Alexandru Stefan Andries, Nicolò Benella

Relazione Architettura 2

Compito : C

Sommario

[Presentazione del lavoro svolto 2](#_Toc485463790)

[Nuove istruzioni aggiunte 2](#_Toc485463791)

[IFGE label : 2](#_Toc485463792)

[INOT : 2](#_Toc485463793)

[ISHL : 2](#_Toc485463794)

[Test delle nuove istruzioni 3](#_Toc485463795)

[Test IFGE : 3](#_Toc485463796)

[Cicli nel Debug 3](#_Toc485463797)

[Registri e Memory Area 4](#_Toc485463798)

[Test INOT : 4](#_Toc485463799)

[Cicli nel Debug 4](#_Toc485463800)

[Registri e Memory Area 5](#_Toc485463801)

[TEST ISHL : 5](#_Toc485463802)

[Cicli nel Debug 5](#_Toc485463803)

[Registri e Memory Area 6](#_Toc485463804)

[Realizzazione dei metodi e del main 6](#_Toc485463805)

[Il main nel nostro caso è semplicemente un richiamo alle due funzione che abbiamo realizzato, ovvero leggi\_int\_base16 e StampaBin. 6](#_Toc485463806)

[Test e commenti 6](#_Toc485463807)

[Caso 1: 0x0 8](#_Toc485463808)

[Caso 2: 0xf 8](#_Toc485463809)

[Caso 3: 0xfffc 9](#_Toc485463810)

[Caso 4: 0xfffffff 10](#_Toc485463811)

[Conclusione 11](#_Toc485463812)

# Presentazione del lavoro svolto

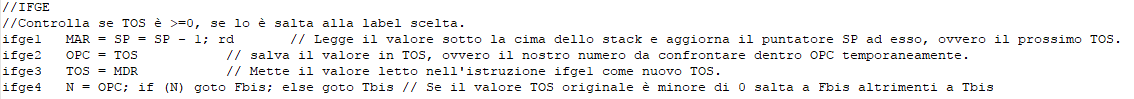
L’esercitazione svolta nel lavoro di gruppo del corso di Architettura 2 consisteva principalmente nel creare un codice IJVM in grado di leggere in input un numero inserito in esadecimale e far stampare a video il suo valore binario.

Per scrivere questo programma è stato necessaria l’aggiunta di 3 nuove istruzioni per il mic1sim e la creazione di funzione intermedie che usate insieme hanno reso possibile la riuscita del compito.

# Nuove istruzioni aggiunte

*Tutte le descrizioni delle istruzione sono state estratte dalla specifica di IJVM presente a questo* [link](http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html)

### IFGE *label* :



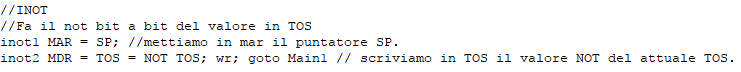
L’istruzione IFGE prende dalla cima dello stack l’ultimo valore inserito, che deve essere obbligatoriamente un valore int, e lo mette a confronto con il valore 0.

La funzione ha successo nel caso in cui il valore è > 0.

In seguito la funzione esegue le classiche operazioni di una normale if e si sposta su uno dei due *branchbyte* per compiere le successive istruzioni.

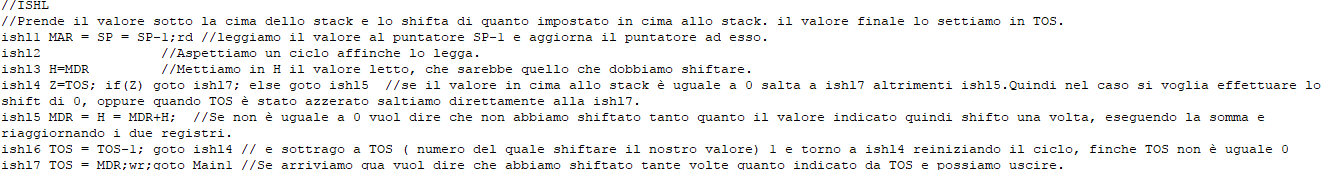
Nel nostro caso se la funzione ha successo si salta alla label indicata.

### INOT :



L’istruzione INOT (non presente nella specifica di IJVM) prende l’ultimo valore inserito sullo stack e calcola il suo not bit a bit, reinserendo poi il risultato nello stessa posizione in cui si trovava il valore inziale.

### ISHL :



L’istruzione ISHL viene utilizzata per effettuare lo shift di un valore.

Essa prende il valore presente in mem[SP-1] ovvero il penultimo valore inserito e ne esegue lo shift di un valore specificato in mem[SP] ovvero l’ultimo valore inserito.

Il risultato è infine inserito in cima allo stack ( solitamente dovrebbe sostituirsi al valore dato per lo shift ).

# Test delle nuove istruzioni

Dopo averle aggiunte, abbiamo ovviamente dovuto testarle per tutti i casi possibili di esecuzione dell’istruzione, in modo tale da avere la certezza che funzionino sempre per ogni tipo di utilizzo.

## Test IFGE :

Il test delle IFGE è stato piuttosto semplice in quanto essendo un if ed accetando solo int, ci sono semplicemente solo 3 casi possibili ovvero.

1. TOS = 0 :

In questo caso visto che la IFGE ha successo se il valore inserito è >= 0, la nostra IFGE completa con successo la sua esecuzione e salla alla label indicata.

1. TOS >0

Anche in questo caso la IFGE ha successo e salta alla label indicata.

1. TOS < 0

In questo caso TOS è minore e quindi la IFGE fallisce ed il programma prosegue nel codice senza effettuare nessun salto.

### Cicli nel Debug

-----------------Start cycle 8----------------------

PC=PC+1;fetch;goto (MBR)

PC: Put 2

H: Put 0x0

ALU: 0 + B + 1 = 0x3

PC: Store 3

PC: Fetch byte 3

MEM: Write value 0x0 to address 0x20004

MEM: Fetch from byte# 0x3 requested. Processing...

JMPC is set: true MBR: 0x9C ADDR: 0x0

NEXT MPC: 0x9C

-----------------Start cycle 9----------------------

SP=MAR=SP-1;rd;goto 0x5

SP: Put 0x8001

H: Put 0x0

ALU: NOT 0 + B = 0x8000

MAR: Store 0x8000

SP: Store 0x8000

MAR: Read from word 0x8000

MEM: Read from word# 0x20000 requested. Processing...

MEM: Fetch value 0x0 from address 0x3

MBR: Store 0x0

Goto ADDR: 0x5

-----------------Start cycle 10----------------------

OPC=TOS;goto 0x6

TOS: Put 0x0

H: Put 0x0

ALU: 0 OR B = 0x0

OPC: Store 0x0

MDR: Read 0x0

MEM: Read value 0x0 from address 0x20000

Goto ADDR: 0x6

-----------------Start cycle 11----------------------

TOS=MDR;goto 0x7

MDR: Put 0x0

H: Put 0x0

ALU: 0 OR B = 0x0

TOS: Store 0x0

Goto ADDR: 0x7

-----------------Start cycle 12----------------------

N=OPC;if (N) goto 0x101; else goto 0x1

OPC: Put 0x0

H: Put 0x0

ALU: 0 OR B = 0x0

Goto ADDR: 0x1

-----------------Start cycle 13----------------------

OPC=PC-1;fetch;goto 0x8

PC: Put 3

H: Put 0x0

ALU: NOT 0 + B = 0x2

OPC: Store 0x2

PC: Fetch byte 3

MEM: Fetch from byte# 0x3 requested. Processing...

Goto ADDR: 0x8

-----------------Start cycle 14----------------------

PC=PC+1;fetch;goto 0x3C

PC: Put 3

H: Put 0x0

ALU: 0 + B + 1 = 0x4

PC: Store 4

PC: Fetch byte 4

MEM: Fetch value 0x0 from address 0x3

MEM: Fetch from byte# 0x4 requested. Processing...

MBR: Store 0x0

Goto ADDR: 0x3C

-----------------Start cycle 15----------------------

H=MBR<<8;goto 0x3D

MBR: Put 0x0

H: Put 0x0

ALU: 0 OR B = 0x0

H: Store 0x0

MEM: Fetch value 0x7 from address 0x4

MBR: Store 0x7

Goto ADDR: 0x3D

-----------------Start cycle 16----------------------

H=H OR MBRU;goto 0x3E

MBR: Put 0x7

H: Put 0x0

ALU: A OR B = 0x7

H: Store 0x7

Goto ADDR: 0x3E

-----------------Start cycle 17----------------------

PC=H+OPC;fetch;goto 0x3F

OPC: Put 0x2

H: Put 0x7

ALU: A + B = 0x9

PC: Store 9

PC: Fetch byte 9

MEM: Fetch from byte# 0x9 requested. Processing...

Goto ADDR: 0x3F

-----------------Start cycle 18----------------------

goto 0x2

MDR: Put 0x0

H: Put 0x7

ALU: 0 AND 0 = 0x0

MEM: Fetch value 0x10 from address 0x9

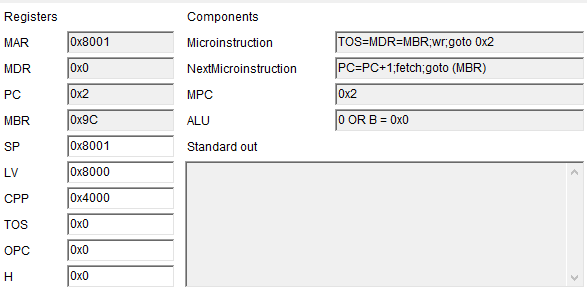
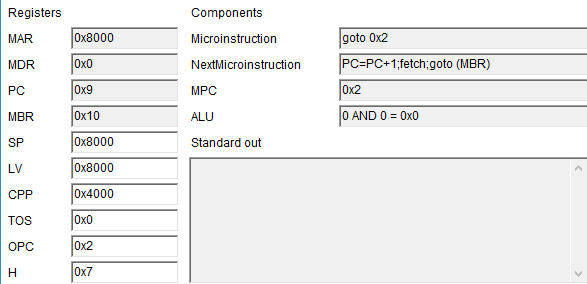
MBR: Store 0x10

Goto ADDR: 0

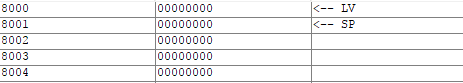
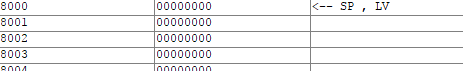
### Registri e Memory Area

**Prima**

**Dopo**



*In questo caso il valore inserito era 0*



In questo caso il funzionamento è corretto in quanto dopo aver inserito un valore come si può vedere dallo stack prima ( il valore è 0), la IFGE preleva quel valore e torna indietro di uno spazio nello stack. Infine siccome l’IFGE è corretta viene effettuato il salto.

Il procedimento è lo stesso per tutti e 3 i casi possibili della IFGE.

## Test INOT :

Il test della INOT è ancora più semplice di quello della IFGE, in quanto essa deve semplicemente fare il NOT bit a bit del valore inserito.

Il test deve quindi semplicemente inserire un valore e vedere se è in grado di fare il NOT.

### Cicli nel Debug

-----------------Start cycle 8----------------------

PC=PC+1;fetch;goto (MBR)

PC: Put 2

H: Put 0x0

ALU: 0 + B + 1 = 0x3

PC: Store 3

PC: Fetch byte 3

MEM: Write value 0x2 to address 0x20004

MEM: Fetch from byte# 0x3 requested. Processing...

JMPC is set: true MBR: 0xA3 ADDR: 0x0

NEXT MPC: 0xA3

-----------------Start cycle 9----------------------

MAR=SP;goto 0xA

SP: Put 0x8001

H: Put 0x0

ALU: 0 OR B = 0x8001

MAR: Store 0x8001

MEM: Fetch value 0x10 from address 0x3

MBR: Store 0x10

Goto ADDR: 0xA

-----------------Start cycle 10----------------------

TOS=MDR=NOT TOS;wr;goto 0x2

TOS: Put 0x2

H: Put 0x0

ALU: NOT B = 0xFFFFFFFD

MDR: Store 0xFFFFFFFD

TOS: Store 0xFFFFFFFD

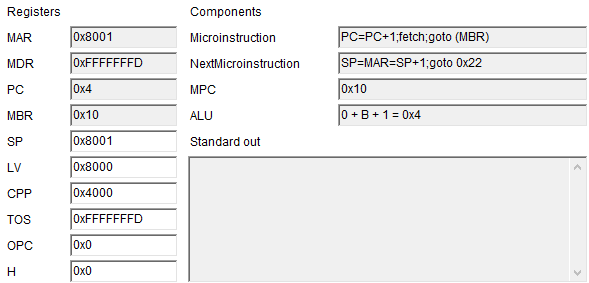
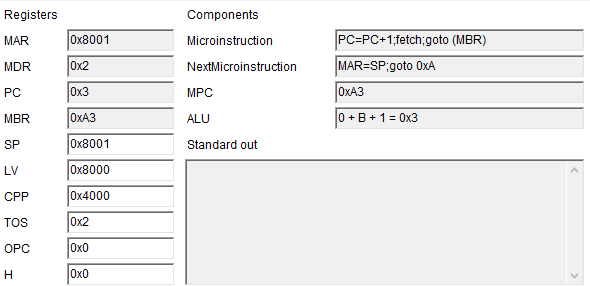
MAR: Write to word 0x8001

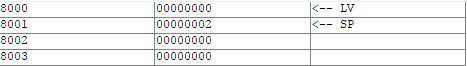
MDR: Write 0xFFFFFFFD

MEM: Write value 0xFFFFFFFD to word# 0x20004 requested. Processing...

Goto ADDR: 0x2

### Registri e Memory Area

https://puu.sh/whTNr/95ec693c78.png **Prima**  **Dopo**



In questo caso il funzionamento è corretto in quanto 0xfd è il not di 2.

Per esserne sicuri si può fare il controllo eseguendo il not di 0xfd ed il risultato è 2.

## TEST ISHL :

Il test dell’istruzione ISHL consiste nel passare in input due valori e controllare se lo shift viene effettuato nel modo corretto.

Non ci sono molti casi particolari.

### Cicli nel Debug

-----------------Start cycle 13----------------------

SP=MAR=SP-1;rd;goto 0xB

SP: Put 0x8002

H: Put 0x0

ALU: NOT 0 + B = 0x8001

MAR: Store 0x8001

SP: Store 0x8001

MAR: Read from word 0x8001

MEM: Read from word# 0x20004 requested. Processing...

MEM: Fetch value 0x10 from address 0x5

MBR: Store 0x10

Goto ADDR: 0xB

-----------------Start cycle 14----------------------

goto 0xC

MDR: Put 0x2

H: Put 0x0

ALU: 0 AND 0 = 0x0

MDR: Read 0x1

MEM: Read value 0x1 from address 0x20004

Goto ADDR: 0xC

-----------------Start cycle 15----------------------

H=MDR;goto 0xD

MDR: Put 0x1

H: Put 0x0

ALU: 0 OR B = 0x1

H: Store 0x1

Goto ADDR: 0xD

-----------------Start cycle 16----------------------

Z=TOS;if (Z) goto 0x103; else goto 0x3

TOS: Put 0x2

H: Put 0x1

ALU: 0 OR B = 0x2

Goto ADDR: 0x3

-----------------Start cycle 17----------------------

H=MDR=H+MDR;goto 0xE

MDR: Put 0x1

H: Put 0x1

ALU: A + B = 0x2

MDR: Store 0x2

H: Store 0x2

Goto ADDR: 0xE

-----------------Start cycle 18----------------------

TOS=TOS-1;goto 0xD

TOS: Put 0x2

H: Put 0x2

ALU: NOT 0 + B = 0x1

TOS: Store 0x1

Goto ADDR: 0xD

-----------------Start cycle 19----------------------

Z=TOS;if (Z) goto 0x103; else goto 0x3

TOS: Put 0x1

H: Put 0x2

ALU: 0 OR B = 0x1

Goto ADDR: 0x3

-----------------Start cycle 20----------------------

H=MDR=H+MDR;goto 0xE

MDR: Put 0x2

H: Put 0x2

ALU: A + B = 0x4

MDR: Store 0x4

H: Store 0x4

Goto ADDR: 0xE

-----------------Start cycle 21----------------------

TOS=TOS-1;goto 0xD

TOS: Put 0x1

H: Put 0x4

ALU: NOT 0 + B = 0x0

TOS: Store 0x0

Goto ADDR: 0xD

-----------------Start cycle 22----------------------

Z=TOS;if (Z) goto 0x103; else goto 0x3

TOS: Put 0x0

H: Put 0x4

ALU: 0 OR B = 0x0

Goto ADDR: 0x3

-----------------Start cycle 23----------------------

TOS=MDR;wr;goto 0x2

MDR: Put 0x4

H: Put 0x4

ALU: 0 OR B = 0x4

TOS: Store 0x4

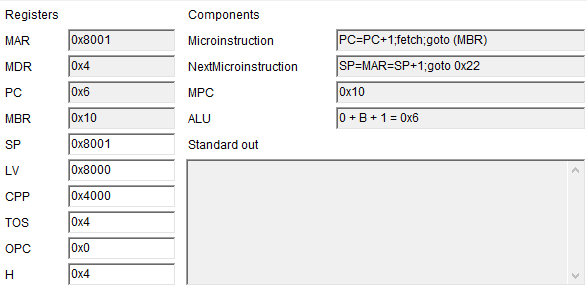
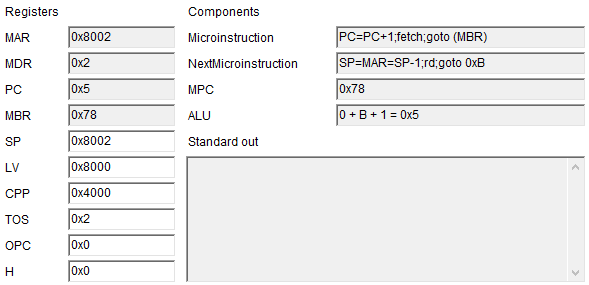
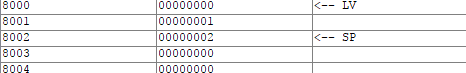
MAR: Write to word 0x8001

MDR: Write 0x4

MEM: Write value 0x4 to word# 0x20004 requested. Processing...

Goto ADDR: 0x2

### Registri e Memory Area

**Prima Dopo**https://puu.sh/whUFv/24eb41ed3c.png

Anche in questo caso il risultato è corretto in quanto uno equivale a 01, shiftato di 2 sarebbe 100 che equivale quindi a 4.

# Realizzazione dei metodi e del main

## Il main nel nostro caso è semplicemente un richiamo alle due funzione che abbiamo realizzato, ovvero leggi\_int\_base16 e StampaBin.

Esso con la prima funzione legge il numero in base 16 che gli diamo in input cifra per cifra.

Il valore finale che ha letto verrà dato in input alla funzione StampaBin.

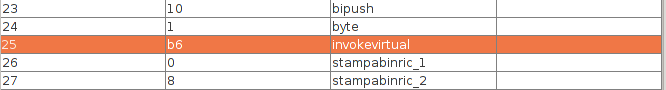
Essa controllare innanzittuo se il valore è semplicemente 0, in quel caso non ha senso andare avanti e stampa 0, altrimenti inzierà una serie di richiami ricorsivi della funzione StampaBinRic, in base a quanto è grande il valore inziale dato in input.

Alla fine dei richiami essa stamperà numero per numero il valore binario del valore esadecimale inserito in precedenza.

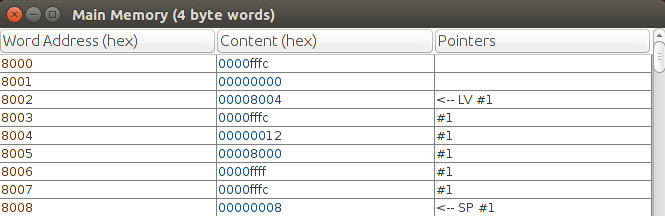
## Test e commenti

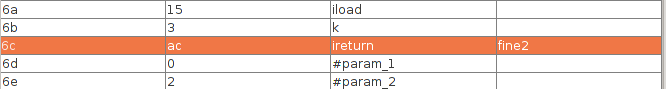
Nella realizzazione dei test abbiamo deciso di utilizzare 4 diversi casi: 0x0, 0xf, 0xfffc, 0xfffffff. Questi 4 casi sono stati scelti perché ognuno di essi rappresenta il caso migliore, medio e maggiore quindi mostriamo come il codice esegua più o meno chiamate ricorsive a seconda del dato inserito. Per poter estrarre i valori degli indirizzi dei parametri in ciascun record di attivazione e per poter controllare la massima profondità di ricorsione abbiamo inserito 2 breakpoint, questi 2 breakpoint sono rispettivamente agli indirizzi 25 e 6c della method area.

L’indirizzo 25 rappresenta l’istruzione INVOKEVIRTUAL nel metodo Stampa\_Bin (n) che richiama StampaBinRic (n, m), abbiamo scelto questo come primo punto di partenza perché all’interno della memoria possiamo controllare quante chiamate sono già presenti e contare quante sono quelle ricorsive che inizieranno nel momento in cui facciamo continuare il programma.

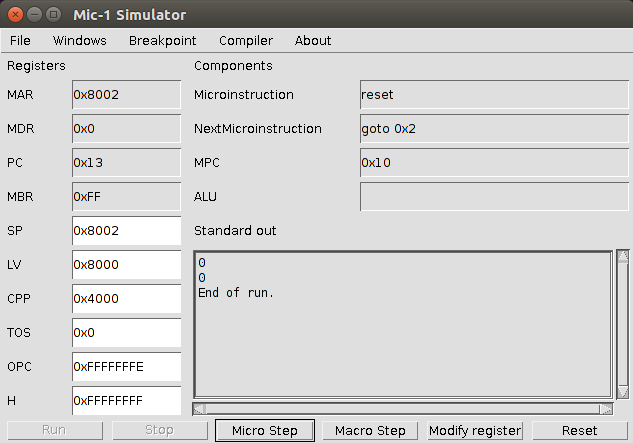


Questo per esempio è nel caso 0xfffc subito prima di iniziare il metodo StampaBinRic (n, m), sullo stack abbiamo in esecuzione il metodo Stampa\_Bin(n).



L’indirizzo 6c è inserito prima dell’esecuzione dell’istruzione IRETURN del metodo StampaBinRic (n, m), inseriamo qui il breakpoint per poter visualizzare il contenuto di ogni chiamata ricorsiva prima che cominci la stampa.

## Caso 1: 0x0

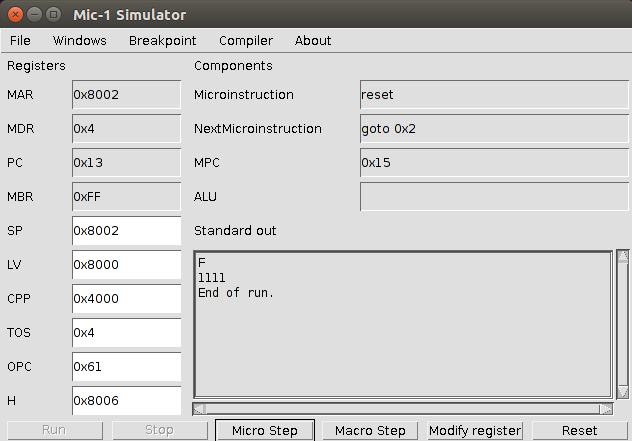
Questo è il caso più semplice in quanto non avviene nessuna chiamata ricorsiva e viene stampato direttamente 0 terminando il programma.

## Caso 2: 0xf

Anche questo è un caso piuttosto semplice in quanto vengono eseguite 5 chiamate ricorsive prima di iniziare la stampa.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Valore A | Valore B | Indirizzo Mem. A | Indirizzo Mem. B |
| Chiamata 1 | 0000000f | 00000001 | 8007 | 8008 |
| Chiamata 2 | 0000000f | 00000002 | 800e | 800f |
| Chiamata 3 | 0000000f | 00000004 | 8015 | 8016 |
| Chiamata 4 | 0000000f | 00000008 | 801c | 801d |
| Chiamata 5 | 0000000f | 00000010 | 8023 | 8024 |

Come possiamo osservare il primo parametro di ogni chiamata ricorsiva sarà sempre il nostro valore da convertire in binario, mentre il secondo parametro inserito subito dopo è il numero m che deve essere sempre multiplo di 2 e che aumenta ad ogni chiamata ricorsiva.



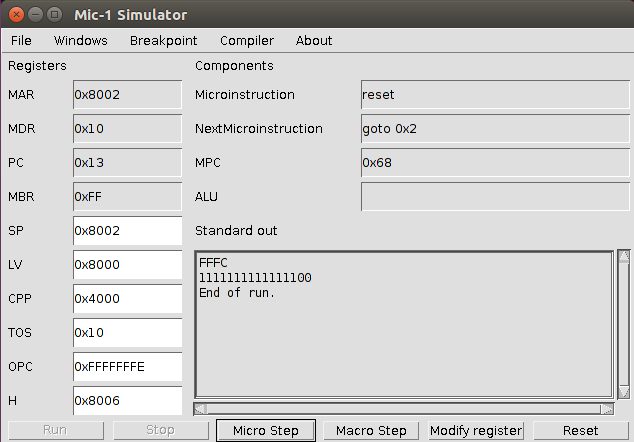
Questo è il risultato finale.

## Caso 3: 0xfffc

Questo caso invece lo abbiamo scelto come caso intermedio in quanto vengono eseguite 17 chiamate ricorsive prima della stampa.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Valore A | Valore B | Indirizzo Mem. A | Indirizzo Mem. B |
| Chiamata 1 | 0000fffc | 00000001 | 8007 | 8008 |
| Chiamata 2 | 0000fffc | 00000002 | 800e | 800f |
| Chiamata 3 | 0000fffc | 00000004 | 8015 | 8016 |
| Chiamata 4 | 0000fffc | 00000008 | 801c | 801d |
| Chiamata 5 | 0000fffc | 00000010 | 8023 | 8024 |
| Chiamata 6 | 0000fffc | 00000020 | 802a | 802b |
| Chiamata 7 | 0000fffc | 00000040 | 8031 | 8032 |
| Chiamata 8 | 0000fffc | 00000080 | 8038 | 8039 |
| Chiamata 9 | 0000fffc | 00000100 | 803f | 8040 |
| Chiamata 10 | 0000fffc | 00000200 | 8046 | 8047 |
| Chiamata 11 | 0000fffc | 00000400 | 804d | 804e |
| Chiamata 12 | 0000fffc | 00000800 | 8054 | 8055 |
| Chiamata 13 | 0000fffc | 00001000 | 805b | 805c |
| Chiamata 14 | 0000fffc | 00002000 | 8062 | 8063 |
| Chiamata 15 | 0000fffc | 00004000 | 8069 | 806a |
| Chiamata 16 | 0000fffc | 00008000 | 8070 | 8071 |
| Chiamata 17 | 0000fffc | 00010000 | 8077 | 8078 |

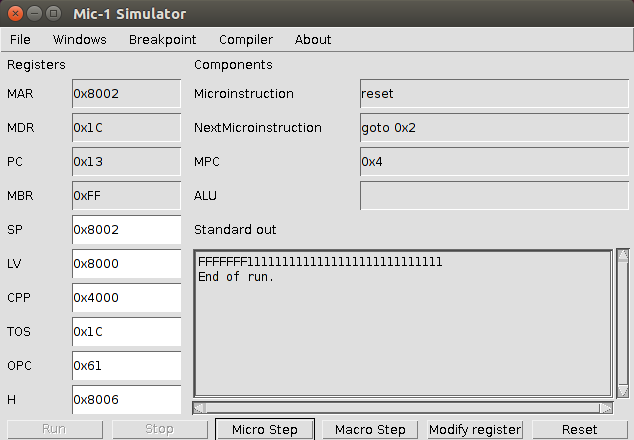
Anche qui il primo parametro sarà sempre il valore da convertire in binario mentre il secondo parametro sarà sempre il numero m multiplo di 2 che aumenta ad ogni chiamata ricorsiva.



## Caso 4: 0xfffffff

Questo è l’ultimo caso che osserveremo perché trattasi del caso in cui inseriamo il numero più grande che possiamo inserire e corrisponde al caso peggiore possibile. In questo esempio avvengono 29 chiamate ricorsive prima della stampa.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Valore A | Valore B | Indirizzo Mem. A | Indirizzo Mem. B |
| Chiamata 1 | 0fffffff | 00000001 | 8007 | 8008 |
| Chiamata 2 | 0fffffff | 00000002 | 800e | 800f |
| Chiamata 3 | 0fffffff | 00000004 | 8015 | 8016 |
| Chiamata 4 | 0fffffff | 00000008 | 801c | 801d |
| Chiamata 5 | 0fffffff | 00000010 | 8023 | 8024 |
| Chiamata 6 | 0fffffff | 00000020 | 802a | 802b |
| Chiamata 7 | 0fffffff | 00000040 | 8031 | 8032 |
| Chiamata 8 | 0fffffff | 00000080 | 8038 | 8039 |
| Chiamata 9 | 0fffffff | 00000100 | 803f | 8040 |
| Chiamata 10 | 0fffffff | 00000200 | 8046 | 8047 |
| Chiamata 11 | 0fffffff | 00000400 | 804d | 804e |
| Chiamata 12 | 0fffffff | 00000800 | 8054 | 8055 |
| Chiamata 13 | 0fffffff | 00001000 | 805b | 805c |
| Chiamata 14 | 0fffffff | 00002000 | 8062 | 8063 |
| Chiamata 15 | 0fffffff | 00004000 | 8069 | 806a |
| Chiamata 16 | 0fffffff | 00008000 | 8070 | 8071 |
| Chiamata 17 | 0fffffff | 00010000 | 8077 | 8078 |
| Chiamata 18 | 0fffffff | 00020000 | 807e | 807f |
| Chiamata 19 | 0fffffff | 00040000 | 8085 | 8086 |
| Chiamata 20 | 0fffffff | 00080000 | 808c | 808d |
| Chiamata 21 | 0fffffff | 00100000 | 8093 | 8094 |
| Chiamata 22 | 0fffffff | 00200000 | 809a | 809b |
| Chiamata 23 | 0fffffff | 00400000 | 80a1 | 80a2 |
| Chiamata 24 | 0fffffff | 00800000 | 80a8 | 80a9 |
| Chiamata 25 | 0fffffff | 01000000 | 80af | 80b0 |
| Chiamata 26 | 0fffffff | 02000000 | 80b6 | 80b7 |
| Chiamata 27 | 0fffffff | 04000000 | 80bd | 80be |
| Chiamata 28 | 0fffffff | 08000000 | 80c4 | 80c5 |
| Chiamata 29 | 0fffffff | 10000000 | 80cb | 80cc |



# Conclusione

In conclusione dunque possiamo dare con quasi assoluta certezza che il programma funziona correttamente in qualsiasi caso possibile, in quanto stampa sempre il valore corretto del numero esadecimale inserito e non ha problemi di overflow siccome sono presenti limiti di inserimento nel codice, specie nel codice di lettura intera base16.