

# count-char.asm – Analisi simulazione

a cura di Daniele Sana

Questo programma conta quante volte il carattere immesso da tastiera compare nella stringa memorizzata.

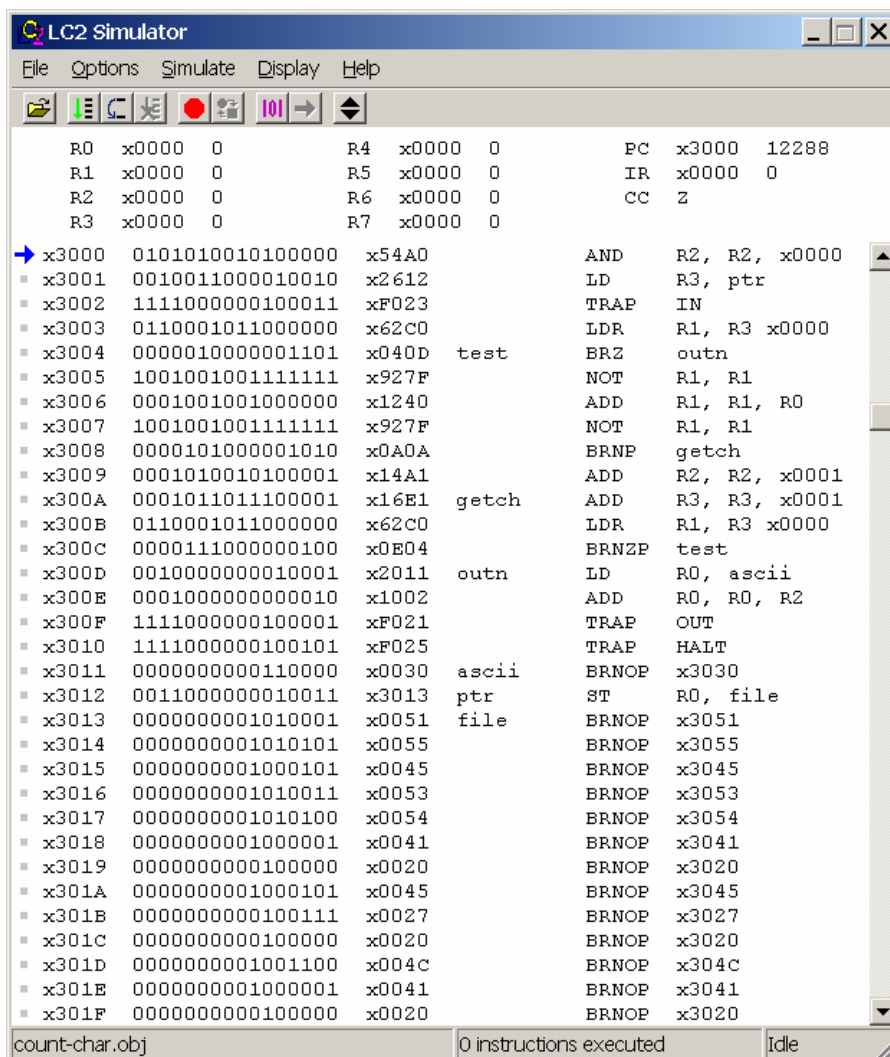
Il testo del programma principale è il seguente:

```
;
; Initialization
;
    .orig x3000
    AND    R2,R2,#0      ; R2 is counter, initialize to 0
    LD     R3,ptr        ; R3 is pointer to characters in the string
    TRAP   x23           ; R0 gets character input
    LDR    R1,R3,#0      ; R1 gets first character in string
;
; Test character for end of string
;
test  BRZ   outn         ; test for NULL; if done, prepare to output
;
; Test character for match
;
    NOT    R1,R1
    ADD    R1,R1,R0      ; if match, R1 = xFFFF
    NOT    R1,R1        ; if match, R1 = x0000
    BRNP   getch         ; if no match, skip the increment
    ADD    R2,R2,#1
;
; Get next character from string
;
getch ADD    R3,R3,#1     ; increment the pointer
    LDR    R1,R3,#0      ; R1 gets next character in string
    BR     test          ; loop to test character
;
; Output the count
;
outn  LD     R0,ascii     ; load the ASCII template for convert
    ADD    R0,R0,R2      ; convert binary to ASCII
    TRAP   x21           ; ASCII code in r0 is displayed
    TRAP   x25           ; Halt machine
;
; Storage for pointer, ASCII template and string
;
ascii .fill x0030
ptr   .fill file
file  .stringz "QUESTA E' LA STRINGA IN CUI CERCARE I CARATTERI"

.end
```

A questo punto, dopo aver compilato il programma utilizzando il programma *LC2Edit*, passiamo allo strumento *Simulate* ed effettuiamo il *debugging*, ossia eseguiamo il programma un'istruzione alla volta.

In *Simulate* la situazione è la seguente:



Vediamo innanzitutto quali saranno gli effetti delle pseudo-istruzioni *Assembly* presenti nel codice:

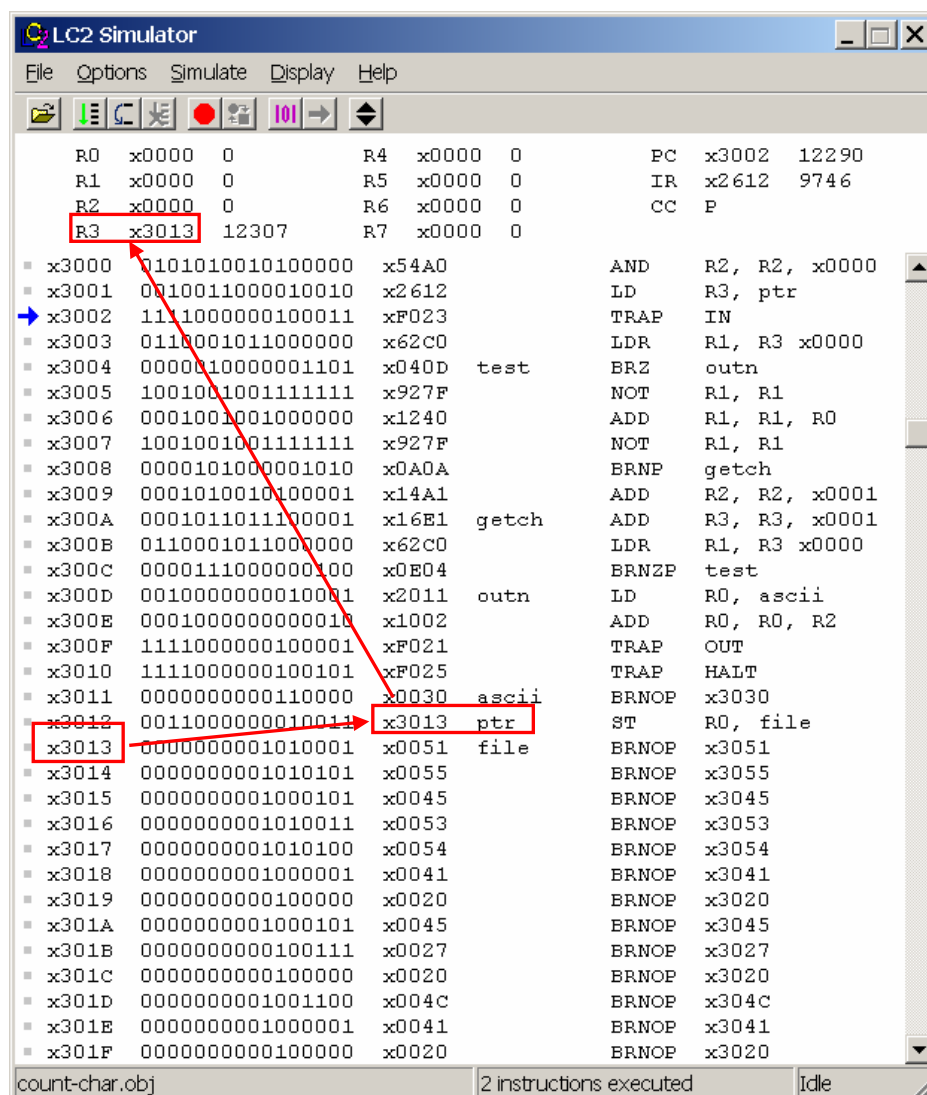
- **.orig x3000** ha segnalato all'*Assembler* che il programma deve essere caricato all'indirizzo x3000;
- la pseudo-istruzione **.fill x0030**, in corrispondenza della label **ascii**, ha riservato una cella di memoria contenente la costante x0030. Si ricorda che in codifica ASCII le cifre decimali partono dalla codifica x0030 (x0030 = '0', x0031 = '1', .. , x0039 = '9');
- la pseudo-istruzione **.fill file**, in corrispondenza della label **ptr**, ha riservato una cella di memoria contenente l'indirizzo del primo carattere della stringa memorizzata (che potrebbe quindi trovarsi ovunque nella memoria dell'LC-2);
- la pseudo-istruzione **.stringz**, in corrispondenza della label **file**, ha inizializzato una sequenza di celle di memoria con la codifica ASCII della frase "QUESTA E' LA STRINGA IN CUI CERCARE I CARATTERI", terminata dal valore 0.

La tabella dei simboli generata dall'Assembler è la seguente:

```
// Symbol table
// Scope level 0:
//      Symbol Name      Page Address
//      -----
//      ascii            3011
//      file              3013
//      getch             300A
//      outn              300D
//      ptr               3012
//      test              3004
```

Facciamo ora partire il programma con il comando *Simulate* → *Step (F8)* e analizziamo ogni singola istruzione.

Dopo la prima istruzione **AND R2, R2, x0000**, che inizializza al valore 0 il contenuto del registro R2, viene eseguita l'istruzione **LD R3, ptr**, che carica nel registro R3 l'indirizzo di partenza della stringa (x3013).



Quando viene eseguita la successiva istruzione di **TRAP IN** la console si pone in attesa di un carattere, che sarà il carattere che andremo a cercare all'interno della stringa memorizzata. La codifica ASCII di tale carattere viene salvata nel registro R0. Supponiamo di voler contare quante volte il carattere 'C' è presente nella frase "QUESTA E' LA STRINGA IN CUI CERCARE I CARATTERI".



L'istruzione `LDR R1, R3, x0000` legge la codifica ASCII del primo carattere della stringa e la salva nel registro R1. Il primo carattere letto è 'Q', pertanto la codifica ASCII è pari a `x0051`.

The screenshot shows the LC2 Simulator interface. At the top, there's a menu bar (File, Options, Simulate, Display, Help) and a toolbar. Below, the register file is displayed with registers R0 through R7, each with its current value and a label. R1 is highlighted with a red box and contains the value `x0051`. The instruction list below shows the current instruction at address `x3004`: `test` (BRZ outn). The instruction list also shows the previous instruction at `x3003`: `LDR R1, R3, x0000`. The status bar at the bottom indicates '4 instructions executed' and 'Idle'.

A questo punto inizia il ciclo del programma, identificato dalla label **test**: il ciclo terminerà quando verrà incontrato un valore nullo, che corrisponde al terminatore della stringa memorizzata. L'uscita dal ciclo è gestita dall'istruzione **BRZ outn**: fino a quando ci saranno caratteri da leggere, tale istruzione non porterà alla cella di memoria identificata dalla label **outn**.

Per confrontare fra loro due valori, in assenza di istruzione macchina specifica, si fa la somma in complemento a 2 e si vede se il risultato è nullo. Per operare tale somma, si può fare il complemento bit a bit del primo numero, sommare uno, quindi sommare il secondo numero e vedere se il risultato è 0; oppure si può fare il complemento bit a bit del primo numero, sommare direttamente il secondo numero e vedere se il risultato è -1 (ovvero `xFFFF`) cioè se il complemento bit a bit del risultato è 0. Gli autori del testo che introduce la struttura della CPU LC-2 hanno scelto questa seconda strada.

Partiamo dalle seguenti ipotesi:

- `R0 = x0043` → codifica ASCII carattere 'C';

- $R1 = x0051 \rightarrow$  codifica ASCII carattere 'Q'.

Il confronto viene effettuato dalla seguente serie di istruzioni:

- **NOT R1, R1**  $\rightarrow R1 = xFFAE = \#-82$
- **ADD R1, R1, R0**  $\rightarrow R1 = xFFF1 = \#-15$
- **NOT R1, R1**  $\rightarrow R1 = x000E = \#14$

Dopo queste tre istruzioni il contenuto di R1 NON è nullo, pertanto:

- l'istruzione **BRNP getch** effettua un salto all'istruzione identificata dalla label **getch**:
  - o **ADD R3, R3, x0001**  $\rightarrow$  R3 contiene ora l'indirizzo del carattere successivo della stringa;
  - o **LDR R1, R3, x0000**  $\rightarrow$  viene letto il nuovo carattere da confrontare;
  - o **BRNZP test**  $\rightarrow$  si ritorna all'inizio del ciclo.

Per contro, quando si incontra un carattere 'C' nella stringa, il confronto è il seguente:

- **NOT R1, R1**  $\rightarrow R1 = xFFBC = \#-68$
- **ADD R1, R1, R0**  $\rightarrow R1 = xFFFF = \#-1$
- **NOT R1, R1**  $\rightarrow R1 = x0000 = \#0$

Dopo queste tre istruzioni il contenuto di R1 è nullo, pertanto:

- l'istruzione **BRNP getch** NON effettua un salto all'istruzione identificata dalla label **getch**;
- **ADD R2, R2, x0001**  $\rightarrow$  il contatore delle occorrenze del carattere da cercare viene incrementato di uno;
- **ADD R3, R3, x0001**  $\rightarrow$  R3 contiene ora l'indirizzo del carattere successivo della stringa;
- **LDR R1, R3, x0000**  $\rightarrow$  viene letto in nuovo carattere da confrontare
- **BRNZP test**  $\rightarrow$  ritorno all'inizio del ciclo.

Al termine del ciclo, con l'ipotesi di aver cercato nella stringa il numero di occorrenze del carattere 'C', il registro R2 contiene il valore 4, ossia il carattere 'C' è stato trovato 4 volte all'interno della stringa memorizzata.

LC2 Simulator

File Options Simulate Display Help

R0 x0043 67 R4 x0000 0 PC x300D 12301  
 R1 x0000 0 R5 x0000 0 IR x040D 1037  
 R2 x0004 4 R6 x0000 0 CC Z  
 R3 x3042 12354 R7 x3003 12291

x3000	0101010010100000	x54A0	AND	R2, R2, x0000
x3001	0010011000010010	x2612	LD	R3, ptr
x3002	1111000000100011	xF023	TRAP	IN
x3003	0110001011000000	x62C0	LDR	R1, R3 x0000
x3004	0000010000001101	x040D	BRZ	outn
x3005	1001001001111111	x927F	NOT	R1, R1
x3006	0001001001000000	x1240	ADD	R1, R1, R0
x3007	1001001001111111	x927F	NOT	R1, R1
x3008	0000101000001010	x0A0A	BRNP	getch
x3009	0001010010100001	x14A1	ADD	R2, R2, x0001
x300A	0001011011100001	x16E1	ADD	R3, R3, x0001
x300B	0110001011000000	x62C0	LDR	R1, R3 x0000
x300C	0000111000000100	x0E04	BRNZP	test
x300D	0010000000010001	x2011	LD	R0, ascii
x300E	0001000000000010	x1002	ADD	R0, R0, R2
x300F	1111000000100001	xF021	TRAP	OUT
x3010	1111000000100101	xF025	TRAP	HALT
x3011	0000000000110000	x0030	BRNOP	x3030
x3012	0011000000001001	x3013	ST	R0, file
x3013	0000000001010001	x0051	BRNOP	x3051
x3014	0000000001010101	x0055	BRNOP	x3055
x3015	0000000001000101	x0045	BRNOP	x3045
x3016	0000000001010011	x0053	BRNOP	x3053
x3017	0000000001010100	x0054	BRNOP	x3054
x3018	0000000001000001	x0041	BRNOP	x3041
x3019	0000000001000000	x0020	BRNOP	x3020
x301A	0000000001000101	x0045	BRNOP	x3045
x301B	0000000001001111	x0027	BRNOP	x3027
x301C	0000000001000000	x0020	BRNOP	x3020
x301D	0000000001001100	x004C	BRNOP	x304C
x301E	0000000001000001	x0041	BRNOP	x3041
x301F	0000000001000000	x0020	BRNOP	x3020

count-char.obj 385 instructions executed Idle

Infine il programma termina con le seguenti istruzioni:

- **LD R0, ascii** → viene caricato in R0 il valore x0030, che rappresenta il contenuto della cella identificata dalla label **ascii**;
- **ADD R0, R0, R2** → si somma a R0 il numero di occorrenze del carattere cercato, in modo da identificare la corretta codifica ASCII del numero da visualizzare;
- **TRAP OUT** → viene visualizzato il risultato sulla Console;
- **TRAP HALT** → il programma viene arrestato.

Si noti che il programma qui presentato funziona solamente se il numero di occorrenze del carattere cercato è minore di 10, e può quindi essere espresso con una sola cifra decimale.

Una possibile evoluzione – peraltro piuttosto complessa – consiste quindi nel costruire un sottoprogramma capace di convertire un valore a 16 bit nei 4 caratteri corrispondenti alle 4 cifre esadecimali che rappresentano tale valore, o addirittura nelle 5 cifre decimali che rappresentano tale valore, magari eliminando gli zeri inutili.