

Lorenzo Ferron, Gabor Galazzo, Alderico Gallo
Primo anno del Corso di Laurea in Informatica

ESERCIZIO 3: Analisi del comportamento di metodi ricorsivi

Architettura degli Elaboratori 2

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

Alessandria
Università degli Studi del Piemonte Orientale "Amedeo Avogadro"
A. A. 2017-2018

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 questo preambolo di quattro byte dalla base del metodo nell'**area dei metodi** sarà presente la prima microistruzione da eseguire. Questa istruzione prima di essere eseguita ha bisogno di parametri addizionali sulla cima dello stack, ovvero i parametri del metodo richiamato, preceduti da un valore chiamato **OBJREF**.

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 Constant 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, le variabili locali del metodo chiamato (dato dai primi quattro byte alla base del metodo nell'**area dei metodi**) e gli operandi. 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.

The image displays two side-by-side code snippets. On the left is the Java source code for the `torriHanoi` method, and on the right is its equivalent in JAS (Java Abstract Syntax) bytecode. Colored dashed boxes connect corresponding lines between the two, illustrating the mapping. Breakpoints are indicated by small colored squares in the JAS code, corresponding to key control flow points in the Java code.

```

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));
    }
}

```

```

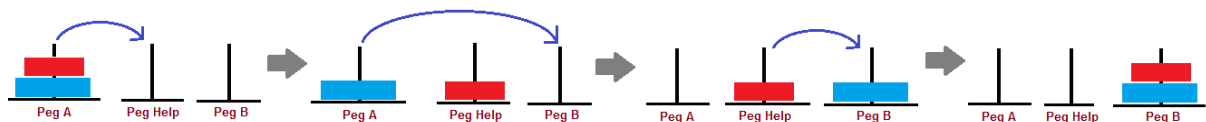
.method torriHanoi(n,a,b)
    .var
    .c
    .end-var
    ILOAD n // Se n=1; stampare mossa: a->b
    BIPUSH 1
    IF_ICMPEQ sposta
    BIPUSH 6 // calcola l'indice del terzo piolo = 6 - a - b
    ILOAD a
    ISUB
    ILOAD b
    ISUB
    ISTORE c // c contiene l'indice del terzo piolo (indici: 1, 2, 3)
    LDC_W OBJREF
    ILOAD n
    BIPUSH 1
    ISUB
    ILOAD a
    ILOAD c
    INVOKEVIRTUAL torriHanoi // richiama torriHanoi(n-1,a,c)
    IADD // ritorna sulla cima dello stack il numero di mosse
    LDC_W OBJREF
    ILOAD a
    ILOAD b
    INVOKEVIRTUAL sposta // stampa lo spostamento di 1 disco dal piolo a al b:
    // proprio come farebbe torriHanoi(1,a,b)
    // ma così si risparmia una INVOKEVIRTUAL
    IADD
    LDC_W OBJREF
    ILOAD n
    BIPUSH 1
    ISUB
    ILOAD c
    ILOAD b
    INVOKEVIRTUAL torriHanoi // richiama torriHanoi(n-1,c,b)
    IADD // aggiorna il numero di mosse
    IRETURN // restituisce il numero complessivo di mosse
sposta:
    LDC_W OBJREF
    ILOAD a
    ILOAD b
    INVOKEVIRTUAL sposta
    IRETURN
.end-method

```

(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 $n = 0 \times 2$ (numero di dischi da spostare), $a = 0 \times 1$ (piolo di partenza), $b = 0 \times 3$ (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.

	Word Address (hex)	Content (hex)	Pointers
50 c	8000	00000000	
51 .end-var	8001	00008006	<-- LV #1
52 ILOAD n // Se n=1: stampare mossa: a->b	8002	00000002	#1
53 BIPUSH 1	8003	00000001	#1
54 (0x38) IF_ICMPEQ sposta	8004	00000003	#1
55 BIPUSH 6 // calcola l'indice del terzo piolo = 6 - a - b	8005	0000001d	#1
56 ILOAD a	8006	00000028	#1
57 ISUB	8007	00008000	#1
58 ILOAD b	8008	00000002	#1
59 ISUB	8009	00000001	<-- SP #1
60 ISTORE c // c contiene l'indice del terzo piolo (indici: 1, 2, 3)	800a	0000000a	
61 LDC_W OBJREF	800b	0000000a	

A questo punto il programma avrà determinato quale sarà il piolo d'appoggio e lo avrà salvato in `c`: infatti all'indirizzo `0x8005` troviamo 0×2 ovvero $0 \times 6 - a - b = 0 \times 6 - 0 \times 1 - 0 \times 3$. 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`).

	Word Address (hex)	Content (hex)	Pointers
50 c	8000	00000000	
51 .end-var	8001	00008006	<-- LV #1
52 ILOAD n // Se n=1: stampare mossa: a->b	8002	00000002	#1
53 BIPUSH 1	8003	00000001	#1
54 (0x38) IF_ICMPEQ sposta	8004	00000003	#1
55 BIPUSH 6 // calcola l'indice del terzo piolo = 6 - a - b	8005	00000002	#1
56 ILOAD a	8006	00000028	#1
57 ISUB	8007	00008000	#1
58 ILOAD b	8008	00000444	#1
59 ISUB	8009	00000001	#1
60 ISTORE c // c contiene l'indice del terzo piolo (indici: 1, 2, 3)	800a	00000001	#1
61 LDC_W OBJREF	800b	00000002	<-- SP #1
62 ILOAD n	800c	0000000a	
63 BIPUSH 1	800d	00000032	
64 ISUB	800e	0000006e	
65 ILOAD a	800f	00008001	
66 ILOAD c	8010	00000001	
67 (0x51) INVOKEVIRTUAL torriHanoi	8011	00000002	
68 // ritorna sulla cima dello stack il numero di mosse	8012	00000003	

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`.

	Word Address (hex)	Content (hex)	Pointers
50 c	8000	00000000	
51 .end-var	8001	00008006	
52 ILOAD n // Se n=1: stampare mossa: a->b	8002	00000002	#1
53 BIPUSH 1	8003	00000001	#1
54 (0x38) IF_ICMPEQ sposta	8004	00000003	#1
55 BIPUSH 6 // calcola l'indice del terzo piolo = 6 - a - b	8005	00000002	#1
56 ILOAD a	8006	00000028	#1
57 ISUB	8007	00008000	#1
58 ILOAD b	8008	0000800d	<-- LV #2
59 ISUB	8009	00000001	#2
60 ISTORE c // c contiene l'indice del terzo piolo (indici: 1, 2, 3)	800a	00000001	#2
61 LDC_W OBJREF	800b	00000002	#2
62 ILOAD n	800c	0000000a	#2
63 BIPUSH 1	800d	00000054	#2
64 ISUB	800e	00008001	#2
65 ILOAD a	800f	00000001	#2
66 ILOAD c	8010	00000001	<-- SP #2
67 (0x51) INVOKEVIRTUAL torriHanoi // richiama torriHanoi(n-1,a,c)	8011	00000002	
68 // ritorna sulla cima dello stack il numero di mosse	8012	00000003	
69 (0x54) LDC_W OBJREF	8013	0000007a	

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 dal piolo sorgente **a** al piolo destinazione **b**. Sulla console verrà ora stampato **1 => 2**

	Word Address (hex)	Content (hex)	Pointers
74	8000	00000000	
75 (0x5e) IADD	8001	00008006	
76 LDC W OBJREF	8002	00000002	#1
77 ILOAD n	8003	00000001	#1
78 BIPUSH 1	8004	00000003	#1
79 ISUB	8005	00000028	#1
80 ILOAD c	8006	00000028	#1
81 ILOAD b	8007	00008000	#1
82 (0x6b) INVOKEVIRTUAL torriHanoi // richiama torriHanoi(n-1,c,b)	8008	0000800d	<-- LV #2
83 (0x6e) IADD // aggiorna il numero di mosse	8009	00000001	#2
84 (0x6f) IRETURN // restituisce il numero complessivo di mosse	800a	00000001	#2
85 sposta:	800b	00000002	#2
86 (0x70) LDC W OBJREF	800c	0000000a	#2
87 ILOAD a	800d	00000054	#2
88 ILOAD b	800e	00002001	<-- SP #2
89 INVOKEVIRTUAL sposta	800f	00000001	
90 (0x7a) IRETURN	8010	00000001	
91 .end-method	8011	00000002	

	ss (hex)	Content (hex)	Pointers
74		00000000	
75 (0x5e) IADD		00008006	
76 LDC W OBJREF		00000002	#1
77 ILOAD n		00000001	#1
78 BIPUSH 1		00000003	#1
79 ISUB		00000002	#1
80 ILOAD c		00000028	#1
81 ILOAD b		00008000	#1
82 (0x6b) INVOKEVIRTUAL torriHanoi // richiama torriHanoi(n-1,c,b)		0000800d	
83 (0x6e) IADD // aggiorna il numero di mosse		00000001	#2
84 (0x6f) IRETURN // restituisce il numero complessivo di mosse		00000001	#2
85 sposta:		00000002	#2
86 (0x70) LDC W OBJREF		0000000a	#2
87 ILOAD a		00000054	#2
88 ILOAD b		00008001	#2
89 INVOKEVIRTUAL sposta		00008012	<-- SP #3
90 (0x7a) IRETURN		00000001	
91 .end-method		00000002	
92 .method sposta(i,j) // Il metodo stampa mossa: i => j		0000007a	<-- LV #3
93 ILOAD i		00008008	
94 LDC W ZERO		00000001	
95 IADD		00000030	
96 OUT			

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**.

	ss (hex)	Content (hex)	Pointers
65 ILOAD a		00000000	
66 ILOAD c		00008006	<-- LV #1
67 (0x51) INVOKEVIRTUAL torriHanoi // richiama torriHanoi(n-1,a,c)		00000002	#1
68		00000001	#1
69 (0x54) LDC W OBJREF // ritorna sulla cima dello stack il numero di mosse		00000003	#1
70 ILOAD a		00000002	#1
71 ILOAD b		00000028	#1
72 (0x5b) INVOKEVIRTUAL sposta // stampa lo spostamento di 1 disco dal piolo a al b:		00008000	#1
73 // proprio come farebbe torriHanoi(1,a,b)		00000001	<-- SP #1
74 // ma così si risparmia una INVOKEVIRTUAL		00000001	
75 (0x5e) IADD		00000001	
76 LDC W OBJREF		00000002	
77 ILOAD n		0000000a	

	Word Address (hex)	Content (hex)	Pointers
65 ILOAD a	8000	00000000	
66 ILOAD c	8001	00008006	<-- LV #1
67 (0x51) INVOKEVIRTUAL torriHanoi // richiama torriHanoi(n-1,a,c)	8002	00000002	#1
68	8003	00000001	#1
69 (0x54) LDC W OBJREF // ritorna sulla cima dello stack il numero di mosse	8004	00000003	#1
70 ILOAD a	8005	00000002	#1
71 ILOAD b	8006	00000028	#1
72 (0x5b) INVOKEVIRTUAL sposta // stampa lo spostamento di 1 disco dal piolo a al b:	8007	00008000	#1
73 // proprio come farebbe torriHanoi(1,a,b)	8008	00000001	#1
74 // ma così si risparmia una INVOKEVIRTUAL	8009	00000001	<-- SP #1
75 (0x5e) IADD	800a	00000001	
76 LDC W OBJREF	800b	00000003	
77 ILOAD n	800c	0000005e	
78 BIPUSH 1	800d	00008001	
79 ISUB	800e	00000001	
80 ILOAD c	800f	00000030	
81 ILOAD b	8010	00000001	

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`).

Word Address (hex)	Content (hex)	Pointers
8000	00000000	
8001	00008006	<-- LV #1
8002	00000002	#1
8003	00000001	#1
8004	00000003	#1
8005	00000002	#1
8006	00000028	#1
8007	00008000	#1
8008	00000002	#1
8009	00000444	#1
800a	00000001	#1
800b	00000002	#1
800c	00000003	<-- SP #1
800d	00008001	
800e	00000001	
800f	00000030	
8010	00000001	
8011	00000002	
8012	0000007a	
8013	00008008	

Con i tre fotogrammi successivi 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`.

Word Address (hex)	Content (hex)	Pointers
8000	00000000	
8001	00008006	
8002	00000002	#1
8003	00000001	#1
8004	00000003	#1
8005	00000002	#1
8006	00000028	#1
8007	00008000	#1
8008	00000002	#1
8009	0000800e	<-- LV #2
800a	00000001	#2
800b	00000002	#2
800c	00000003	#2
800d	00008001	#2
800e	0000000e	#2
800f	00008001	#2
8010	00000001	#2
8011	00000002	<-- SP #2
8012	0000007a	
8013	00008008	

Word Address (hex)	Content (hex)	Pointers
8000	00000000	
8001	00008006	
8002	00000002	#1
8003	00000001	#1
8004	00000003	#1
8005	00000002	#1
8006	00000028	#1
8007	00008000	#1
8008	00000002	#1
8009	0000800e	<-- LV #2
800a	00000001	#2
800b	00000002	#2
800c	00000003	#2
800d	00008001	#2
800e	0000000e	#2
800f	00008001	<-- SP #2
8010	00000001	
8011	00000001	

Word Address (hex)	Content (hex)	Pointers
8000	00000000	
8001	00008006	
8002	00000002	#1
8003	00000001	#1
8004	00000003	#1
8005	00000002	#1
8006	00000028	#1
8007	00008000	#1
8008	00000002	#1
8009	0000800e	<-- LV #2
800a	00000001	#2
800b	00000002	#2
800c	00000003	#2
800d	00008001	#2
800e	0000000e	#2
800f	00008001	#2
8010	00000001	<-- SP #2
8011	00000002	

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*.

The screenshot shows the UVM Editor with assembly code on the left and a memory dump on the right. The code is for a recursive function *torriHanoi* and a helper function *sposta*. The memory dump shows the stack frame for *torriHanoi* at address 8000, with parameters *a*, *b*, and *c* stored in registers. The stack pointer *SP* is at 8009, and the instruction pointer *IP* is at 8008.

Word Address (hex)	Content (hex)	Pointers
8000	00000000	
8001	00008006	<-- LV #1
8002	00000002	#1
8003	00000001	#1
8004	00000003	#1
8005	00000002	#1
8006	00000028	#1
8007	00008000	#1
8008	00000002	#1
8009	00000001	<-- SP #1
800a	00000001	
800b	00000002	
800c	00000003	
800d	00008001	

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

The screenshot shows the UVM Editor with the final state of the program execution. The code is for a recursive function *torriHanoi* and a helper function *sposta*. The memory dump shows the stack frame for *torriHanoi* at address 8000, with parameters *a*, *b*, and *c* stored in registers. The stack pointer *SP* is at 8009, and the instruction pointer *IP* is at 8008.

Word Address (hex)	Content (hex)	Pointers
8000	00000000	<-- LV
8001	00000000	<-- SP
8002	00000003	
8003	00000001	
8004	00000003	
8005	0000002f	
8006	00000000	
8007	00000003	
8008	00000001	
8009	00000001	
800a	0000000a	
800b	00000003	
800c	00000001	
800d	00000004	
800e	00000001	
800f	00000001	
8010	00000001	
8011	00000002	
8012	00000003	
8013	0000007a	

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 limite, la chiamata a *sposta*(*a*,*b*) occupa 6 word, allora il massimo spazio che può occupare sullo stack una chiamata con *n* dischi è (8*n* + 6) word e, dato che, una word sono (in questa architettura) 4 byte alla fine lo spazio occupato è di 4(8*n* + 6) byte. Nel nostro emulatore la dimensione totale dello stack è di 0x10000 word (256kB) e l'indirizzo di partenza è 0x8000 ovvero la metà lasciando a disposizione 0x8000 word (128kb) per allocare record di attivazione, quindi

$$(8n + 6) < 32768 \rightarrow n < 4096$$

oppure

$$(0x8)n + 0x6 < 0x8000 \rightarrow n < 0x0FFF$$

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 $(0x8)n + 0x6 + 0x8000$ dove n è il numero di dischi inseriti in input.