Nozioni di assembly utilizzate in alcune lezioni del corso

Prof. Mauro Negri

Dipartimento di Elettronica e Informazione (D.E.I.)

1 settembre 2012

Indice

1. Istruzioni e modalità di indirizzamento

1.1 Istruzioni

Le istruzioni che saranno utilizzate non corrispondono a quelle di alcun assembly reale, ma ad una loro estrema semplificazione al fine di introdurre il minimo numero di istruzioni sufficienti a rendere più chiari alcuni aspetti della programmazione strutturata.

Le istruzioni utilizzate sono le seguenti (il significato delle istruzioni è autoesplicativo):

[W:] ADD oper1,oper2	oper1 + oper2 in oper2
[W:] MOV oper1, oper2	copia oper1 in oper2
[W:] BR I	passa ad eseguire l'istruzione alla parola con etichetta I
[W:] BREQ oper, I:	se oper =0 allora SALTA a istruzione con etichetta I
[W:] JTS I	eseguire funzione che inizia all'etichetta I
[W:] RTS	ritorno da funzione
[W:] EXIT	termina esecuzione programma
[W:] READ oper:	carica valore letto da tastiera in oper
[W:] WRITE oper:	scrive su video valore in oper

L'etichetta opzionale W permette di identificare una riga del programma. Si supponga che le istruzioni READ e WRITE trattino direttamente valori del tipo di interesse (intero, reale, stringa,…) e non si devono quindi considerare problemi di conversione tra un tipo e la sequenza di caratteri.

1.2 I modi di indirizzamento

I modi di indirizzamento utilizzabili per specificare gli operandi utilizzati nelle istruzioni sono indicati nel seguito (la notazione Ri indica un registro generale della macchina, con $0 \leq i \leq 7$ R7 equivale anche a SP: stack pointer). il simbolo X può anche essere un numero.

Modo di indirizzamento	notazione simbolica	spiegazione
Registro	Ri	L'operando è il contenuto del registro Ri
Registro indiretto	(Ri)	L'operando è il contenuto della parola di memoria il cui indirizzo è contenuto in Ri
Immediato	#X	L'operando è X
Relativo	X	L'operando è il contenuto della parola di memoria di indirizzo X

Esempio

Si ipotizzi i seguenti contenuti:

R1contenga 100
R2 contenga 200
X = 50

Memoria (stato iniziale)

Indirizzo	Contenuto
50	153
99	106
100	102
150	152
152	500
153	600

Esito delle istruzioni

MOV R1, R2	R2= 100
MOV (R1), R2	R2= 102
MOV #X, R2	R2= 50
MOV X, R2	R2 = 153
MOV #99, R2	R2 = 99
MOV 99, R2	R2 = 106
MOV X, 99	copia 153 nella parola di memoria di indirizzo 99

1.3 Direttive o pseudo istruzione

Le direttive più importanti sono quelle utilizzate per riservare parole di memorie

[W:] .RES N alloca N parole di memoria consecutive e associa l'etichetta X alla prima parola

END [X] fine programma e riporta l'etichetta della prima istruzione del programma

La direttiva .RES permette di riservare da 1 parola (variabile semplice) a N parole (vettore).

```
MIAVAR: .RES 1
MIOVETTORE: .RES 100
```

Esempio di algoritmo

```
int X,Y,Z
Z=0
X=0?  ->(true) Scrivi Z; fine
      ->(false)  Z=Z+Y; X=X-1;
```

Programma assembly

```
etichette   direttive/istruzioni
X:              RES 1
Y:              RES 1
Z:              RES 1
START:          READ X
                READ Y
```

```
        MOV #0, Z
CICLO:  BREQ X, FINE
        ADD Y, Z
        ADD #-1, X
        BR CICLO
FINE: WRITE Z
        EXIT
        END START (prima istruzione eseguibile)
```

2. Traduzione dal C all'assembly

In questa parte si vedrà la traduzione di alcune strutture del linguaggio C nel linguaggio assembly.

1. In questo capitolo non ci occuperemo di come il compilatore esegua la traduzione ma ci limiteremo a mostrare come è fatto il risultato della traduzione stessa; la realizzazione di un compilatore costituisce un argomento che esula dagli scopi di questo testo.
2. I compilatori generano normalmente moduli oggetto binari; noi invece descriveremo il risultato della compilazione usando il linguaggio assembly, per godere di una buona leggibilità e avere un programma che è sostanzialmente assai simile a uno binario.
3. L'importanza data ai vari argomenti trattati non è proporzionale alla loro importanza per il programmatore C, ma solo alla loro importanza rispetto agli aspetti dell'architettura dei sistemi di elaborazione che si vogliono analizzare.

2.1 Vettori e record

int V[100]; implica V: .RES 100 dove V rappresenta l'indirizzo della parola di memoria di indirizzo più basso

Per indirizzare un elemento ad esempio, V[j] sarà necessario allocare una parola per memorizzare la variabile j (ad esempio j: .RES 1) e poi dopo aver caricato un valore nella variabile J indirizzare l'opportuno elemento del vettore. Pertanto l'istruzione C V[j]=3; implica ad esempio la seguente sequenza di istruzioni

```
MOV  #V, R1        carica in R1 l'indirizzo di V[0]
ADD  j, R1  somma la distanza del j-esimo elemento dal primo elemento del vettore
MOV  #3, (R1) carica 3 nella parola puntata da R1
```

Nel caso dei record si alloca memoria per ogni elemento del record assegnando come nome dell'etichetta il nome del campo del record; se a sua volta un campo è un vettore si memorizza poi il campo come precedentemente descritto.

2.2 I sottoprogrammi

Se una stessa porzione di codice deve essere eseguito più volte in diversi punti del programma può essere utile poter scrivere tale porzione di codice una sola volta, eseguendo un salto alla prima istruzione di tale codice ogni volta che risulta necessario. A differenza di un salto normale, in questo caso si vuole essere in grado, dopo aver eseguito la porzione di codice desiderata, di saltare "indietro" e riprendere l'esecuzione del programma nel punto dal quale si era partiti.

A questo scopo servono le due istruzioni JTS (jump to subroutine) che permette di saltare alla prima istruzione del sottoprogramma e RTS (return da subroutine) che permette di saltare indietro quando il sottoprogramma è terminato.

L'istruzione JTS ha come unico operando l'etichetta della prima istruzione del sottoprogramma cui deve saltare. L'istruzione RTS non ha operandi e deve determinare in un modo spiegato nel seguito l'indirizzo di ritorno che cambia di volta in volta; ciò è possibile perché JTS oltre a iniziare l'esecuzione del sottoprogramma deposita in una locazione di memoria concordata l'indirizzo dell'istruzione che segue l'esecuzione del sottoprogramma e RTS preleva tale indirizzo e lo utilizza come destinazione del salto di ritorno. La locazione di memoria che contiene l'indirizzo di ritorno è una parte della memoria chiamata per semplicità pila (stack) di sistema il cui puntatore è il registro SP (R7) e usata da ogni programma in esecuzione sul calcolatore. A questo livello è compito del programmatore di riservare un'area di memoria per contenere tale pila nel proprio programma e di inizializzare il registro SP in modo che punti alla coda della pila, ossia alla parola della pila di indirizzo più grande, prima di eseguire una JTS.

Le operazioni elementari eseguite da JTS SUB sono:

- Copia nella parola indirizzata da SP il contenuto del registro PC (program counter) incrementato di un'unità (indirizzo dell'istruzione successiva alla JTS);
- Incrementa di un'unita il valore del registro SP
- Carica in PC l'indirizzo equivalente all'etichetta SUB

Le operazioni elementari eseguite da RTS sono:

- Decrementa il valore di SP
- Preleva il contenuto della parola indirizzata da SP e lo carica in PC

Il precedente meccanismo impone al programmatore di utilizzare con attenzione il registro SP.

Il meccanismo di base non sarebbe molto utile se non fosse possibile che il sottoprogramma operi ad ogni esecuzione su dati diversi (parametri). Il passaggio parametri di linguaggi come il C non è disponibile in assembly e quindi deve essere simulato attraverso una serie di operazioni che devono essere eseguite prima della JTS (dati in ingresso) e prima o dopo l'esecuzione della RTS come sarà descritto più avanti. In particolare sarà illustrato come avvenga il passaggio parametri con il sottoprogramma utilizzando la stessa pila di sistema nel quale è caricato l'indirizzo di ritorno.

Come detto in precedenza un sottoprogramma assembly utilizza una pila di sistema per memorizzare l'indirizzo di ritorno. Ora questa azione viene arricchita con altre operazioni che permettono la traduzione di una funzione C in assembly. In particolare, oltre alle operazioni preliminari di allocazione della pila e inizializzazione di SP si devono eseguire le seguenti operazioni all'atto dell'invocazione della funzione C:

1. Viene riservata una parola di memoria senza contenuto allo scopo di contenere il valore ritornato dalla funzione (si ipotizzi che sia sempre utilizzata anche nel caso di tipo ritornato void).
2. I parametri attuali della funzione C vengono accoppiati ordinatamente con i parametri formali della funzione che devono essere allocati sulla pila. Per ogni parametro passato per valore si riserverà lo spazio necessario per contenere il valore passato. Per ogni parametro passato per indirizzo si riserverà lo spazio per contenere l'indirizzo della parola di memoria che dovrà essere modificata dalla funzione.
3. Viene caricato il valore di ritorno dalla JTS.
4. Per ogni variabile locale dovrà essere riservato spazio sulla pila per contenerne il valore generato dalla funzione.

Terminata l'esecuzione della funzione tutte le parole di memoria allocate nella pila per contenere dati locali della funzione e parametri dovranno essere deallocate, ad eccezione della parola di memoria che contiene il valore ritornato dalla funzione. L'istruzione RTS provvederà ad estrarre l'indirizzo di ritorno e ad eliminarlo dalla pila. Infine il programma chiamante concluderà l'esecuzione della funzione prelevando e consumando il valore prodotto dalla funzione in base a quanto previsto dal programma e ad eliminarlo dalla pila che si ritrova nello stato iniziale che aveva prima dell'invocazione della funzione.

Dato che una funzione può invocare se stessa in modo ricorsivo, è necessario un meccanismo che permetta di allocare una nuova area dati a una nuova funzione che viene invocata, mentre una precedente invocazione non è ancora terminata; ciò significa che non si possa adottare un modello statico che allochi un'area di memoria per ogni funzione definita all'atto della compilazione. Pertanto è necessaria un'allocazione dinamica della memoria, ossia che avviene durante l'esecuzione del programma.

Questo è il motivo per cui l'area di memoria necessaria ad una funzione viene allocata in un'area dinamica chiamata STACK e gestita col meccanismo a pila (last in first out) perché le funzioni terminano la loro esecuzione nell'ordine inverso a quello in cui si attivano.

Lo stack pertanto conterrà in un certo istante di tempo tutte le aree create dalle funzioni che sono attualmente attive. L'area di memoria dedicata ad ogni funzione è chiamata **record di attivazione (RDA)**. Dato che il RDA viene allocato all'atto dell'esecuzione di una funzione, il compilatore non può conoscere a priori il suo indirizzo nella fase di compilazione. Per tradurre le istruzioni che richiedono di accedere a dati che sono nell'RDA si ricorre pertanto ad un meccanismo misto: il compilatore all'attivazione della funzione carica in un registro chiamato base locale (R0 in questo contesto) il valore dello

SP in modo da conservare l'inizio del RDA nello stack e poi identifica le singole parole dell'RDA, utilizzando la distanza relativa che esse hanno dall'indirizzo conservato nel registro R0. Si noti che il registro R0 è quindi dedicato a questo scopo e non può essere mai modificato nel contenuto per latri fini. Non si usa SP poiché il suo valore cambia ad esempio se la funzione corrente attiva un'altra funzione e quindi non può costituire un riferimento fisso per tutte le funzioni; inoltre lo stack è usato anche per altri scopi, ad esempio la valutazione delle espressioni (non trattata), impedendo allo SP di svolgere questo ruolo. Il registro base locale invece contiene un valore fisso durante l'esecuzione della funzione e quindi le distanze delle singole parole di memoria dall'indirizzo contenuto in R0 sono costanti.

Dato che durante l'esecuzione di una funzione può essere attivata un'altra funzione, che a sua volta utilizzerà R0 come base locale, è necessario che ogni funzione salvi il contenuto del registro R0 prima di utilizzarlo come propria base locale e rispristini il contenuto precedente prima di ritornare al chiamante. Pertanto, nell'RDA di una funzione deve essere previsto anche lo spazio per il salvataggio di R0. Vediamo ora come il record di attivazione di una procedura possa venire costruito all'attivazione della funzione ed eliminato alla terminazione.

La costruzione del record di attivazione è in parte compito del chiamante e in parte compito della funzione chiamata.

Le operazioni che il chiamante deve fare al momento dell'attivazione della funzione sono:

1. Riservare lo spazio sulla pila per il risultato
2. Caricare i parametri sulla pila di sistema.
3. Eseguire un'istruzione JTS che salva sulla pila l'indirizzo di ritorno e inizia l'esecuzione della funzione.

Le istruzioni che svolgono queste operazioni sono quindi generate dal compilatore in corrispondenza dell'istruzione C di invocazione della procedura e costituiscono la *sequenza di chiamata* della funzione.

Le operazioni che la funzione deve fare subito, appena inizia la propria esecuzione, e che terminano la costruzione del record di attivazione, sono:

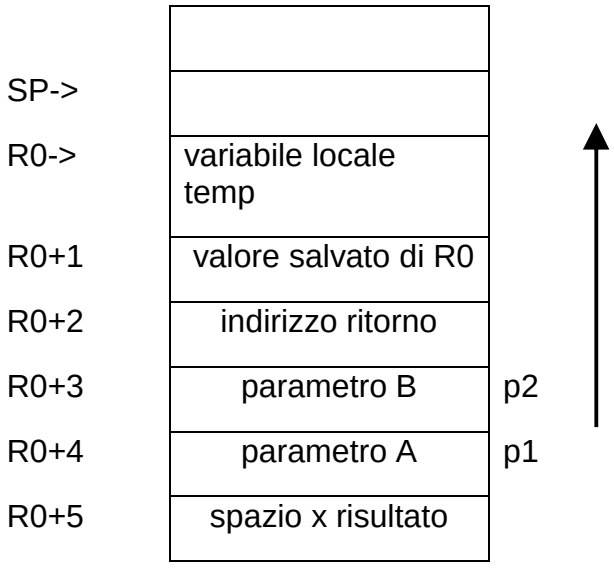
4. Salvare il valore del registro R0 sulla pila
5. Caricare il valore di SP in R0 (in questo modo R0 diventa base locale della funzione, assumendo un indirizzo fisso rispetto agli altri elementi del record di attivazione).
6. Allocare lo spazio per le variabili locali sulla pila incrementando opportunamente SP.

La sequenza di istruzioni che esegue le operazioni 4-6 è generata dal compilatore all'inizio del codice della funzione; tale sequenza è detta *prologo* della funzione.

Dopo l'esecuzione di tutte le operazioni 1-6, il record di attivazione è completamente costruito. Nella figura seguente sono mostrati un programma C che definisce e invoca una funzione per il calcolo di una somma e il record di attivazione della funzione.

```
int A, B, C;
int sum(int p1, int p2)
3.   {int temp;
4.    temp = p1+ p2;
5.   return(temp);
   }
void main ()
  1.    {A=2; B=3;
  2.     C = sum(A,B);
```

// RDA alla massima estensione



La figura mostra anche gli indirizzi di tutti gli elementi del record di attivazione con riferimento alla base locale R0.

Alla fine dell'esecuzione della procedura devono essere eliminati dalla pila gli elementi del record di attivazione; ciò viene fatto dalla funzione stessa fino all'indirizzo di ritorno, che viene eliminato dall'Istruzione RTS, poi dal chiamante, che deve eliminare i parametri.

Nella figura seguente è riportata prima la traduzione in Assembly delle istruzioni del programma C che invoca la funzione SUM e poi la traduzione della funzione.

Traduzione del main

```
A: .RES 1                    allocazione statica delle 3 variabili globali
B: .RES 1
```

```
C: .RES 1
STACK: .RES 1000                allocazione dello stack
                                START etichetta la prima istruzione eseguibile del main
START:   MOV #STACK, SP
         ADD #999, SP                   inizializzazione SP
         MOV #2, A              1)
         MOV #3, B              1)
                                invocazione della funzione sum
         ADD #-1, SP                        2) spazio RDA per il risultato
         MOV A, (SP)        ADD #-1,SP          2) spazio RDA per parametri
         MOV B, (SP)    ADD #-1,SP          2)
         JTS SUM                        2) invocazione sum
```

Stato dello stack all'atto della Invocazione di sum

SP->

valore di B
valore di A
risultato

```
                                quando la funzione ha eseguito RTS
RET:      ADD #2, SP                2) elimina parametri da RDA
          ADD #1, SP                2) risultato ritornato in C e
          MOV (SP), C               2) elimina spazio risultato
            EXIT
.END START
```

Osservazioni:
L'istruzione JTS carica l'indirizzo di ritorno nello stack e modifica il PC, eseguendo le seguenti operazioni descritte in assembly:

```
MOV #RET, (SP)
ADD #-1, SP
MOV #SUM, PC
```

Traduzione della funzione

```
                                prologo della funzione
sum: MOV R0, (SP)      ADD #-1, SP    salva registro  R0
       MOV SP, R0                      carica valore di SP in R0
       ADD #-1, SP                     3)spazio RDA per variabile locale
                                       Seguono le istruzioni della funzione: temp=p1+p2
                                       è scomposta in temp=p1; temp=temp+p2;
        MOV R0, R1                     4) R1 = indirizzo di riferimento
        ADD  #4, R1                    4) R1 contiene l'indirizzo assoluto di p1
        MOV (R1), (R0)                 4) temp=p1
        MOV R0, R1                     4) R1 = indirizzo di riferimento
        ADD  #3, R1                    4) R1 contiene l'indirizzo assoluto di p2
        ADD (R1), (R0)                 4)temp=temp+p2


                                return temp  1) spostamento risultato della funzione
        MOV R0, R1                     5) R1 = indirizzo di riferimento
        ADD   #5, R1                   5) R1 contiene l'indirizzo assoluto di risfun
```

```
        MOV (R0), (R1)          5) risfun = temp

                                return temp  2) riattiva la funzione chiamante main
        ADD #1, SP              5)      elimina var locale da RDA
        ADD #1, SP              5)      ripristina R0 togliendolo da RDA
        MOV (SP), R0            5)
        RTS                     5)
```

Osservazione:
La funzione RTS preleva dall'RDA l'indirizzo di ritorno e inizializza il PC, eseguendo le seguenti operazioni descritte in Assembly

```
        ADD#1, SP
        MOV (SP), PC)
```

Osservazioni:

- l'esempio ha utilizzato il passaggio dei parametri per valore, tuttavia il passaggio per indirizzo non richiede di modificare lo schema adottato; basta copiare nel RDA l'indirizzo del parametro attuale e poi nelle istruzioni della funzione utilizzare tale indirizzo per raggiungere la variabile puntata dal parametro (esempi sono disponibili nei temi d'esame risolti).
- I meccanismi per arricchire l'RDA al fine di gestire le regole del campo di validità, che permettono ad esempio, ad una funzione di usare direttamente nelle proprie istruzioni una variabile globale non sono trattati.

Nel caso di funzioni ricorsive si adotta sempre il meccanismo descritto, ma con l'accortezza di considerare le invocazioni successive la prima come operazioni che preleveranno i parametri dallo stesso stack e non dalle variabili globali.
A titolo di esempio si consideri il seguente programma C per il calcolo ricorsivo del fattoriale:

```
int N,R;
int fatt(int n)
{if (n == 0)
        return(1);
  else return (n * fatt(n-1);}

void main()
        {printf("\nvalore di n: "); scanf("%d", &N);
          R= fatt(N);
          printf("il fattoriale di %d è %d", N, R);
        }
```

Il programma è tradotto nel seguente modo (omesse le printf e scanf):

```
N: .RES 1
R: . RES 1
START ADD #-1, SP                                       R=fatt(N)
        MOV N, (SP)  ADD #-1, SP
        JSR fatt
ret:     ADD #+1, SP
        ADD #+1, SP  MOV (SP), R
        EXIT
.END START
```

```
fatt:   MOV R0, (SP)        ADD #-1, SP                 fatt (n)
        MOV SP, R0 ...
        MOV R0, R1  ADD #3, R1
        BREQ (R1), zero

ric:    MOV R0, R1
        ADD #3, R1
        MOV (R1), R1                    n in R1
        ADD #-1, SP
        MOV R0, R3
        ADD  #3, R3
        MOV (R3), (SP)
        ADD #-1, (SP)
        ADD #-1, SP                     n-1 nello stack
        JSR fatt
Ret1:ADD #1, SP
        ADD #1, SP
        MOV (SP), R2                    valore di fatt(n-1)

        //R2=R2*R1              n*fatt(n-1)
        MOV R0,R1
        ADD#4, R1
        MOV R2, (R1)
        BR fine

zero:   MOV R0, R1
        ADD #4, R1
        MOV #1, (R1)

Fine:   ADD #1, SP
        MOV (SP), (R0)
        RTS
```