Lezione 13: Il linguaggio assembly di LC-3

Laboratorio di Elementi di Architettura e Sistemi Operativi

30 Maggio 2012

Abbiamo visto...

- L'ISA di LC-3;
- Ogni istruzione è identificata da un insieme di 1 e di 0;
- Ogni locazione di memoria è individuata da un indirizzo a 16 bit.

Tuttavia...
scrivere programmi sottoforma di 1 e 0 non è certo il massimo. Vediamo quindi come rappresentare in modo più comprensibile i nostri programmi.

Ricorda...

- un algoritmo descrive con un linguaggio "naturale" un insieme di passi da fare per risolvere un problema;
- l'algoritmo viene trasformato in programma attraverso l'uso di un linguaggio di programmazione che può essere:
  - ad alto livello: C, JAVA, Pascal, COBOL, ecc;
  - ad un livello più vicino al linguaggio macchina: ASSEMBLY (ASM);
  - in linguaggio macchina (specifico per un'ISA).
- nei primi due casi il programma deve essere tradotto in un insieme di istruzioni specifiche per l'ISA.

Il linguaggio ASSEMBLY

Il linguaggio assembly di LC-3

- è un linguaggio a basso livello;
- ha lo scopo di rendere la programmazione più semplice rimanendo comunque vicino all'ISA;
- vi è una corrispondenza uno a uno tra istruzioni ASM e istruzioni specifiche dell'ISA;
- mette a disposizione nomi simbolici al posto degli opcode delle istruzioni; ad esempio ADD identifica l'opcode 0001;
- permette inoltre di definire nomi simbolici per le locazioni di memoria (symbolic address).
- Vediamo i dettagli del linguaggio ASM di LC-3 attraverso un esempio.

- Consideriamo il seguente programma scritto in C (quindi ad alto livello) che moltiplica per 6 l'intero memorizzato in NUMBER.
- Nota: abbiamo visto la volta scorsa che nell'ISA *non c'è la moltiplicazione*; occorre quindi creare un ciclo (for) che somma NUMBER a se stesso il numero di volte desiderate.

```
int main(){
 int NUMBER; // da inizializzare col valore desiderato
 int i;
 int r=0;
 for(i=6;i>0;i--){
   r=r+NUMBER;
 }
}
```

- Il suo corrispondente in assembly è:

```
; Programma che moltiplica un intero per la costante 6
; Prima dell'esecuzione, il numero da moltiplicare
; va memorizzato in NUMBER
;
        .ORIG   x3000
        LD      R1, SIX
        LD      R2, NUMBER
        AND     R3, R3, #0      ; Azzera R3. Al termine
                                ; conterrà il risultato
; Ciclo FOR
AGAIN   ADD     R3, R3, R2
        ADD     R1, R1, #-1     ; R1 è il contatore
        BRp     AGAIN           ; delle iterazioni
;
        HALT
;
NUMBER  .BLKW   1
SIX     .FILL   x0006
;
        .END
```

Le istruzioni

Istruzioni Assembly

- Il formato generale di un'istruzione assembly è:

```
LABEL    OPCODE    OPERANDS    ; COMMENTS
```

- LABEL: assegna un nome simbolico ad un indirizzo (che può contenere un'istruzione o un indirizzo di memoria). È opzionale;
- OPCODE e OPERANDS: sono specifici per ogni istruzione (come già visto la volta scorsa);
- COMMENTS: identifica un commento (come il comando // del C).
- Una label è un nome simbolico che può essere usato per identificare una zona di memoria;
- viene usata nel programma per fare un riferimento esplicito alla zona di memoria associata;
- in LC-3 una LABEL può essere lunga al massimo 20 caratteri.
- Esempio: la locazione *AGAIN* identifica l'inizio del ciclo, e viene usata nell'istruzione di salto "BRp AGAIN":
  - se il risultato dell'istruzione "ADD R1,R1,#-1" è positivo il programma "salta" alla locazione indicata da AGAIN ed esegue un'altra iterazione;

  - altrimenti si esce dal ciclo.

- Come già visto, un'istruzione ha un *OPCODE* e un certo numero di *OPERANDI*;
- l'OPCODE:
  - è un nome simbolico che corrisponde ad un'istruzione LC-3;
  - l'idea è che i nomi simbolici (ADD, AND, LDR) sono più facilmente ricordabili rispetto a 0001, 0101, 0110;
- gli OPERANDI: sono specifici per ogni istruzione; ad esempio:
  - LD R2, NUMBER carica il contenuto della zona di memoria indicato da NUMBER in R2;
  - AND R3,R3,#0 cancella il contenuto di R3;
  - Nota: nelle operazioni che richiedono valori costanti, come ad esempio l'istruzione appena vista, si utilizza *#* per indicare un decimale, *x* per indicare un esadecimale, e *b* per indicare un numero binario.
- Nel linguaggio assembly i commenti sono indicati dal **";"**
- contengono messaggi in linguaggio naturale che non hanno effetti sul processo di esecuzione;
- ciò che segue il *;* viene completamente ignorato.

Pseudo direttive assembly

- Per scrivere un programma in assembly sono necessarie alcune *pseudo direttive* (chiamate anche *pseudo-ops*);
- esse non sono vere e proprie istruzioni da eseguire, ma...

servono in fase di traduzione da ASM a linguaggio macchina per far si che il traduttore interpreti correttamente il contenuto del programma.

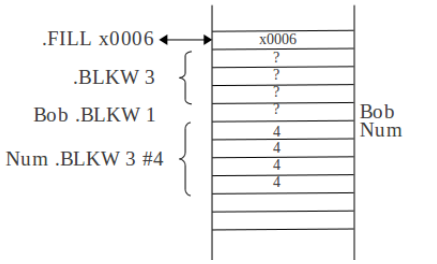
- si riconoscono perché iniziano con il **"."** e, ovviamente, sono nomi riservati.

Le pseudo-direttive del linguaggio assembly di LC-3 sono:

- **.ORIG**: indica l'indirizzo di partenza del programma LC-3; ad esempio alla riga 05 viene detto che il programma deve iniziare all'indirizzo (esadecimale su 16 bit) x3050;
- **.END**: indica dove finisce il programma. Attenzione: questa istruzione non dice che il programma deve essere terminato, ma l'indirizzo a cui terminano le sue istruzioni!
- **.FILL**: indica che la prossima locazione di memoria contiene il valore indicato dall'operando; ad esempio alla riga 13 viene detto che la nona locazione (a partire dall'inizio del programma) contiene il valore x0006;

- **.BLKW**: indica di riservare una sequenza (contigua), lunga tanto quanto indicato dall'operando, di locazioni di memoria.

  Si possono anche inizializzare;




```
        .ORIG     x3010
HELLO   .STRINGZ  "Hello, World!"
```

```
x3010: x0048
x3011: x0065
x3012: x006C
x3013: x006C
x3014: x006F
x3015: x002C
x3016: x0020
x3017: x0057
x3018: x006F
x3019: x0072
x301A: x006C
x301B: x0064
x301C: x0021
x301D: x0000
```

- **.STRINGZ**: indica di iniziare una sequenza lunga n+1 di locazioni di memoria. L'argomento è una sequenza di n caratteri compresa fra i doppi apici.

  I caratteri vengono rappresentati su 16bit.

  Il carattere terminatore `'\0'` è sottointeso.

Da codice assembly a codice macchina

L'assemblatore

- Abbiamo visto che il linguaggio assembly permette di scrivere programmi a più alto livello rispetto le istruzioni macchina;
- tuttavia un'architettura è in grado di eseguire solo istruzioni in linguaggio macchina: *occorre quindi una traduzione*;
- questo processo di traduzione viene fatto dall'**ASSEMBLATORE**.
- In generale, nel linguaggio assembly vi è una corrispondenza 1 a 1 fra le istruzioni ASM e le istruzioni macchina;
- si può quindi pensare di prendere il programma scritto in ASM e, riga per riga, determinare la corrispondente istruzione macchina.
- Esempio:

```
01  .ORIG x3050
02  AND R3,R3,#0 ; reset di R2
03  LD R1, SIX
...
```

```
01  x3050:    0101011011100000
...
```

- *PROBLEMA: cosa succede quando si arriva alla riga 03?*
- ***Alla riga 03 l'assembler NON SA ANCORA A COSA SI RIFERISCE "SIX"!!!***

Soluzione:
Occorre un processo di traduzione in due fasi:

1. creare la tabella dei simboli;
2. generare il programma in linguaggio macchina.

Il processo di traduzione

1. Creazione della tabella dei simboli:
   - La tabella dei simboli contiene la corrispondenza fra i nomi simbolici (ad esempio AGAIN) e gli indirizzi di memoria a 16 bit;
   - si scorre l'intero programma fra le istruzioni ".ORIG" e ".END" tenendo traccia dell'indirizzo delle istruzioni, e inserendo nella tabella dei simboli le LABEL ed il relativo indirizzo;
   - per far questo si usa una locazione chiamata Location Counter (LC), inizializzato al valore di ".ORIG".
   - Esempio:

SIMBOLO	INDIRIZZO
AGAIN	x3053
NUMBER	x3056
SIX	x3057

2. Generazione del programma in linguaggio macchina:
   - Si rianalizza riga per riga il programma assembly e con l'ausilio della tabella dei simboli si traduce ogni istruzione nel linguaggio macchina di LC-3;
   - si utilizza ancora una volta LC per determinare gli indirizzi;
   - quando si raggiunge ".END" il processo termina, e il risultato è un file binario corrispondente al programma scritto.

I tool di LC-3

La traduzione va fatta a mano?
No! Esistono tool che ci aiutano a fare questa traduzione. Ad esempio vedremo *lc3as*, il quale a partire da un file ".asm" genera due file: un file oggetto *.obj* contenente il programma in linguaggio macchina, e un file *.sym* contenente la tabella dei simboli generata dopo la prima analisi del file sorgente.

Come testare il nostro programma?
Attualmente LC-3 non esiste, ma esiste un simulatore che simula l'ISA di LC-3 sfruttando l'ISA si un'architettura x86 o x64. Noi useremo la versione per linux chiamata *lc3sim-tk*.

Esercitazione

Esempio:
Vogliamo realizzare un programma che sommi tra loro dieci numeri contenuti in memoria e ponga il risultato nel registro R1.

- Quello che dobbiamo fare è:
  - creare il nostro programma assembly, che chiameremo "addnums.asm";
  - memorizzare i dati in memoria.

Creazione del programma assembly

- Il primo passo è quello di creare il nostro programma assembly;
- utilizzando un qualsiasi editor di testo, creiamo il file "addnums.asm" e scriviamo il nostro programma:

```
      .ORIG x3000
      AND R1,R1,x0  ; cancella R1 usato per memorizzare la somma
      AND R4,R4,x0  ; cancella R4 usato come contatore
      ADD R4,R4,xA  ; carica in R4 il valore 10 (contatore)
      LEA R2,DATA   ; mette in R2 l'indirizzo di partenza dei dati
LOOP  LDR R3,R2,x0  ; carica il numero da aggiungere
      ADD R2,R2,x1  ; incrementa il puntatore
      ADD R1,R1,R3  ; aggiunge il numero corrente alla somma
      ADD R4,R4,x-1 ; decrementa il contatore
      BRp LOOP      ; se il contatore non è zero torna a LOOP
      HALT
DATA  .BLKW 10      ; riserva 10 celle per i numeri da sommare
      .END
```

Traduzione del programma in linguaggio macchina

- Il prossimo passo consiste nell'invocare l'*assembler* per tradurre il nostro programma assembly in programma macchina.
- Utilizziamo il comando:

```
lc3as addnums.asm
```

- Otteniamo così due file:
  - il nostro programma oggetto: *addnums.obj*;
  - la tabella dei simboli: *addnums.sym*:

```
// Symbol table
// Scope level 0:
//     Symbol Name      Page Address
//     ----------------  ------------
//     LOOP            3004
//     DATA            300A
```

Inserimento dei dati in memoria

- Il prossimo passo da fare è mettere i dati in memoria.
- Ci sono tre modi:
  - a mano, cliccando sulla cella di memoria e inserendo il valore nel campo "Value"; SCOMODO.
  - inserendo i dati direttamente nel codice assembly con la direttiva .FILL
  - attraverso un file di testo che verrà passato al simulatore.
- Useremo il secondo metodo. Modifichiamo la parte finale del file `addnums.asm`:

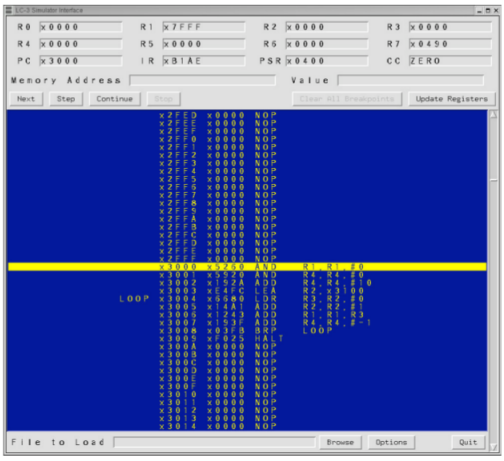
```
DATA  .FILL x0001  ; riserva 10 celle per i numeri da sommare
      .FILL x0002
      .FILL x0003
      .FILL x0004
      .FILL x0005
      .FILL x0006
      .FILL x0007
      .FILL x0008
      .FILL x0009
      .FILL x000A
      .END
```

- e assembliamo nuovamente il programma:

```
lc3as addnums.asm
```

Il simulatore

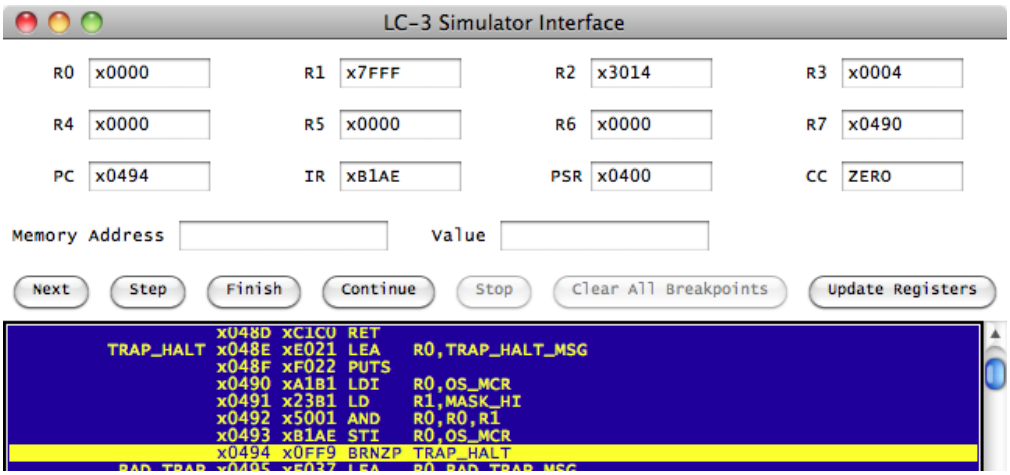
- A questo punto possiamo lanciare il simulatore con il comando ***lc3sim-tk***.



- Il simulatore è composto da 4 zone principali:
  - in alto troviamo gli 8 registri general-purpose, il PC, l'IR, e CC (condition codes);
  - segue un'area in cui possiamo visualizzare/settare gli indirizzi di memoria;
  - l'area blu che visualizza tutti gli indirizzi della macchina simulata. In tutto sono 2^{16}=65536 zone di memoria; per visualizzare un indirizzo specifico basta scriverlo nel campo "Memory Address" e dare invio.
  - in basso abbiamo il campo "File to Load" che ci permette di caricare dei file esterni.
- Possiamo quindi caricare il nostro programma cliccando su ″Browse", e notiamo che esso viene caricato a partire dall'indirizzo x3000.
- Per visualizzarlo usiamo il campo "Memory Address".
- Dopo esserci assicurati che il PC contenga il valore x3000 possiamo eseguire il nostro programma passo passo cliccando su *"Next"*.
- Prima dell'esecuzione dell'istruzione HALT troveremo la somma calcolata nel registro R1.

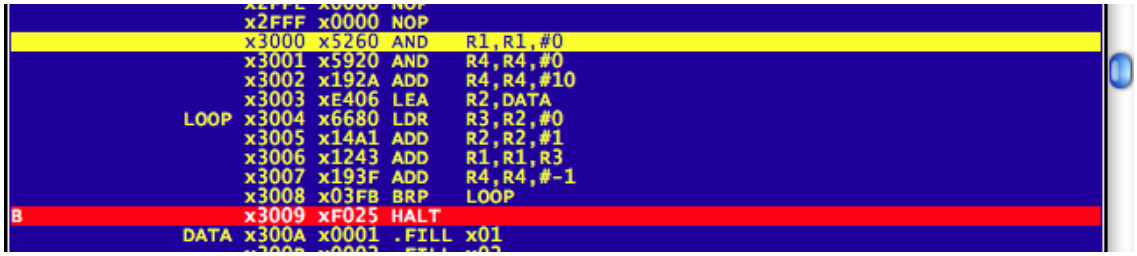
I breakpoint

- Dopo aver resettato il simulatore e ricaricato, proviamo ad eseguire il programma in una volta sola cliccando su *"Continue"*:

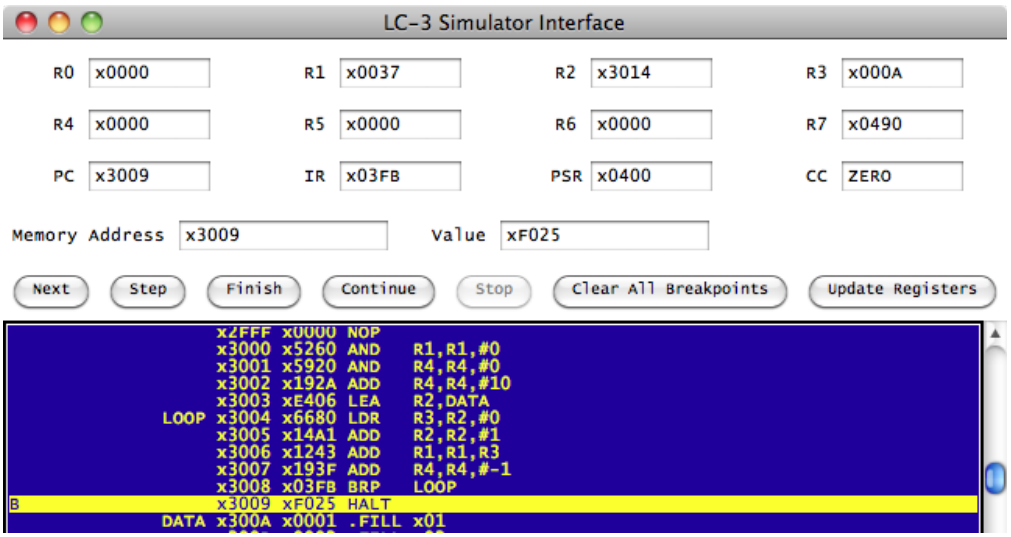


Il risultato non è quello atteso!
L'istruzione HALT fa saltare ad una zona di memoria riservata, e modifica il contenuto dei registri

- Per poter interrompere l'esecuzione del programma *prima dell'esecuzione dell'istruzione HALT* si usano i *breakpoint*.
- Resettiamo nuovamente il simulatore e facciamo un doppio click sulla locazione di memoria che contiene l'istruzione HALT:



- La B all'inizio della linea identifica il breakpoint:
  - forza il simulatore a fermarsi *prima* dell'esecuzione dell'istruzione HALT
- Ora il comando *"Continue"* ci consente di vedere il risultato del calcolo:



- x0037 è l'esadecimale per 55: la somma dei numeri da 1 a 10
  - controllate usando la calcolatrice in modalità *Programmazione*