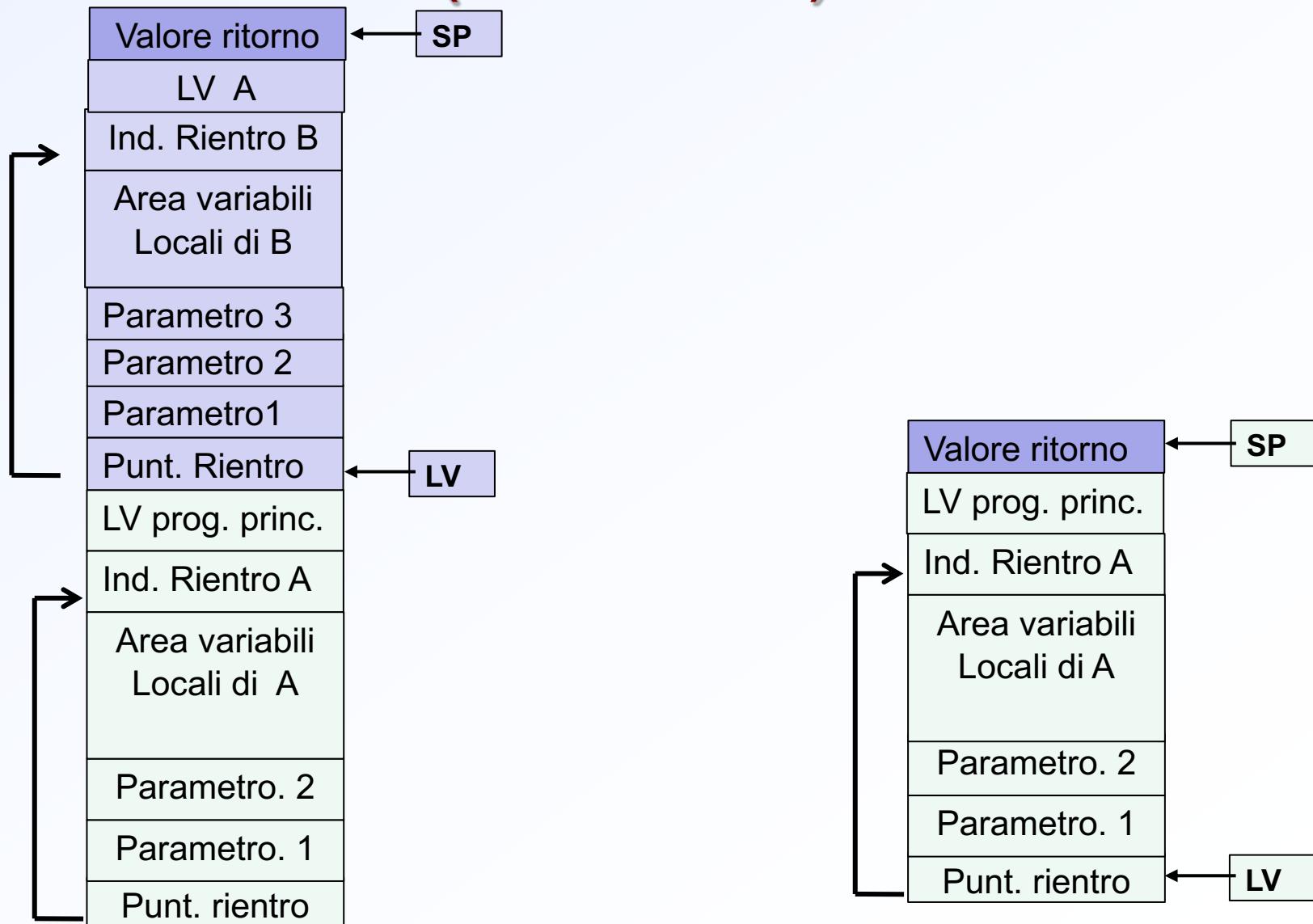
A chiama la funzione B (invokevirtual)



# Funzionamento del ritorno

- Il sottoprogramma chiamato scrive sulla cima dello stack il valore da restituire al chiamante (un intero)
- Il sottoprogramma chiamato esegue l'istruzione **IRETURN** che riporta l'esecuzione al sottoprogramma chiamante, all'istruzione successiva a **INVOKEVIRTUAL**
- L'istruzione **IRETURN** ricostruisce l'ambiente di esecuzione del chiamante:
  - elimina l'area di attivazione del chiamato
  - lascia sulla cima dello stack il valore restituito
  - ripristina i registri **LV**, **SP** e **PC** ai valori precedenti la chiamata
  - riporta l'esecuzione al chiamante

# Rientro da sottoprogramma (IRETURN)



# INVOKEVIRTUAL: esempio

## Codice Assembler

.main

```
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
BIPUSH 1
:
```

.

method exm (J,K)

```
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

La sequenza di istruzioni IJVM chiama un metodo **esempio (J, K)** caratterizzato da due parametri che calcola  $2^*(J+K)$

Il metodo viene chiamato sui parametri  
 $J = 5$  e  $K = 7$

# INVOKEVIRTUAL: esempio

## Codice Assembler

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
.
method exm (J,K)
    ILOAD J
    ILOAD K
    IADD
    DUP
    IADD
    IRETURN
.end method
```

## address Area codice eseguibile

address	Area	codice eseguibile
0x250		
0x251		
0x252		...
0x254		0x13 0x15
0x256		0x10 0x05
0x258		0x10 0x07
0x25A		0xB6 0x21
:		:
:		:
0x3F1		
0x3F5		0x00 0x03 0x00 0x00
0x3F7		0x15 0x01
0x3F9		0x15 0x02
0x3FA		0x60
0x3FB		0x59
0x3FC		0x60
		0xAC
		:

# INVOKEVIRTUAL: esempio

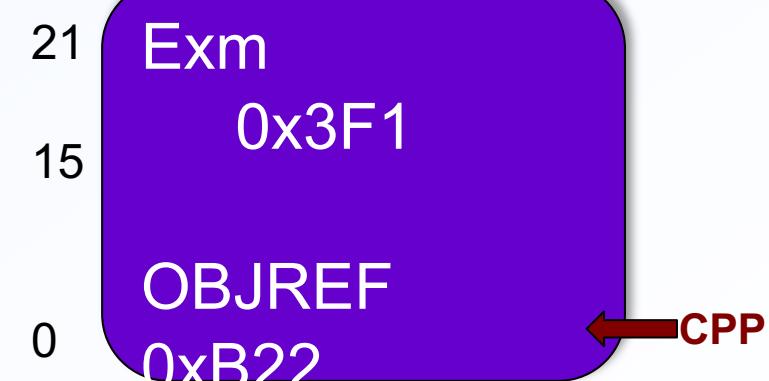
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	...
0x251	...
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21
:	:
:	:
:	:
0x3F1	0x00 0x03 0x00 0x00
0x3F5	0x15 0x01
0x3F7	0x15 0x02
0x3F9	0x60
0x3FA	0x59
0x3FB	0x60
0x3FC	0xAC
:	:

## Constant Pool



## STACK

SP

LV

LV precedente  
PC precedente  
Variabili  
locali del  
chiamante  
Par. del chiamante  
Link PTR

# INVOKEVIRTUAL: esempio

# Codice Assembler Address

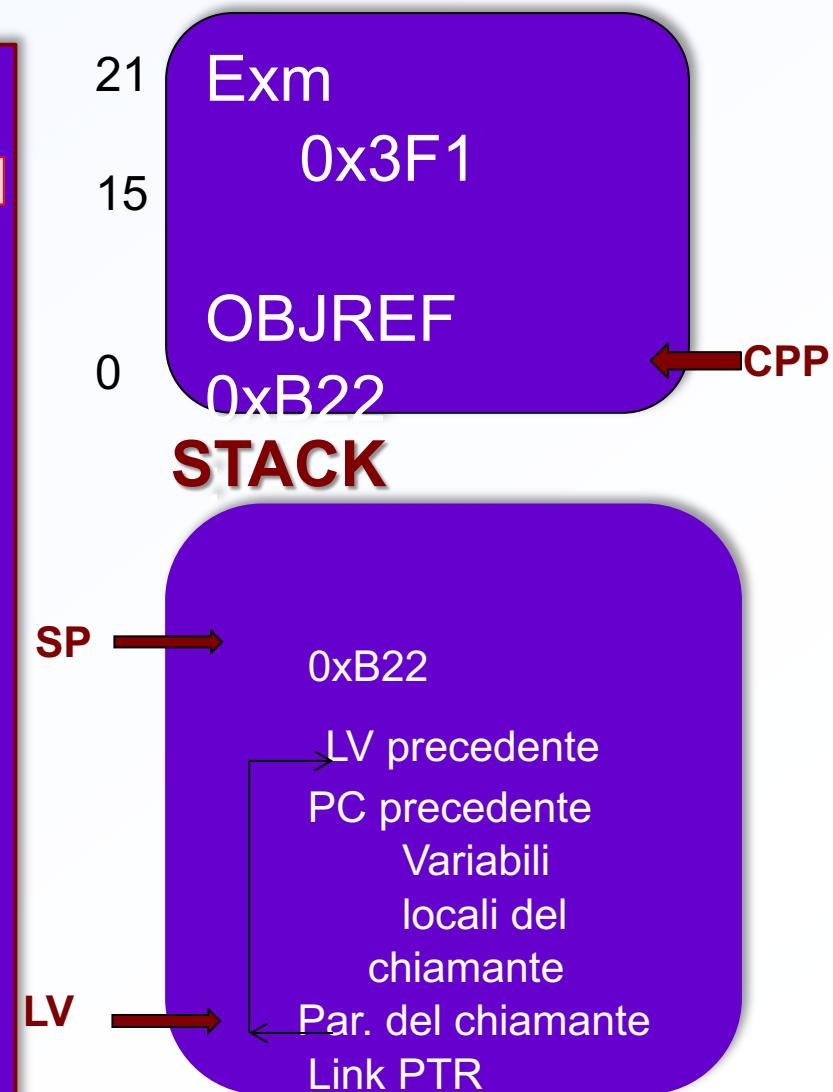
```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
    :
    :
```

```
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice esequibile

0x250			
0x251			
0x252	...		<- P
0x254	0x13	0x15	
0x256	0x10	0x05	
0x258	0x10	0x07	
0x25A	0xB6	0x21	
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮
0x3F1			
0x3F5	0x00	0x03	0x00
0x3F7	0x15	0x01	
0x3F9	0x15	0x02	
0x3FA	0x60		
0x3FB	0x59		
0x3FC	0x60		
	0xAC		

# Constant Pool



# INVOKEVIRTUAL: esempio

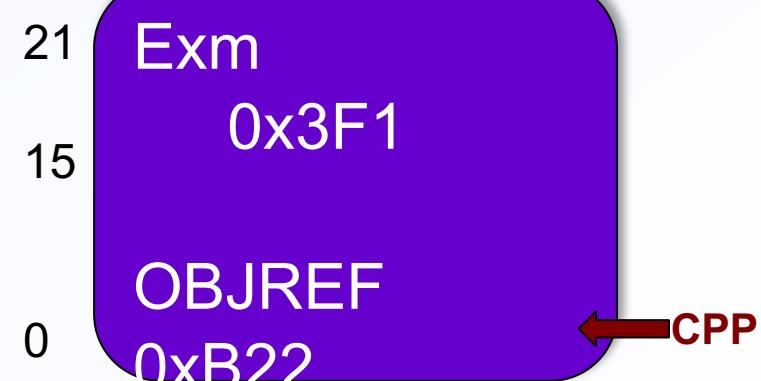
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

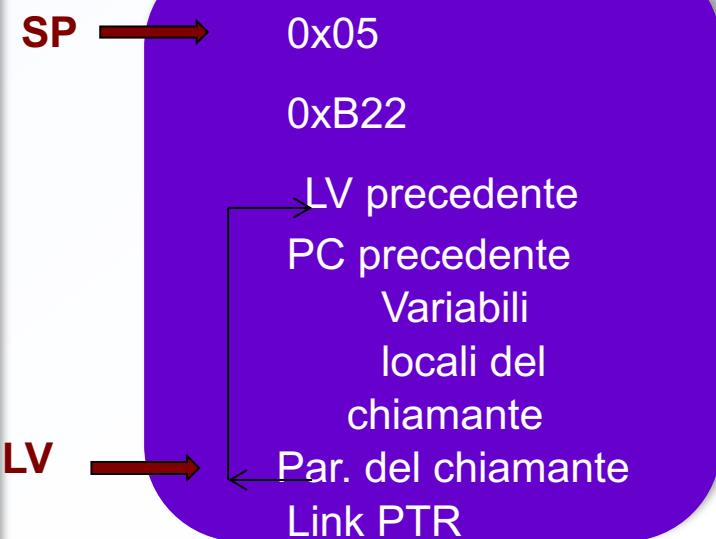
## Area codice eseguibile

0x250	
0x251	
0x252	
0x254	0x13 0x15 <- PC
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21
:	:
:	:
:	:
0x3F1	
0x3F5	0x00 0x03 0x00 0x00
0x3F7	0x15 0x01
0x3F9	0x15 0x02
0x3FA	0x60
0x3FB	0x59
0x3FC	0x60
	0xAC
	:

## Constant Pool



## STACK



# INVOKEVIRTUAL: esempio

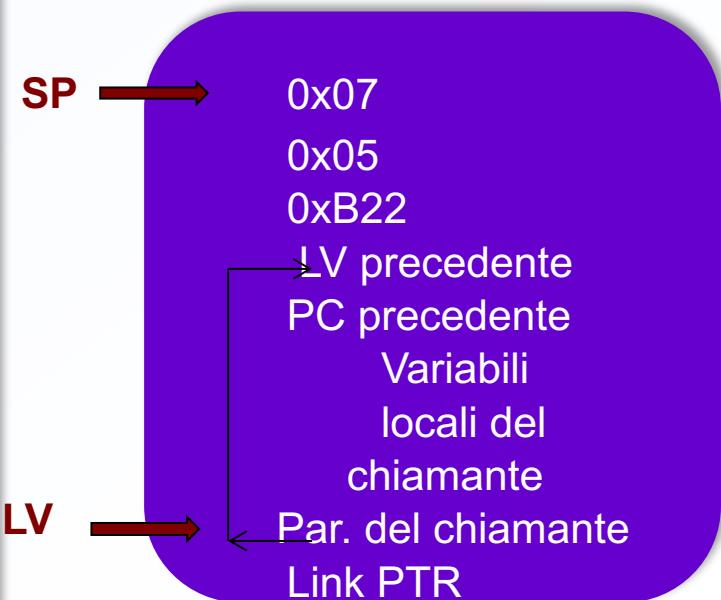
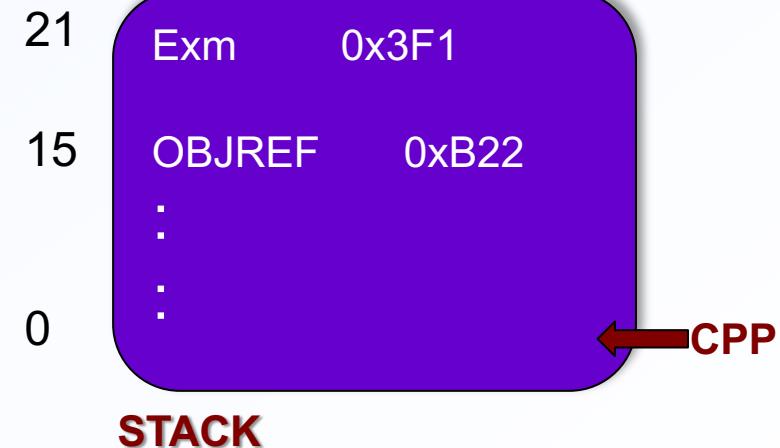
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	
0x251	
0x252	
0x254	
0x256	0x10 0x05 <- PC
0x258	0x10 0x07
0x25A	0xB6 0x21
:	
:	
:	
0x3F1	
0x3F5	0x00 0x03 0x00 0x00
0x3F7	0x15 0x01
0x3F9	0x15 0x02
0x3FA	0x60
0x3FB	0x59
0x3FC	0x60
	0xAC
	:

## Constant Pool



# INVOKEVIRTUAL: esempio

## Codice Assembler Address

I primi 4 byte a partire da 0x3F1 servono a riservare memoria sullo stack.

I primi due byte indicano il numero di parametri (compreso OBJREF), gli altri due il numero di variabili locali.

0x250  
0x251  
0x252  
0x254  
0x256  
0x258  
0x25A  
:  
:  
:  
0x3F1  
0x3F5  
0x3F7  
0x3F9  
0x3FA  
0x3FB  
0x3FC

## Area codice eseguibile

...  
0x13 0x15  
0x10 0x05  
0x10 0x07  
0xB6 0x21  
:  
:  
:  
0x00 0x03 0x00 0x00  
0x15 0x01  
0x15 0x02  
0x60  
0x59  
0x60  
0xAC

21  
15  
0

## STACK

SP

0x07

0x05

0xB22

LV precedente

PC precedente

Variabili

locali del

chiamante

Par. del chiamante

Link PTR

LV

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

→

←

# INVOKEVIRTUAL: esempio

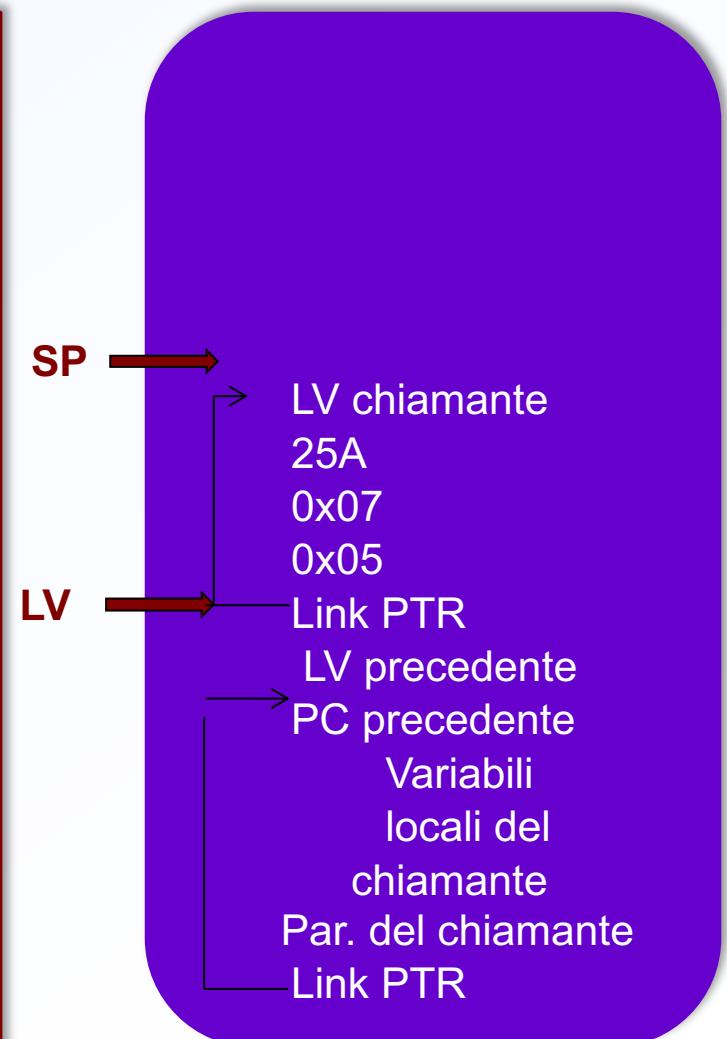
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	...
0x251	...
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21
:	:
:	:
:	:
0x3F1	0x00 0x03 0x00 0x00
0x3F5	0x15 0x01
0x3F7	0x15 0x02
0x3F9	0x60
0x3FA	0x59
0x3FB	0x60
0x3FC	0xAC
:	:

## STACK



# INVOKEVIRTUAL: esempio

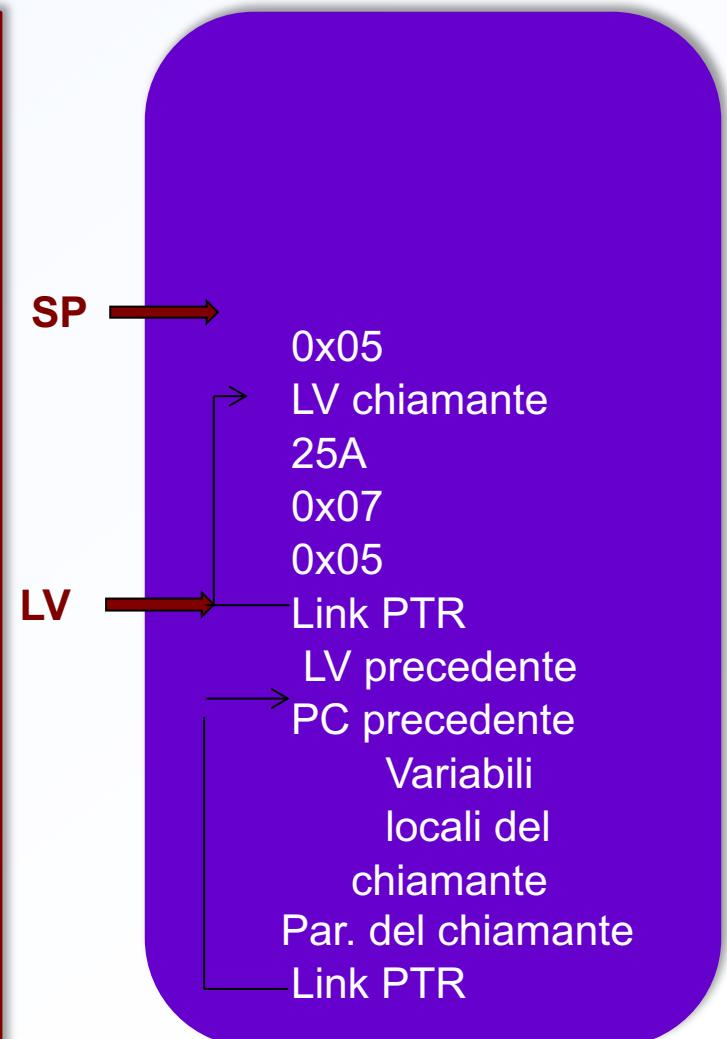
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	...
0x251	...
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21
:	:
:	:
:	:
0x3F1	0x00 0x03 0x00 0x00 <- PC
0x3F5	0x15 0x01
0x3F7	0x15 0x02
0x3F9	0x60
0x3FA	0x59
0x3FB	0x60
0x3FC	0xAC
:	:

## STACK



# INVOKEVIRTUAL: esempio

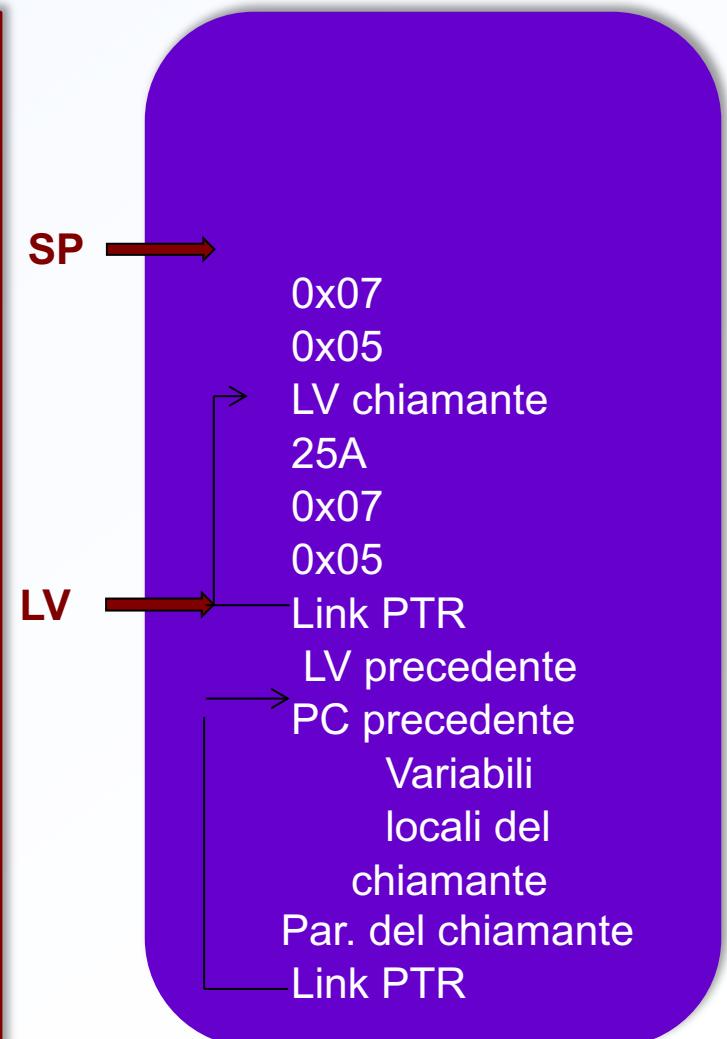
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	...
0x251	...
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21
:	:
:	:
:	:
0x3F1	0x00 0x03 0x00 0x00
0x3F5	0x15 0x01
0x3F7	0x15 0x01 <- PC
0x3F9	0x15 0x02
0x3FA	0x60
0x3FB	0x59
0x3FC	0x60
	0xAC
	:

## STACK



# INVOKEVIRTUAL: esempio

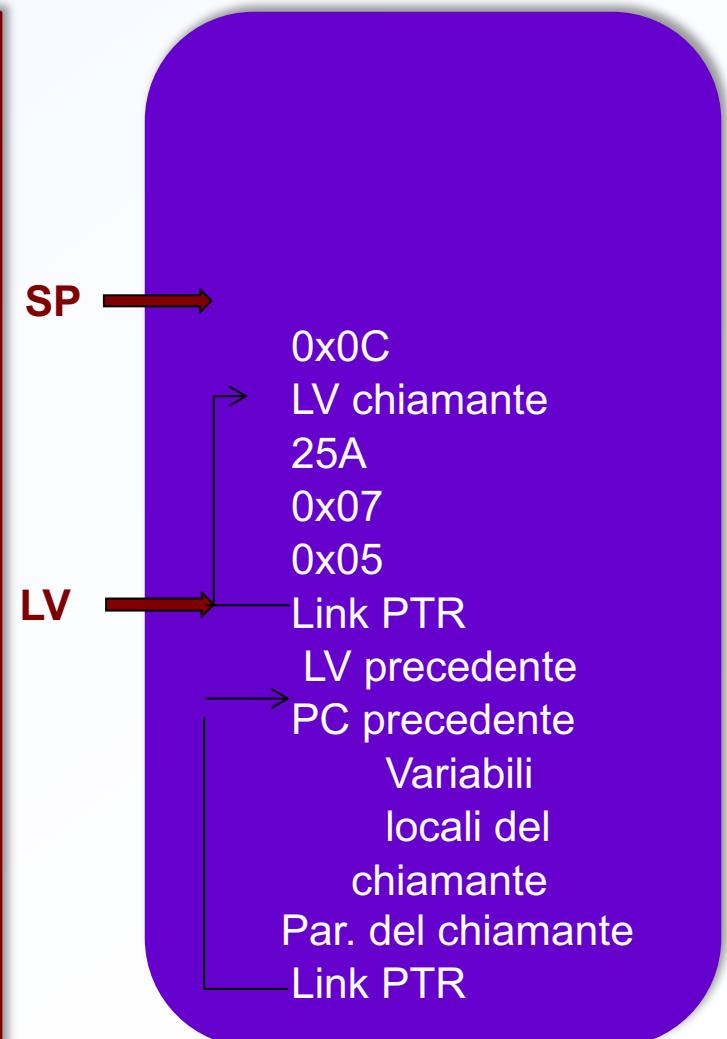
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	...
0x251	...
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21
:	:
:	:
:	:
0x3F1	0x00 0x03 0x00 0x00
0x3F5	0x15 0x01
0x3F7	0x15 0x02 <- PC
0x3F9	0x60
0x3FA	0x59
0x3FB	0x60
0x3FC	0xAC
:	:

## STACK



# INVOKEVIRTUAL: esempio

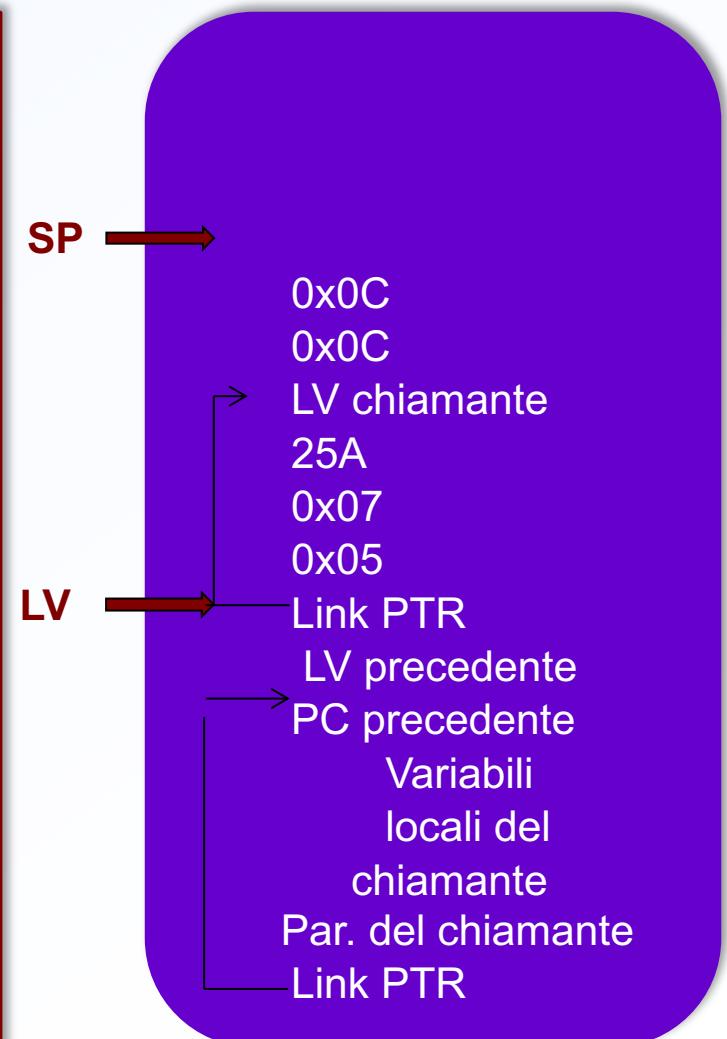
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	...
0x251	...
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21
:	:
:	:
:	:
0x3F1	0x00 0x03 0x00 0x00
0x3F5	0x15 0x01
0x3F7	0x15 0x02
0x3F9	0x60 <- PC
0x3FA	0x59
0x3FB	0x60
0x3FC	0xAC
:	:

## STACK



# INVOKEVIRTUAL: esempio

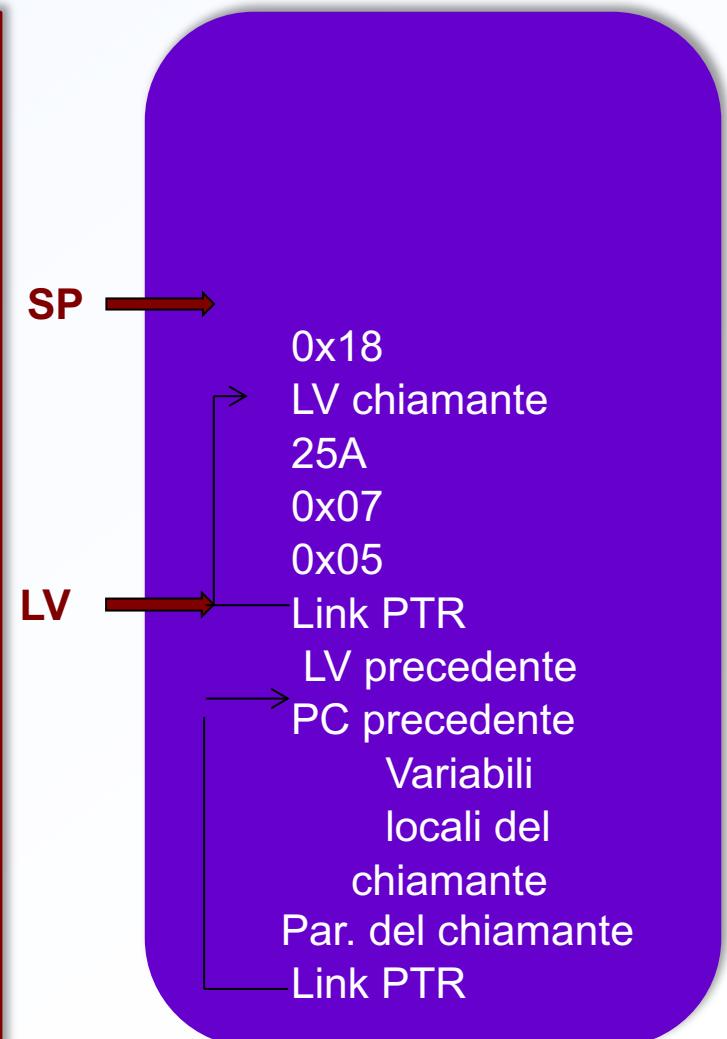
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	
0x251	
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21
:	:
:	:
:	:
0x3F1	0x00 0x03 0x00 0x00
0x3F5	0x15 0x01
0x3F7	0x15 0x02
0x3F9	0x60
0x3FA	0x59
0x3FB	0x60
0x3FC	0xAC

## STACK



# IRETURN: esempio

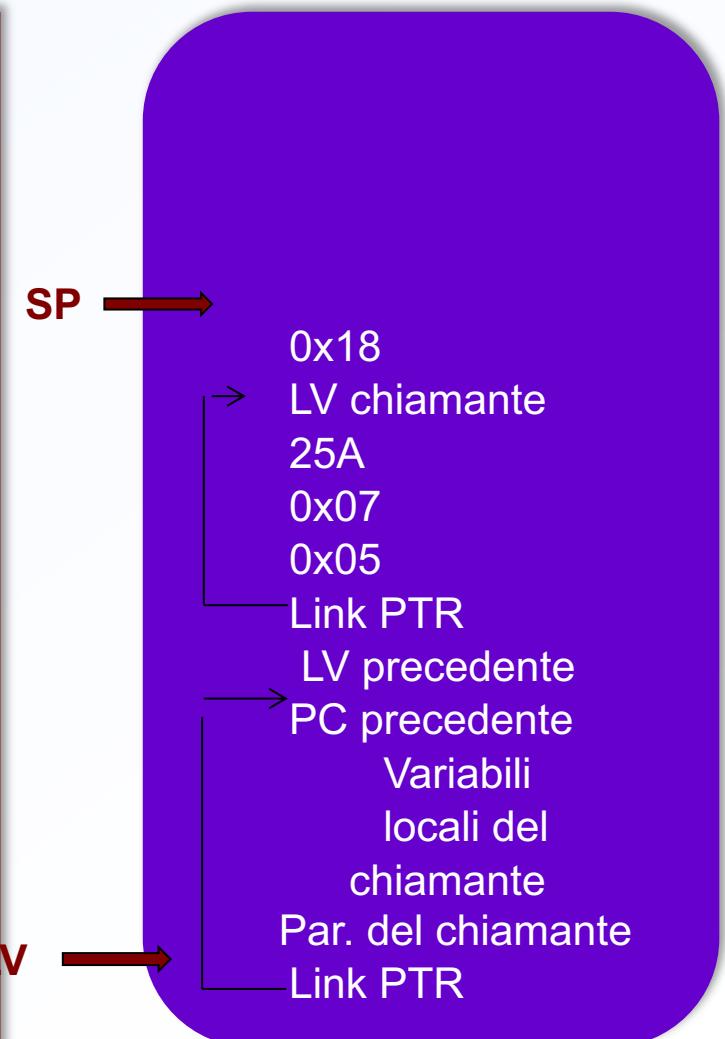
## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	...
0x251	...
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21
:	:
:	:
:	:
0x3F1	0x00 0x03 0x00 0x00
0x3F5	0x15 0x01
0x3F7	0x15 0x02
0x3F9	0x60
0x3FA	0x59
0x3FB	0x60
0x3FC	0xAC

## STACK



# IRETURN: esempio

## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	
0x251	
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21 <- PC
:	⋮
0x3F1	⋮
0x3F5	0x00 0x03 0x00 0x00
0x3F7	0x15 0x01
0x3F9	0x15 0x02
0x3FA	0x60
0x3FB	0x59
0x3FC	0x60
	0xAC

## STACK

SP

LV

0x18  
→ LV chiamante  
25A  
0x07  
0x05  
Link PTR  
→ LV precedente  
PC precedente  
Variabili  
locali del  
chiamante  
Par. del chiamante  
Link PTR

# IRETURN: esempio

## Codice Assembler Address

```
.main
...
LDCW objref
BIPUSH 5
BIPUSH 7
INVOKEVIRTUAL exm
:
:
method exm (J,K)
ILOAD J
ILOAD K
IADD
DUP
IADD
IRETURN
.end method
```

## Area codice eseguibile

0x250	
0x251	
0x252	...
0x254	0x13 0x15
0x256	0x10 0x05
0x258	0x10 0x07
0x25A	0xB6 0x21 <- PC
:	⋮
0x3F1	⋮
0x3F5	0x00 0x03 0x00 0x00
0x3F7	0x15 0x01
0x3F9	0x15 0x02
0x3FA	0x60
0x3FB	0x59
0x3FC	0x60
	0xAC
	⋮

## STACK

