Architettura degli Elaboratori 2

Esercizi di gruppo validi come esonero per la parte pratica dell’esame

ESERCIZIO 3: Analisi del comportamento di metodi ricorsivi

INTRODUZIONE

L’INVOKEVIRTUAL è un’istruzione su 3 byte: il cui codice (relativo al nostro microprogramma) è 0xB6, gli altri due byte insieme formano un offset che indica in quale word, a partire dalla **CPP**, è salvato l’indirizzo di base del metodo da richiamare nell’area dei metodi. Il byte a cui punta **CPP**+(offset x 4) e quello successivo compongono il numero di parametri + 1 (**OBJREF**) e i due successivi il numero di variabili. Dopo questi primi quattro byte dalla base del metodo nell’area dei metodi sarà presente la prima microistruzione da eseguire. Questa istruzione prima di essere eseguita ha bisogni di parametri addizionali sulla cima dello stack, ovvero i parametri del metodo richiamato preceduti da un valore chiamato **OBJREF**.

(0x4000(CPP) + 0x08) << 2 = 0x10020

|  |  |  |  |
| --- | --- | --- | --- |
| Byte Address (HEX) | Content  (HEX) | Mnemonic | Labels |
| 24 | … | … | … |
| 25 | b6 | invokevirtual |  |
| 26 | 0 | torrihanoi\_1 |  |
| 27 | 8 | torrihanoi\_2 |  |
| 28 | … | … | … |
| … | … | … | … |
| 2f | … | … | … |
| 30 | 0 | #param\_1 |  |
| 31 | 4 | #param\_2 |  |
| 32 | 0 | #var\_1 |  |
| 33 | 1 | #var\_2 |  |
| 34 | 15 | iload |  |
| 35 | … | … | … |
| … | … | … | … |
| 10001f | … | … | … |
| 10020 | 0 | torrihanoi\_1 |  |
| 10021 | 0 | torrihanoi\_2 |  |
| 10022 | 0 | torrihanoi\_3 |  |
| 10023 | 30 | torrihanoi\_4 |  |

(Img. 1: Rappresentazione di parte dell’area dei metodi e Constants Pool corrispettiva alla compilazione del file *TorriHanoi.jas*)

Ad ogni chiamata della INVOKEVIRTUAL viene creato sullo stack un nuovo record di attivazione: ovvero uno spazio in memoria per i parametri e le variabili locali del metodo chiamato (dato dai primi quattro byte alla base del metodo nell’area dei metodi). Inoltre saranno anche presenti i valori di **LV** e **PC** al momento della call per riuscire a ripristinarli all’esecuzione della IRETURN.

Alla base del record di attivazione (**LV**) si troverà il link pointer che sostituirà esattamente **OBJREF** nello stack, in esso verrà salvato l’indirizzo di una cella all’interno del record in cui è salvato il valore di **PC** da assegnare alla fine dell’esecuzione del metodo. Successivamente troviamo i parametri, che si troveranno già sullo stack, e le variabili. In seguito troveremo il valore di **PC** (mem[link pointer])ed **LV** per l’esecuzione della IRETURN. A questo punto lo **SP** punterà all’indirizzo successivo a questi ultimi.

|  |  |  |
| --- | --- | --- |
| Word Address (HEX) | Content(hex) | Pointers |
| … | … | … |
| 801c | 0000006e | #4 |
| 801d | 0000800f | #4 |
| **801e** | 00008023 | #5 |
| 801f | 00000003 | #5 |
| 8020 | 00000002 | #5 |
| 8021 | 00000001 | #5 |
| 8022 | 00000003 | #5 |
| 8023 | 00000054 | #5 |
| 8024 | 00008017 | #5 |
| 8025 | **0000802a** | <-- LV #6 |
| 8026 | **00000002** | #6 |
| 8027 | **00000002** | #6 |
| 8028 | **00000003** | #6 |
| 8029 | **00000001** | #6 |
| **802a** | **00000054** | #6 |
| 802b | **0000801e** | #6 |
| 802c | 00000002 | <-- SP #6 |

|  |  |  |  |
| --- | --- | --- | --- |
| Byte Address (HEX) | Content  (HEX) | Mnemonic | Labels |
| … | … | … | … |
| 51 | b6 | invokevirtual |  |
| 52 | 0 | torrihanoi\_1 |  |
| 53 | 8 | torrihanoi\_2 |  |
| **54** | 13 | ldc\_w |  |

(Img. 2: Rappresentazione di parte della RAM e area dei metodi corrispettiva al sesto livello di chiamata ricorsiva alla funzione torriHanoi(n,a,b) del file *torriHanoi.jas*)

IL METODO ASSEGNATO

Il metodo è torriHanoi(n,a,b) (line 45, file *TorriHanoi.jas*). La prima istruzione si trova codificata all’indirizzo 0x34 dell’area dei metodi. L’indirizzo 0x30 fa sempre parte del metodo ma serve solo alla INVOKEVIRTUAL per poter inizializzare correttamente lo stack.

Esso viene sempre richiamato nella linea 34 del file che corrisponde all’indirizzo 0x25 dell’area dei metodi e potrebbe venir chiamato (non con una certezza assoluta) alle linee 67 e 82 del file che corrispondono agli indirizzi 0x51 e 0x6b. Questi ultimi fanno parte della ricorsione all’interno del metodo.

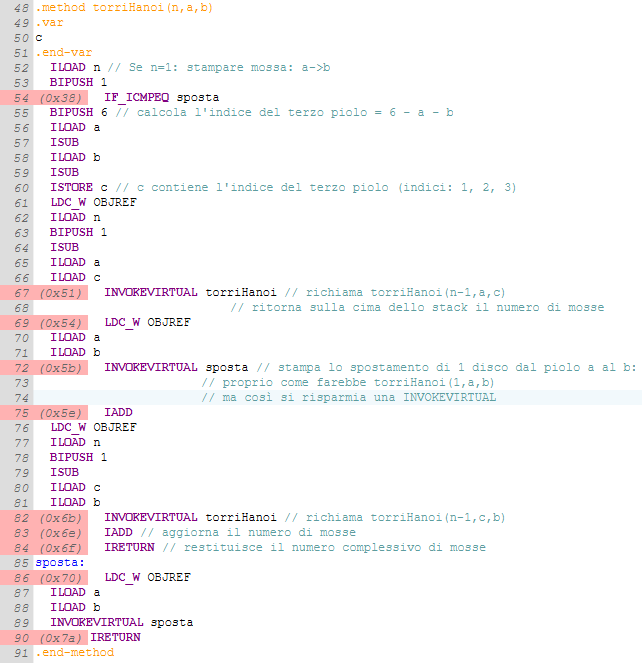
Nelle zone dell’area dei metodi, evidenziate in azzurro, troviamo: il caricamento sullo stack dei parametri e la chiamata all’INVOKEVIRTUAL con i successivi due byte che compongono l’offset dalla **CPP** (0x08).

|  |  |  |  |
| --- | --- | --- | --- |
| Byte Address (HEX) | Content  (HEX) | Mnemonic | Labels |
| 20 | … | … | … |
| 21 | 10 | bipush |  |
| 22 | 1 | byte |  |
| 23 | 10 | bipush |  |
| 24 | 3 | byte |  |
| 25 | b6 | invokevirtual |  |
| 26 | 0 | torrihanoi\_1 |  |
| 27 | 8 | torrihanoi\_2 |  |
| 28 | … | … | … |
| … | … | … | … |
| 30 | 0 | #param\_1 |  |
| 31 | 4 | #param\_2 |  |
| 32 | 0 | #var\_1 |  |
| 33 | 1 | #var\_2 |  |
| 34 | 15 | iload |  |
| 35 | 1 | n |  |
| 36 | 10 | bipush |  |
| 37 | 1 | byte |  |
| 38 | 9f | if\_icmpeq |  |
| 39 | 0 | sposta\_1 |  |
| 3a | 38 | sposta\_2 |  |
| 3b | 10 | bipush |  |
| 3c | 6 | byte |  |
| 3d | 15 | iload |  |
| 3e | 2 | a |  |
| 3f | 64 | isub |  |
| 40 | 15 | iload |  |
| 41 | 3 | b |  |
| 42 | 64 | isub |  |
| 43 | 36 | istore |  |
| 44 | 4 | c |  |
| 45 | 13 | ldc\_w |  |
| 46 | 0 | objref\_1 |  |
| 47 | 0 | objref\_2 |  |
| 48 | 15 | iload |  |
| 49 | 1 | n |  |
| 4a | 10 | bipush |  |
| 4b | 1 | byte |  |
| 4c | 64 | isub |  |
| 4d | 15 | iload |  |
| 4e | 2 | a |  |
| 4f | 15 | iload |  |
| 50 | 4 | c |  |
| 51 | b6 | invokevirtual |  |
| 52 | 0 | torrihanoi\_1 |  |
| 53 | 8 | torrihanoi\_2 |  |
| 54 | 13 | ldc\_w |  |
| 55 | 0 | objref\_1 |  |
| 56 | 0 | objref\_2 |  |
| 57 | 15 | iload |  |
| 58 | 2 | a |  |
| 59 | 15 | iload |  |
| 5a | 3 | b |  |
| 5b | b6 | invokevirtual |  |
| 5c | 0 | sposta\_1 |  |
| 5d | 9 | sposta\_2 |  |
| 5e | 60 | iadd |  |
| 5f | 13 | ldc\_w |  |
| 60 | 0 | objref\_1 |  |
| 61 | 0 | objref\_2 |  |
| 62 | 15 | iload |  |
| 63 | 1 | n |  |
| 64 | 10 | bipush |  |
| 65 | 1 | byte |  |
| 66 | 64 | isub |  |
| 67 | 15 | iload |  |
| 68 | 4 | c |  |
| 69 | 15 | iload |  |
| 6a | 3 | b |  |
| 6b | b6 | invokevirtual |  |
| 6c | 0 | torrihanoi\_1 |  |
| 6d | 8 | torrihanoi\_2 |  |
| 6e | 60 | iadd |  |
| 6f | ac | ireturn |  |
| 70 | 13 | ldc\_w | sposta |
| 71 | 0 | objref\_1 |  |
| 72 | 0 | objref\_2 |  |
| 73 | 15 | iload |  |
| 74 | 2 | a |  |
| 75 | 15 | iload |  |
| 76 | 3 | b |  |
| 77 | b6 | invokevirtual |  |
| 78 | 0 | sposta\_1 |  |
| 79 | 9 | sposta\_2 |  |
| 7a | ac | ireturn |  |
| 7b | … | … | … |

(Img. 3: Rappresentazione di parte dell’area dei metodi corrispettiva alla compilazione del file *TorriHanoi.jas*)

ANALISI DEL COMPORTAMENTO DEL METODO RICORSIVO

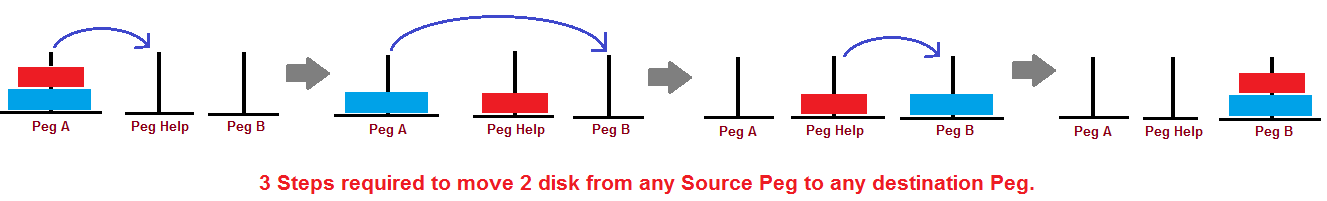
Per poter effettuare uno studio approfondito del comportamento del metodo ricorsivo abbiamo implementato un nuovo code editor con relativo Debugger. I breakpoint sono stati inseriti alla prima istruzione di salto in torriHanoi(n,a,b) [0x38], nei punti di ritorno [0x6f, 0x7a], ad ogni chiamata ricorsiva [0x51, 0x6b] ed ad ogni istruzione successiva ad esse [0x54, 0x6e], ad ogni call di sposta(i,j) [0x5b, 0x77] ed istruzione successiva [0x5e] in fine alla prima istruzione della label sposta [0x70]. Con questa impostazione abbiamo la possibilità di osservare lo stack subito prima di una chiamata ricorsiva e subito dopo.



**private static int** torriHanoi(**int** n, **int** a, **int** b) {  
 **int** c;  
 **int** numMosse = 0; *// variabile in stack*  
  
 **if** (n == 1) {  
 *// dato che 1 è l'unità di disco più piccola  
 // significa che non ci sono dischi spora  
 // di lui e che ovunque lo metta non potrà  
 // stare sopra un disco più piccolo di lui  
 // quindi posso semplicemente spostarlo sul  
 // piolo destinazione "b"* **return** *sposta*(a, b);  
 } **else** {  
 *// Determino quale sarà il piolo d'appoggio  
 // assegnandolo alla variabile "c"* c = 6 - a - b;  
  
 *// Chiamo la ricorsivamente torriHanoi per  
 // i dischi sovrastanti cercando di spostarli  
 // tutti sul piolo d'appoggio* numMosse = *torriHanoi*(n - 1, a, c);  
  
 *// a questo punto non ci sono dischi più piccoli  
 // di "n" sul piolo di partenza "a" quindi  
 // procedo a spostare il disco "n" sul  
 // piolo di destinazione "b"* numMosse += *sposta*(a, b);  
  
 *// richiamo torriHanoi per spostare tutti  
 // i dischi che ho posizionato sul piolo  
 // d'appoggio "c" al piolo di destinazione "b"* **return** (numMosse += *torriHanoi*(n - 1, c, b));  
 }  
}

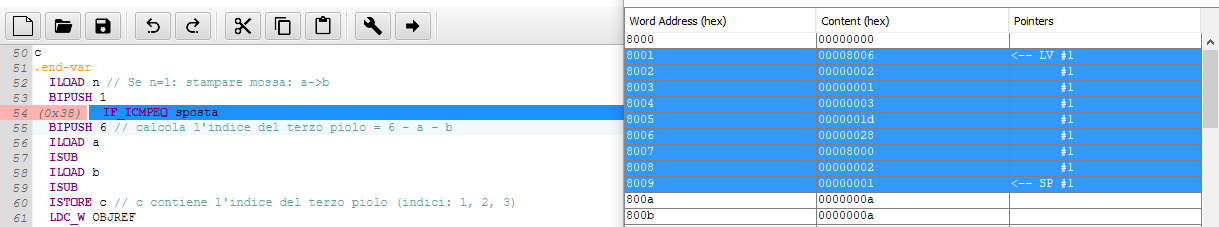
(Img. 4: A sinistra codifica in Java di torriHanoi(n,a,b) a destra la codifica in JAS all’interno del nuovo code editor con relativi breakpoints. Le linee colorate tratteggiate rappresentano il mapping tra i due linguaggi)

Per analizzare l’algoritmo abbiamo deciso di svolgere una torre di Hanoi con il più piccolo numero di dischi per avviare la ricorsione: 2.

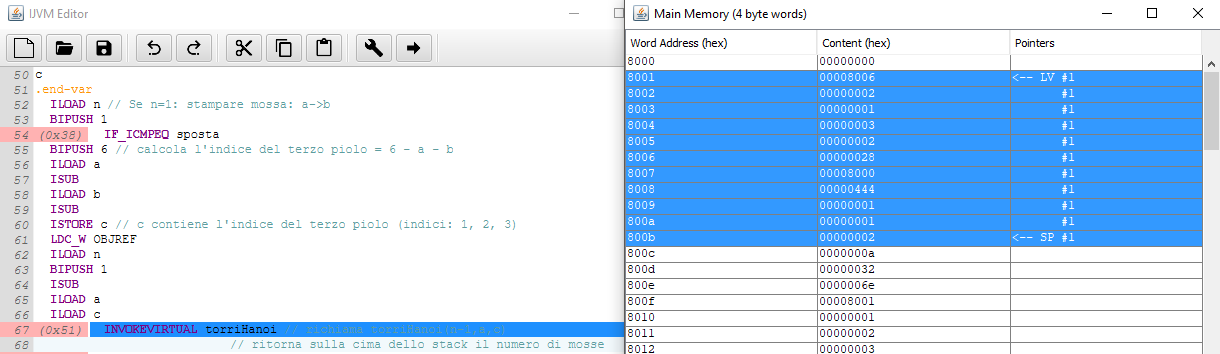


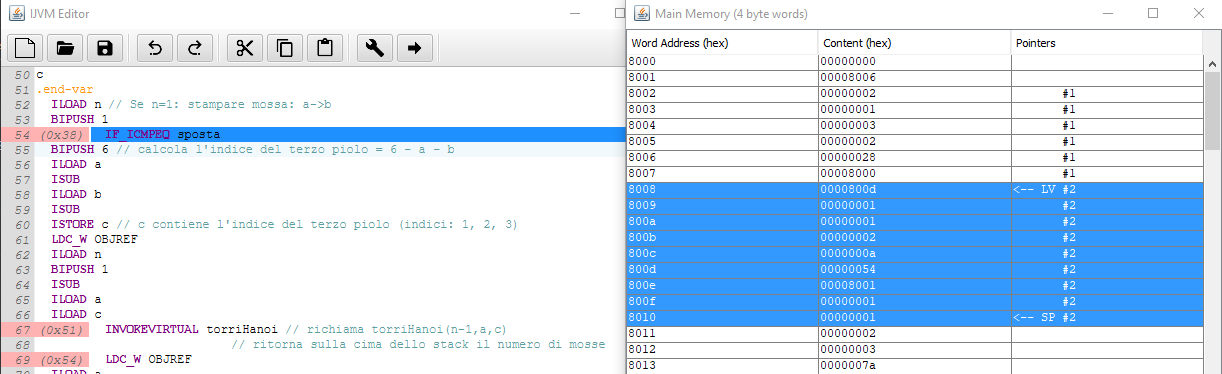
(Img. 5: Risoluzione della torre di Hanoi con due dischi)

In questo punto è stata richiamata per la prima volta dal main torriHanoi(n,a,b) con n = 0x2 (numero di dischi da spostare), a = 0x1 (piolo di partenza), b = 0x3 (piolo di destinazione). Tali valori si possono vedere sullo stack agli indirizzi 0x8002, 0x8003, 0x8004, successivamente all’indirizzo 0x8005 troviamo lo spazio per la variabile locale c sopra essa il **PC** di ritorno e dopo ancora l’**LV** del chiamante. In fine sulla cima dello stack troviamo prima il valore di n (caricato da ILOAD) e poi un 0x0001 dovuto alla BIPUSH e siamo pronti all’esecuzione della IF\_ICMPEQ che fallirà in quanto i due elementi sulla cima dello stack non sono uguali proseguendo quindi con il set di istruzioni subito sotto.

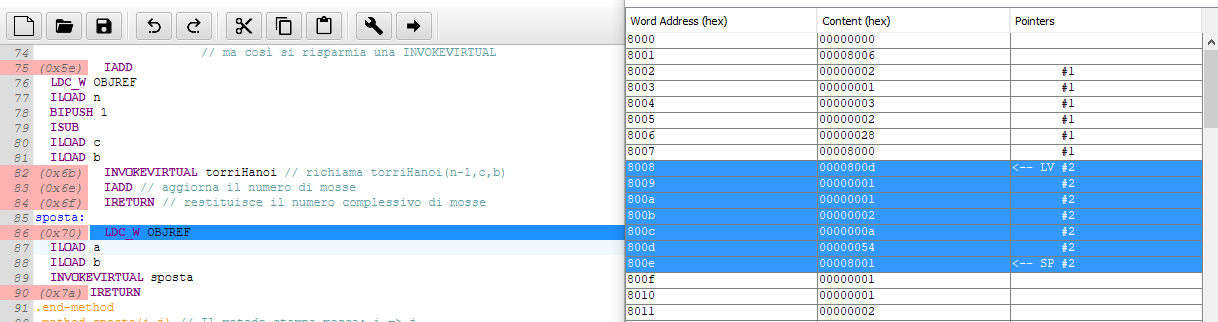


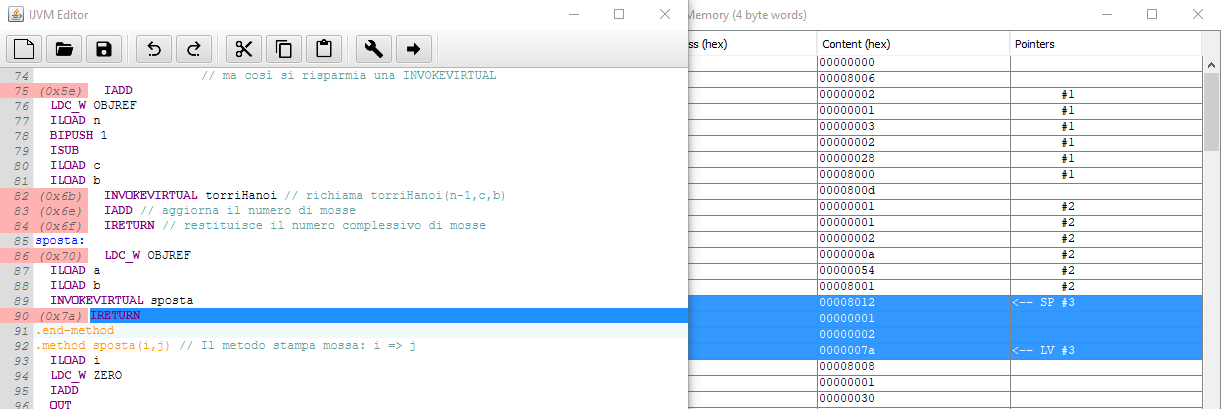
A questo punto il programma avrà determinato quale sarà il piolo d’appoggio e lo avrà salvato in c infatti all’indirizzo 0x8005 troviamo 0x2 ovvero 0x6–a–b = 0x6–0x1–0x3. Adesso chiameremo torriHanoi(n-1,a,c) proprio per spostare tutti i dischi più grandi di n sul piolo d’appoggio al fine di poter spostare il disco n sul piolo destinazione b. In cima allo stack troviamo (partendo da 0x8008): 0x0444 (OBJREF caricato tramite LDC\_W), 0x0001 (n-1), 0x0001 (a caricata tramite ILOAD), 0x0002 (c caricata tramite ILOAD).



Ci troviamo ora in un primo livello di ricorsione, infatti i record di attivazione sullo stack adesso sono due. Siamo nella stessa situazione della prima immagine solo che ora la IF\_ICMPEQ avrà successo portandoci alla label sposta.

Se ci troviamo in questo frammento di codice significa che ci troviamo nel caso “base” della torriHanoi ovvero un solo disco. In questo caso si procede solo con lo spostamento di un disco al piolo sorgente a al piolo destinazione b. Sulla console verrà ora stampato 1 => 2

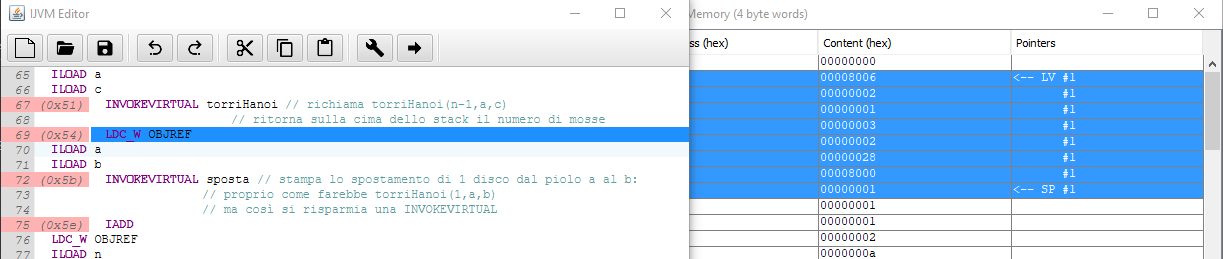


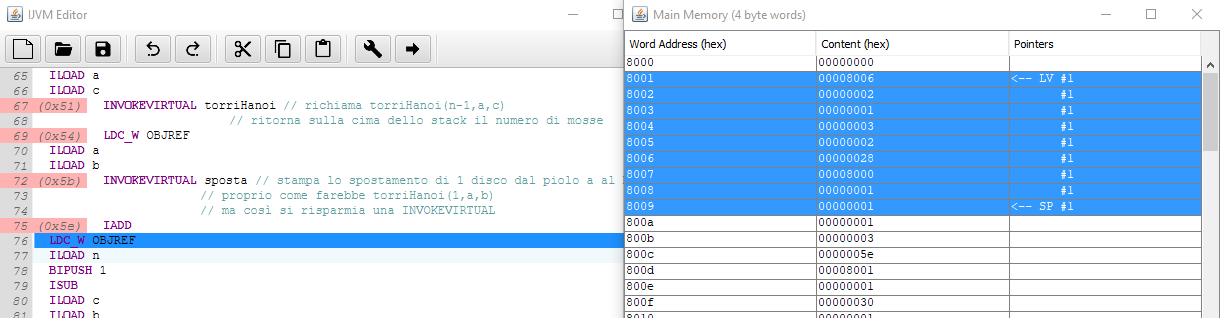


Invocando la IRETURN si ritorna al record precedente (#1) e si prosegue con l’istruzione successiva. Si vuol far notare che si riprende l’esecuzione da 0x54 ovvero il valore presente all’indirizzo 0x800d del record di attivazione precedente (#2) ovvero il suo **PC** di ritorno.

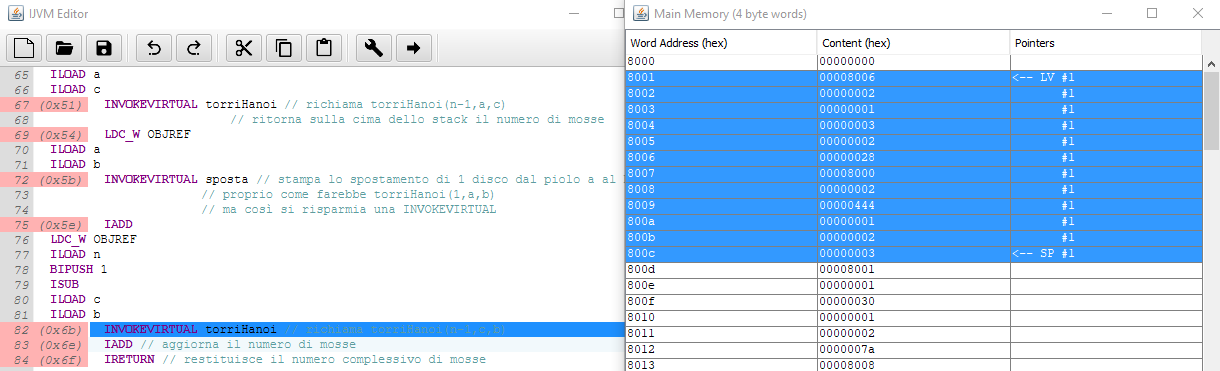
Presupponendo che torriHanoi(n-1,a,c) abbia spostato tutti i dischi più piccoli di n sul piolo d’appoggio c, procediamo con lo spostare il disco n sul piolo destinazione b stampando 1 => 3 sulla console. E successivamente sommiamo il numero di passaggi che ci sono voluti per eseguire torriHanoi(n-1,a,c) con sposta(a,b).

Nell’immagine seguente troviamo sulla cima dello stack il valore 0x1 che è il numero di mosse impiegate per eseguire torriHanoi(n-1,a,c). In quella dopo troviamo un altro 0x1 che è il numero di mosse impiegato per eseguire sposta(a,b) mentre l’elaboratore sta finendo di eseguire la IADD ed iniziando LDC\_W.

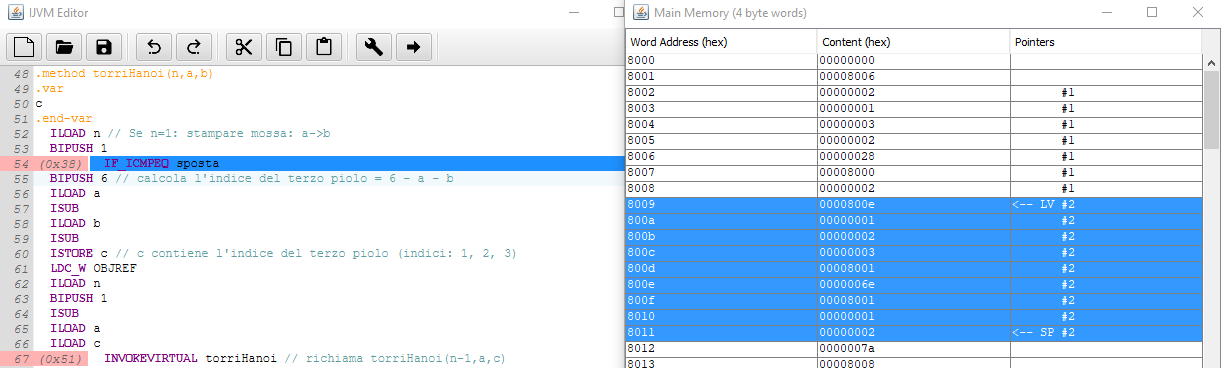


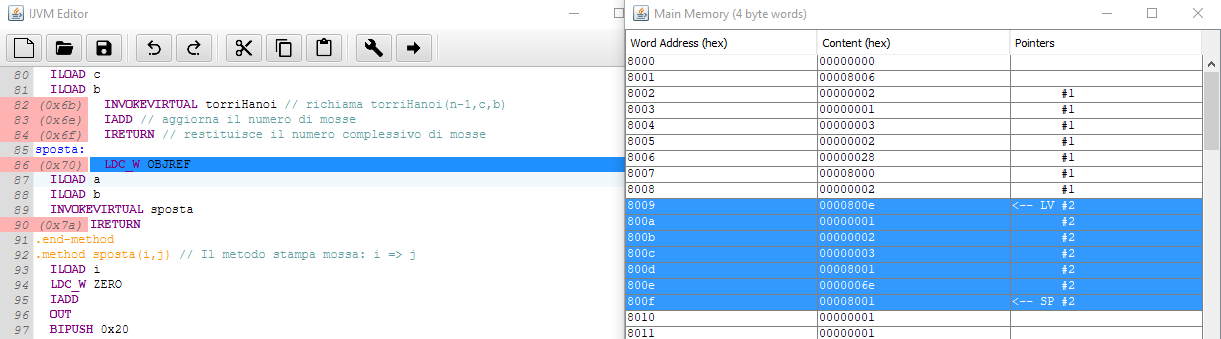


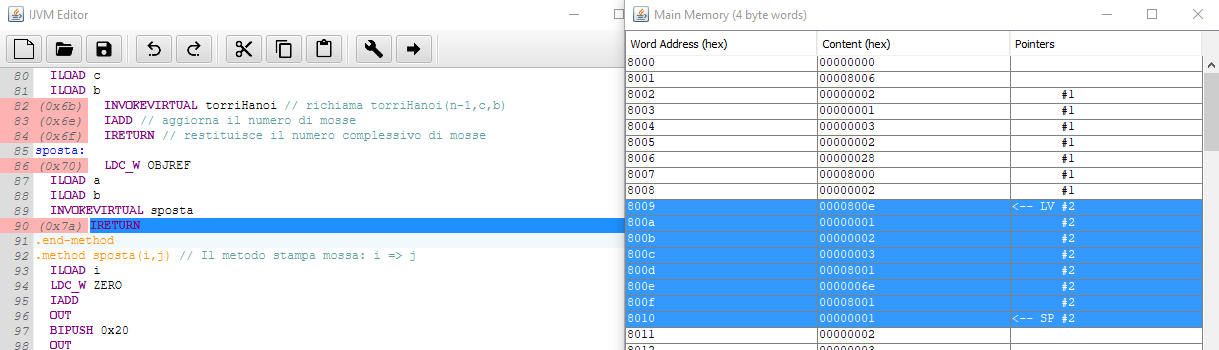
Dato che ora siamo riusciti a spostare il disco n al piolo destinazione b non ci resta che spostare tutti i dischi che avevamo messo sul piolo d’appoggio c sul piolo destinazione b richiamando torriHanoi(n-1,c,b). In cima allo stack troviamo (partendo da 0x8008): 0x0002 (il numero di passaggi impiegati fin ora per riuscire a spostare il disco n dal piolo a al piolo b), 0x0444 (OBJREF caricato tramite LDC\_W), 0x0001 (n-1), 0x0002 (c caricata tramite ILOAD), 0x0003 (b caricata tramite ILOAD).



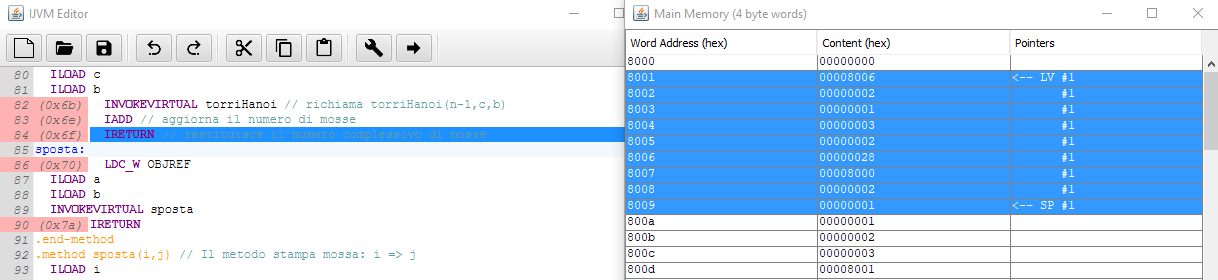
Con i tre fotogrammi sucessivi ci ritorviamo in una condizione precedentemente analizzata solo che il risultato finale sarà lo spostamento di un disco dal piolo 2 al piolo 3 stampando sulla console 1 => 3.



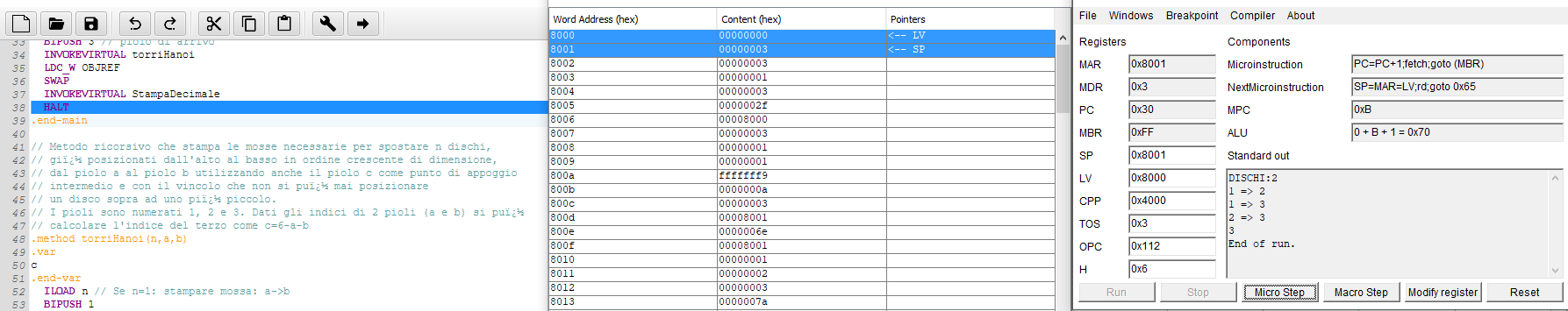




Una volta spostati tutti i dischi dal piolo d’appoggio a quello destinazione possiamo restituire il numero di passaggi impiegati. In cima allo stack troviamo (partendo da 0x8008): 0x0002 ovvero il numero di passaggi impiegati fin ora per riuscire a spostare il disco n dal piolo sorgente a al piolo destinazione b, 0x0001 ovvero il numero di passaggi impiegati fin ora per riuscire a spostare i dischi messi sul piolo d’appoggio c al piolo destinazione b. In quest’ultimo fotogramma l’elaboratore sta finendo di eseguire la IADD ed iniziando IRETURN.



Questa è la situazione finale al termine dell’esecuzione del programma con 2 dischi.



Il numero di record di attivazione creati da torriHanoi(n,a,b) è n+1 ovvero n chiamate ricorsive a torriHanoi(n,a,b) + una chiamata a sposta(a,b). Grazie a questa constatazione possiamo determinare qual è il numero massimo di dischi spostabili dall’algoritmo nel nostro elaboratore. Ogni record di torriHanoi(n,a,b) al momento della call ricorsiva occupa al massimo 8 word mentre, al massimo, la chiamata a sposta(a,b) occupa 6 word, allora il massimo spazio che può occupare sullo stack una chiamata con n dischi è word e, dato che, una word sono (in questa architettura) 4 byte alla fine lo spazio occupato è di 4byte. Nel nostro emulatore la dimensione totale dello stack è di 0x10000 word (256kB) e l’indirizzo di partenza è 0x8000 ovvero la metà lasciano a disposizione 0x8000 word (128kb) per allocare record di attivazione, quindi

oppure

Il numero intero più grande minore di 4096 è 4095 ovvero il massimo numero di dischi che il nostro programma *torriHanoi.jas* sul nostro elaboratore **mic1** è in grado di supportare.

È possibile verificare quanto scritto sopra all’interno dell’emulatore a noi in dotazione nel seguente modo: si esegua il programma *torriHanoi.jas* con un qualsiasi numero di dischi. Al termine dell’esecuzione si potrà constatare che l’ultima cella “sporca” (dovrebbe contenere 0x0030: il valore utilizzato da sposta(a,b) per stampare una cifra numerica in ASCII) dello stack si troverà all’indirizzo dove è il numero di dischi inseriti in input.

c