

Architettura degli Elaboratori – Corso B

Turno di Laboratorio 2

Docente: Claudio Schifanella

Quarta Lezione

IJVM

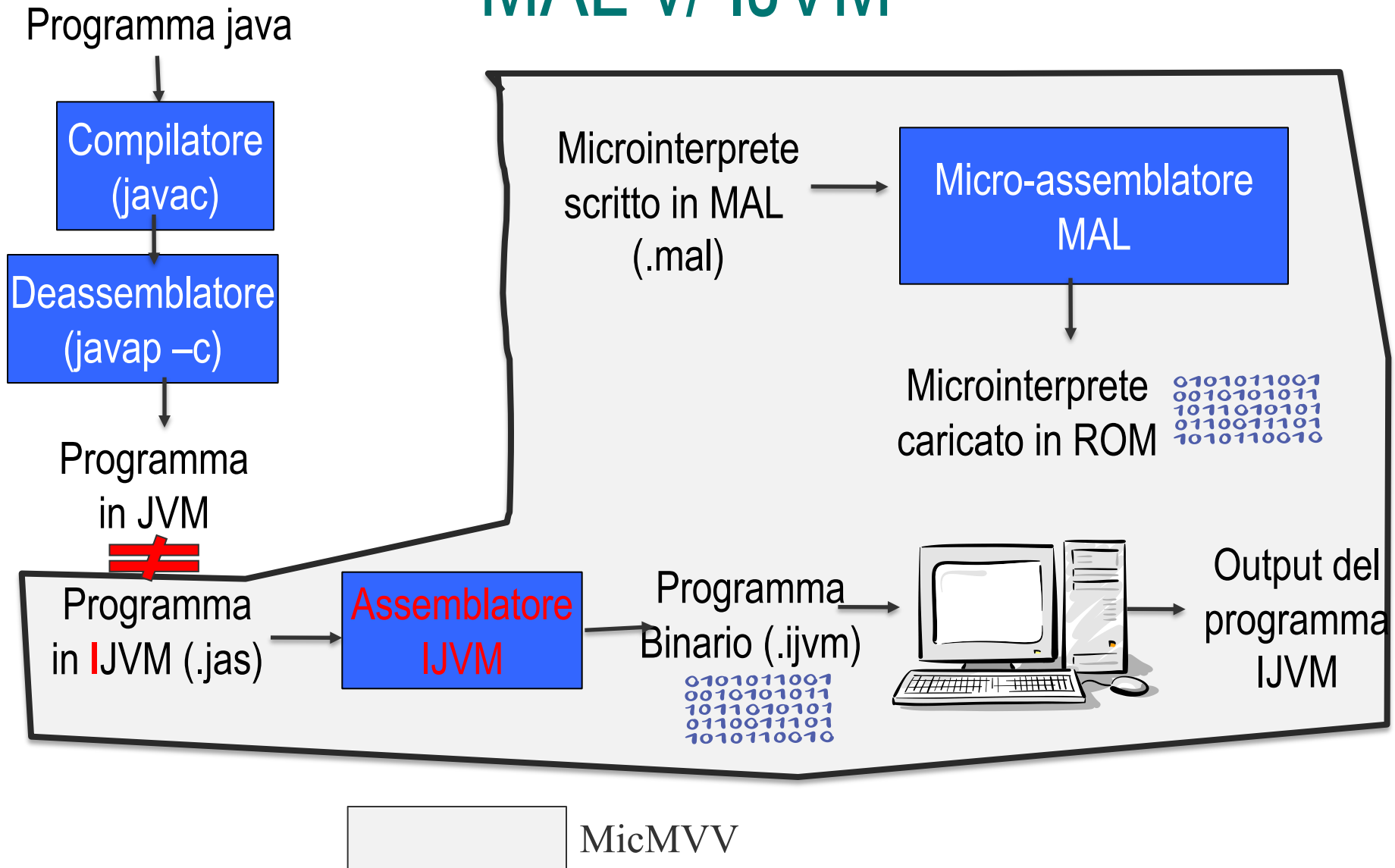
Simulatore Mic1MMV

- Che cos'è ?
- Un simulatore interattivo dell'architettura MIC-1 vista a lezione (Tanenbaum - cap.4)
- Si può utilizzare a diverse velocità (cioè livelli di granularità)
- È implementato in Java

Simulatore Mic1MMV – Funzionalità

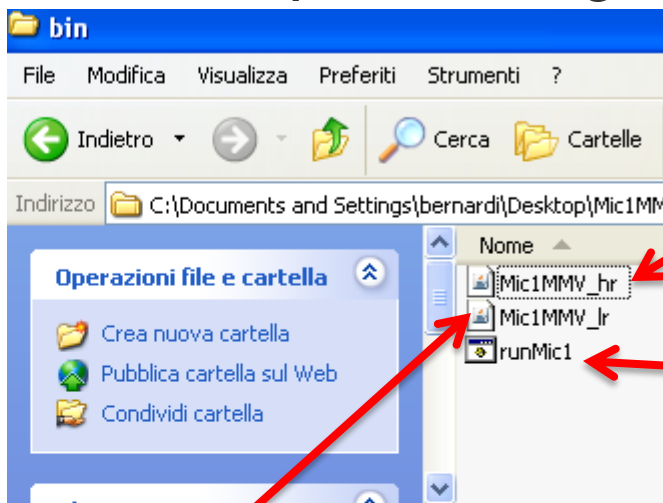
- Interpreta le istruzioni IJVM
- Assembla programmi scritti in IJVM (file .jas compilati in file .ijvm)
- Assembla programmi scritti in MAL (file .mal compilati in file .mic)

MAL v/ IJVM



Simulatore Mic1MMV – Installazione

- Scaricare il file Mic1MMV.zip da Moodle
- Decomprimerlo, gli eseguibili in: ./Mic1MMV/bin

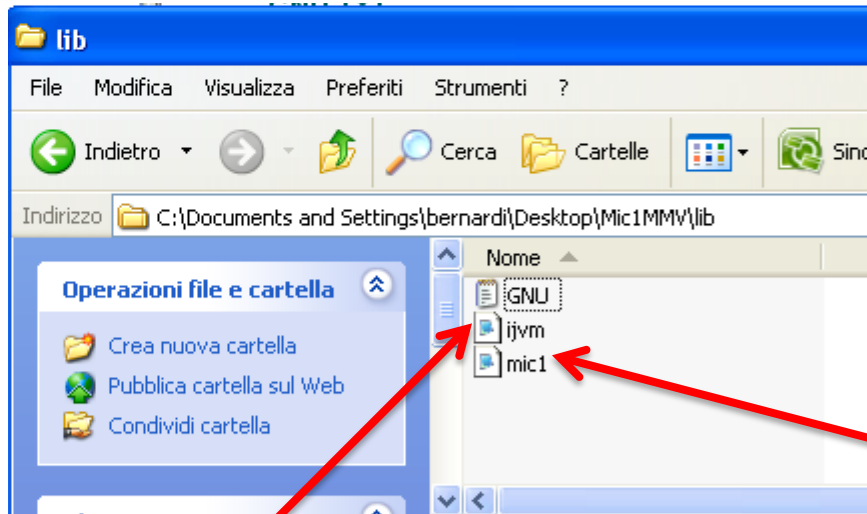


Mic1MMV_hr.jar schermi ad alta risoluzione (da 1280 x 960)

In alternativa: **runMic1.bat**
batch file da editare e usare per lanciare il simulatore.

Mic1MMV_lr.jar schermi a bassa risoluzione

Simulatore Mic1MMV – Lib

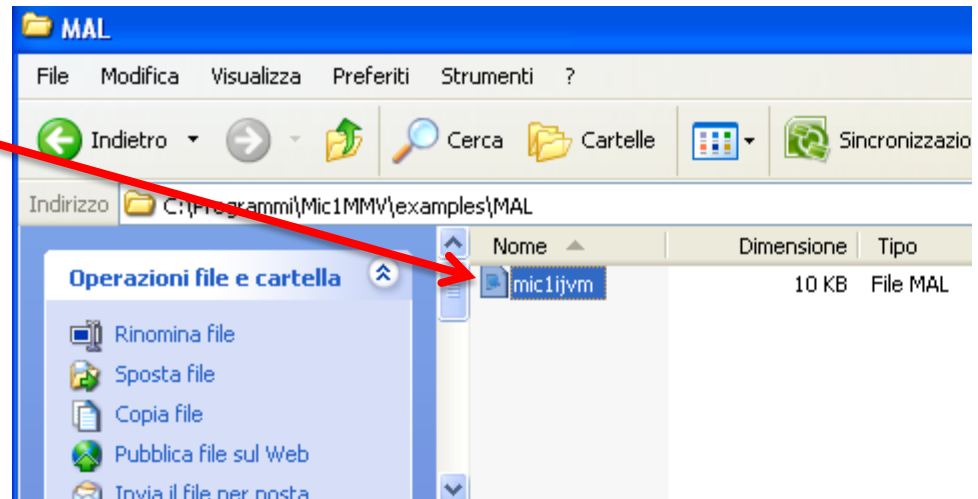
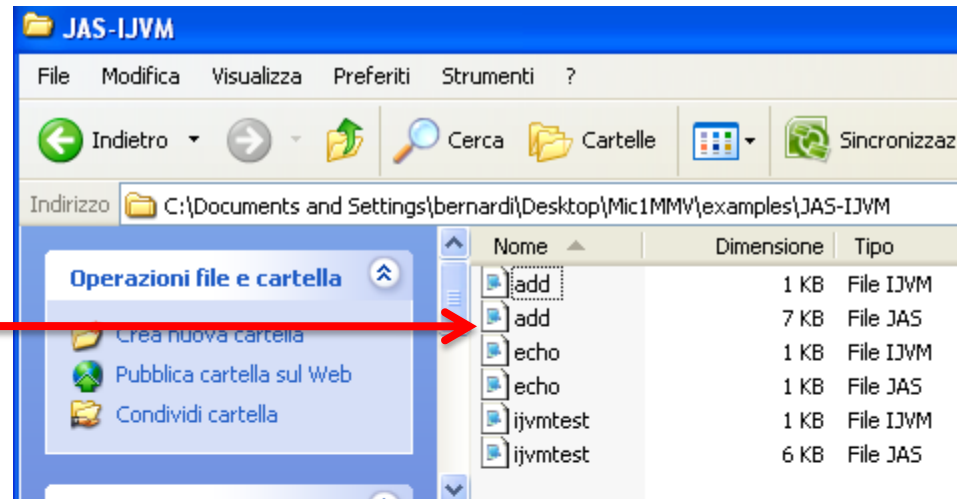


mic1.properties file per impostare le proprietà di default

ijvm.conf File di configurazione per l'assemblatore ijvmasm. Descrive il linguaggio IJVM

Simulatore Mic1MMV – Esempi

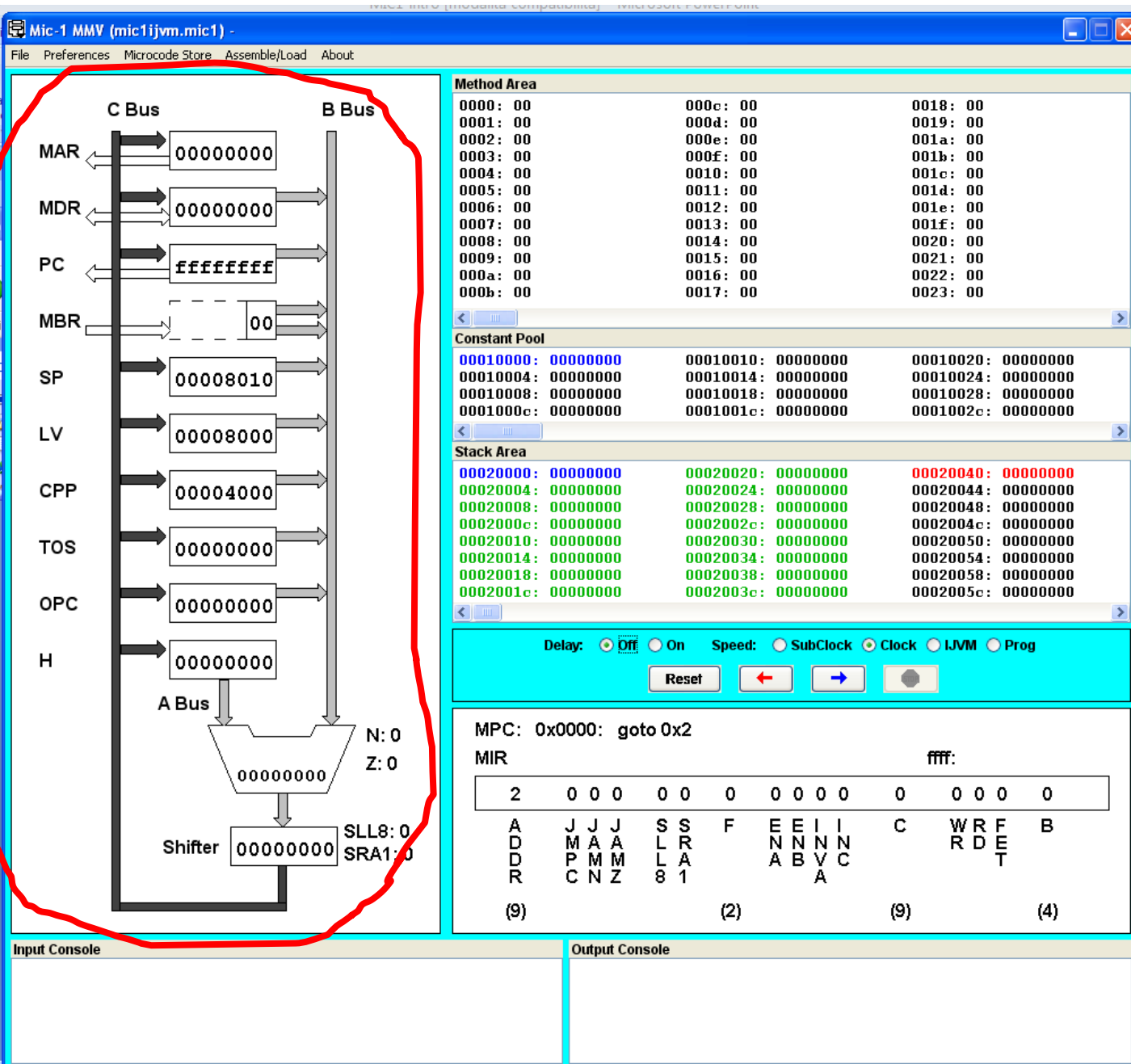
- **JAS-IJVM**: esempi programmi IJVM in codice sorgente (.jas) e assemblato (.ijvm)
- **MAL**: contiene il microinterprete di IJVM scritto in MAL (**mic1ijvm.mal**).



Simulatore Mic1MMV – Altro

- /doc: documentazione
- /src: sorgente simulatore

Datapath



Area
Metodi

Pool
costante

Stack

Console
comandi



MIR

Console
output

Console
input

Console comandi



- **Delay** ON=animazione datapath
- **Speed** permette di simulare a diversi livelli di granularità
 - **SubClock** simulazione per sottocicli di macchina (0-1-2-3). Ciascun passo esegue un sottociclo. Il datapath è animato sia con Delay=ON che Delay=OFF
 - **Clock** simulazione per cicli di clock. Ciascun passo esegue un ciclo di macchina.
 - **IJVM** simulazione di una istruzione IJVM.
 - **Prog** simulazione programma intero, senza interruzioni fino a fine progr.
- **Buttons**
 - **Reset** per tornare allo stato iniziale
 -  esecuzione di un passo all' indietro
 -  esecuzione di un passo in avanti
 - **Stop** stop simulazione

File JAS: template

- Costanti:

- si dichiarano globalmente
- sintassi:
`.constant`
`constant1 value1`
`constant2 value2`
`.end-constant`
- si riferenziano per nome con `LDC_W`

- Variabili:

- solamente locali
- si possono dichiarare solo in `.main` o in ciascun `.method`
- sintassi:
`.var`
`var1`
`var2`
`.end-var`
- si riferenziano per nome con `ILOAD`, `ISTORE`

- Etichette:

- marcano punti del programma a cui si può saltare;
- sono accessibili solo all'interno del metodo in cui appaiono
- esempio:

```
IFLT lt_max  
GOTO gte_max  
lt_max:    HALT  
gte_max:    
  
ERR
```

- Main

- esattamente uno per programma
- sintassi
`.main`
`-- variable declaration --`
`-- program contents --`
`.end-main`

File JAS: template

- Metodi:
 - sintassi:

```
.method method_name(par1,par2,...)
-- variable declaration --
-- method contents --
.end-method
```
 - il nome del metodo dichiarato si aggiunge alla tabella globale delle costanti
 - e quindi può essere referenziato per nome con `INVOKEVIRTUAL`
 - l'invocazione di un metodo richiede le seguenti azioni:

```
PUSH objref                (ignorato, ma necessario per compatibilità)
PUSH par1
PUSH par2
...
INVOKEVIRTUAL method_name
```
 - i parametri sono automaticamente aggiunti alle variabili locali del metodo e possono essere referenziate con `ILOAD par1`
 - i metodi devono essere dichiarati dopo il main;

Istruzioni per l'assemblatore IJVM

- **Nel file jas potete usare:**

- Mnemonici per le variabili locali
- Etichette per individuare i punti di arrivo dei salti
- Nomi per i metodi

- **Costanti:**

.constant

objref 0xCAFE // may be any value. Needed by invokevirtual.

my_max 100

.end-constant

- **Blocchi main e relative variabili:**

.main

.var

x

y

.end-var

Istruzioni del main

.end-main

Istruzioni per l'assemblatore IJVM

- **Blocchi metodi e relative variabili:**

- .method nomemetodo (par1, ..,parn)

- .var

- x

- .end-var

- Istruzioni del main*

- .end-method

I metodi terminano con un'istruzione di IRETURN, mentre il main termina normalmente con HALT

Esercizio 2: Caricare ed eseguire un progr. IJVM

1. Lanciare il simulatore
2. Selezionare dalla barra menu **File | Load IJVM program**
3. Navigare a "examples/JAS-IJVM"
4. Scegliere **ijvmtest.ijvm**. Il programma IJVM appare nell'area dei metodi
5. Selezionare la velocita' **Prog** nella console comandi
6. Clickare la freccia blu nella console comandi per eseguire il programma con il micro-programma. Alla fine exec. si legge nell'area di testo della console il messaggio **OK**.

Esercizio 3: Assemblare un programma IJVM

1. Scaricare dalla pagina del corso il file **Calcola2.jas**.
2. Selezionare dalla barra menu **File | Assemble/Load jas file**
3. Selezionare **Calcola2.jas**. Appare una finestra *Assembling Calcola2.jas ...* Quando finisce il processo di assemblaggio (creazione Calcola2.ijvm) clickare il bottone **Load**.
4. Il programma viene caricato e appare nell'area dei metodi.
5. Visualizzare il microcodice selezionando dalla barra menu **MicrostoreCode/ViewMicrostore**

...simulazione

Esercizi

- Per ciascuno dei seguenti frammenti di codice:
 1. Scrivere la sequenza di istruzioni JVM corrispondente.
 2. Contare il numero di byte occupati da tale frammento di codice JVM.
 3. Scrivere un semplice programma Java che includa tale insieme di istruzioni, compilarlo (`javac progr.java`) e generare il codice JVM corrispondente utilizzando il decompilatore Java (`javap -c progr_compilato`).
 4. Confrontare il codice JVM generato al punto 3) con quello scritto al punto 1) e verificare se ci sono differenze.

NOTA: Per risolvere gli esercizi seguenti (scrittura di programmi JVM, compilazione/deassemblaggio con `javac/javap -c` a partire da frammenti di codice java), utilizzare i file **template.java** e **template.jas**

Esercizio 4

- Frammento di codice Java:

```
if (x == 0)
    x = y;
else
    x = z;
y = x + 1;
```

Assumere che le variabili x,y,z siano memorizzate nelle posizioni 1, 2 e 3 del blocco delle variabili locali.

Esercizio 5

- Frammento di codice Java:

```
x = x - y;  
if (x < 0)  
    x = 0;  
y = y - 1;
```

dove le variabili x,y sono memorizzate nelle posizioni 1 e 2, rispettivamente, del blocco delle variabili locali.

Esercizio 6

- Frammento di codice Java:

```
x = (x - 2) + y;  
if (x < y)  
    x = x + y;
```

Assumere che le variabili x,y siano memorizzate nelle posizioni 1 e 2 del blocco delle variabili locali.

Esercizio 7

- Frammento di codice Java:

```
while(x < 0)
    x = x + y;
```

Assumere che le variabili x,y siano memorizzate nelle posizioni 1 e 2 del blocco delle variabili locali.

Esercizio 8

- Scrivere un frammento di codice JVM che calcola il quadrato di un numero intero positivo.