Programmare in assembly in GNU/Linux con sintassi AT&T

Fulvio Ferroni *fulvioferroni@teletu.it*

2011.08.05

Questo documento intende dare le basi essenziali per la programmazione assembly in ambiente GNU/Linux con l'assemblatore GAS (sintassi AT&T); viene anche trattato l'uso di porzioni di codice assembly all'interno di programmi scritti in '`c`' (assembly inline).

Indice generale

Premessa

In queste dispense viene esaminata la programmazione assembly in ambiente GNU/Linux con l'uso dell'assemblatore GAS (*GNU Assembler*).

La trattazione non ha alcuna pretesa di completezza ma ha scopi prettamente didattici, come supporto per le attività pratiche nella disciplina «sistemi informatici» nelle classi terze degli ITIS ad indirizzo informatico.

Sono prerequisiti indispensabili per l'uso di queste dispense:

1. conoscenza dei sistemi di numerazione binario, ottale, esadecimale e delle modalità di rappresentazione dei dati, numerici e non, all'interno del sistema di elaborazione;
2. conoscenza dell'algebra di Boole e degli operatori logici;
3. conoscenza della struttura e del funzionamento del sistema di elaborazione (macchina di Von Neumann), del PC e delle sue periferiche;
4. conoscenza teorica delle fasi che portano alla traduzione ed esecuzione di un programma scritto in un generico linguaggio assembly;
5. conoscenza dei principi di base della programmazione e capacità di scrivere semplici programmi in linguaggio '`c`';
6. capacità di utilizzo della piattaforma GNU/Linux, soprattutto nella modalità a «linea di comando».

Riguardo al punto 1 si possono consultare le dispense dal titolo: «Rappresentazione dei dati nell'elaboratore»; per il punto 2 ci sono le dispense dal titolo: «Algebra di Boole e reti logiche» mentre per i punti 3 e 4 sono disponibili le dispense dal titolo: «Architettura dei sistemi di elaborazione, PC e linguaggi a basso livello».

Tutto questo materiale è opera dello stesso autore delle presenti dispense ed è reperibile all'indirizzo ⟨*http://www.maxplanck.it/materiali*⟩ selezionando la scelta «VEDI MATERIALE» e cercando poi in base al docente scegliendo il nome ***Ferroni Fulvio***.

Riguardo ai punti 5 e 6 la quantità di documentazione disponibile è notevole sia in forma cartacea che in rete; citiamo solo una delle opere più complete: gli «Appunti di informatica libera» di Daniele Giacomini, disponibile all'indirizzo ⟨*http://a2.swlibero.org*⟩.

Capitolo 1

Richiami sulle caratteristiche dei processori x86

Pur ribadendo che la conoscenza delle caratteristiche dei processori Intel x86 (e compatibili) è da considerare un prerequisito per l'uso di queste dispense, in questo capitolo riassumiamo le loro caratteristiche fondamentali.

1.1 Architettura dell'8086

Il processore 8086 ha le seguenti peculiarità:

- gestisce la memoria in modo segmentato e con ordinamento *little endian*;
- gestisce le periferiche con l''`I/O isolato`' e possiede quindi, nel suo repertorio, istruzioni dedicate al trasferimento dei dati (***in*** e ***out***);
- prevede istruzioni a due indirizzi (con qualche eccezione riguardante ad esempio le moltiplicazioni e le divisioni); sono ammesse tutte le combinazioni nei due operandi tra locazioni di memoria, registri e valori immediati eccetto la combinazione «memoria-memoria»
- è '`non ortogonale`' in quanto i registri sono specializzati.

Essendo la gestione della memoria segmentata, gli indirizzi si indicano nella forma: ***regseg:offset***, cioè: *indirizzo di segmento : indirizzo della cella all'interno di quel segmento*, ricordando che i valori si esprimono preferibilmente in esadecimale.

Ricordiamo inoltre che la memoria gestita è di 1 MB suddivisi in 65.536 segmenti da 64 KB ciascuno, parzialmente sovrapposti (ognuno inizia 16 locazioni di memoria dopo il precedente); di conseguenza una stessa cella fisica può essere individuata da molti indirizzi segmentati diversi.

Per calcolare l'indirizzo effettivo associato a un indirizzo segmentato, come ad esempio a0e3:1b56, occorre effettuare un semplice calcolo: si moltiplica per 16 l'indirizzo del segmento (a questo scopo basta aggiungere uno zero alla sua destra) e poi si somma con l'offset.

Applicando la regola all'indirizzo prima citato dobbiamo sommare $\boldsymbol{a0e30_{16}}$ e $\boldsymbol{1b56_{16}}$ ottenendo $\boldsymbol{a2986_{16}}$.

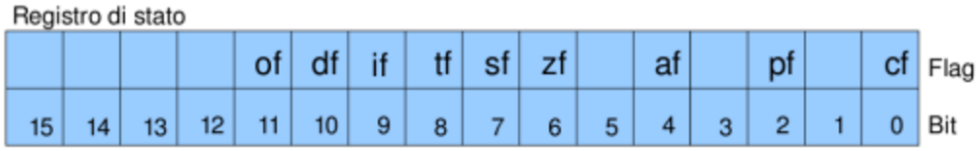
Nella CPU 8086 abbiamo i seguenti 14 registri che hanno un'ampiezza di 16 bit:

- 4 registri di segmento: ***cs*** (Segmento Codice o istruzioni), ***ds*** (Segmento Dati), ***ss*** (Segmento Stack), ***es*** (Segmento Extra dati);
- 4 registri accumulatori o generali: ***ax***, ***bx***, ***cx***, ***dx***, di cui si possono utilizzare anche le rispettive metà, «alta» e «bassa», da 8 bit, identificate con ***ah***, ***al***, ***bh***, ***bl***, ***ch***, ***cl***, ***dh***, ***dl***;
- 2 registri indice: ***di*** e ***si***;
- 2 registri per la gestione della pila o stack: ***sp*** (*Stack Pointer*) e ***bp*** (*Base Pointer*);
- il registro ***flags*** o registro di stato;
- il registro contatore di programma, ***ip*** (*Instruction Pointer*), che però non viene usato direttamente dal programmatore ma serve a contenere l'indirizzo della prossima istruzione che il processore deve eseguire.

Nel registro di stato solo 9 dei 16 bit sono significativi (vedi figura 1.1) e si dividono in:

- flag di stato (bit 0, 2, 4, 6, 7, 11): sono influenzati dai risultati delle operazioni di calcolo aritmetico svolte dalla ALU; anche se esistono istruzioni che ne forzano il cambiamento di valore, vengono soprattutto «consultati» dalle istruzioni di salto condizionato;
- flag di controllo (bit 8, 9, 10): permettono di impostare alcune modalità operative della CPU grazie ad apposite istruzioni che impostano il loro valore.

Figura 1.1.



Vediamo in dettaglio il ruolo di ogni singolo flag:

- flag di stato:
  - flag di carry (***cf***): in caso di operazioni su numeri senza segno, vale 1 se la somma precedente ha fornito un riporto (*carry*) o se la differenza precedente ha fornito un prestito (*borrow*);
  - flag di parità (***pf***): vale 1 se l'operazione precedente ha dato un risultato che ha un numero pari di bit con valore uno;
  - flag di carry ausiliario (***af***): opera come il flag ***cf*** però relativamente a prestiti e riporti tra il bit 3 e il bit 4 dei valori coinvolti, cioè opera a livello di semibyte (o *nibble*); è un flag utile soprattutto quando si opera con valori espressi nel codice BCD (*Binary Coded Decimal*);
  - flag di zero (***zf***): vale 1 se l'ultima operazione ha dato risultato zero;
  - flag di segno (***sf***): vale 1 se il segno del risultato dell'ultima operazione è negativo; essendo i valori espressi in complemento a due, tale flag viene valorizzato semplicemente tramite la copia del bit più significativo del risultato, che vale appunto 1 per i numeri negativi;
  - flag di overflow (***of***): ha la stessa funzione del flag ***cf*** ma per numeri con segno; in pratica segnala se l'ultima operazione ha fornito un risultato che esce dall'intervallo dei valori interi rappresentabili nell'aritmetica del processore;
- flag di controllo:
  - flag di *trap* (***tf***): viene impostato a 1 per avere l'esecuzione del programma *step by step*; utile quando si fa il *debug* dei programmi;
  - flag di *interrupt* (***if***): viene impostato a 1 (istruzione ***sti***) per abilitare le risposte ai segnali di interruzione provenienti dalle periferiche; viene impostato a 0 (istruzione ***cli***) mascherare le interruzioni;
  - flag di direzione (***df***): si imposta per stabilire la direzione secondo la quale operano le istruzioni che gestiscono i caratteri di una stringa (1 verso sinistra, o in ordine decrescente di memoria, 0 verso destra, o in ordine crescente di memoria).

Notiamo che, quando si usano i flag del registro di stato, cioè nelle istruzioni di salto condizionato, si fa riferimento solo alla prima lettera del loro nome (ad esempio il flag di zero è identificato solo con ***z***).

1.2 Novità dell'architettura IA-32 e IA-64

Con IA-32 si ha il passaggio a 32 bit dell'architettura dei processori; il capofila della nuova famiglia è stato l'80386, che ha portato molte novità riguardanti i registri e la gestione della memoria, pur mantenendo inalterate alcune caratteristiche (ad esempio la non ortogonalità, la gestione delle periferiche):

- i registri aumentano di numero e aumentano quasi tutti la lunghezza a 32 bit; rimangono a 16 bit i registri di segmento che assumono il nome di '**`registri selettori`**' e ai quali si aggiungono due ulteriori registri per segmenti extra: ***fs*** e ***gs***;
- i registri accumulatori, il registro di stato e quelli usati come offset diventano tutti a 32 bit e il loro nome cambia con l'aggiunta di una «e» iniziale; abbiamo quindi: ***eip***, ***esp***, ***ebp***, ***esi***, ***edi***, ***eflags***, ***eax***, ***ebx***, ***ecx***, ***edx***;
- i registri accumulatori rimangono utilizzabili anche in modo parziale: abbiamo infatti ***eax***, ***ebx***, ***ecx***, ***edx*** a 32 bit ma si possono ancora usare ***ax***, ***bx***, ***cx***, ***dx*** per usufruire delle rispettive loro parti basse di 16 bit, a loro volta divisibili nelle sezioni a 8, bit come in precedenza.

La successiva evoluzione verso i 64 bit, con l'architettura x86-64, ha portato l'ampiezza dei registri, appunto, a 64 bit con le seguenti caratteristiche:

- i nomi dei registri sono gli stessi di prima con una «r» al posto della «e»;
- ci sono ulteriori otto «registri estesi» indicati con le sigle da ***r8*** a ***r15***;
- rimane la possibilità dell'uso parziale (porzioni di 8, 16, 32 bit, ad esempio ***al*** o ***ah***, ***ax***, ***eax*** nel caso di ***rax***) dei registri accumulatori e dei registri indice;
- per questi ultimi diviene possibile usare anche il solo byte meno significativo, (ad esempio per ***rsp*** si può usare anche ***spl*** oltre a ***esp*** e ***sp***) cosa non prevista nelle CPU x86.

Riguardo alla memoria rimane la possibilità di gestirne 1 MB in modo segmentato come nel-l'8086 (si dice allora che si lavora in '**`modalità reale`**'); la novità è però la '**`modalità protetta`**' nella quale si gestiscono 4 GB di memoria con indirizzi nella forma ***selettore:offset*** chiamati '**`indirizzi virtuali`**'.

Questo nome dipende dal fatto che non è detto che tali indirizzi corrispondano a reali indirizzi fisici di memoria (solo da poco tempo i comuni Personal Computer hanno una dotazione di memoria centrale di qualche GB); i processori sono infatti in grado di gestire la '**`memoria virtuale`**' con un meccanismo detto di '**`paginazione dinamica`**' sfruttato poi in modo opportuno anche dai moderni sistemi operativi.

Questo argomento richiederebbe notevoli approfondimenti che però esulano dagli scopi di queste dispense.

Concludiamo citando la presenza della '**`tabella dei descrittori`**', alle cui righe si riferiscono i registri selettori, che contiene indirizzi iniziali e dimensioni dei vari segmenti; questi ultimi hanno infatti dimensione variabile (massimo 4 GB).

Di ogni selettore solo 13 dei 16 bit sono usati per individuare una riga della tabella; i segmenti sono quindi al massimo 8192.

Gli indirizzi virtuali nella forma ***selettore:offset*** con offset a 32 bit, vengono tradotti in '**`indirizzi lineari`**' a 32 bit che, come detto, non è detto che corrispondano direttamente a indirizzi fisici di memoria.

Lo spazio virtuale effettivamente a disposizione di ogni processo è comunque di soli 3 GB perché la zona più alta, ampia 1 GB, è uno spazio comune a tutti i processi in esecuzione e riservata al *kernel* di Linux.

Capitolo 2

Dal sorgente all'eseguibile

L'assemblatore AT&T per linux si chiama '**as**' o GAS e fa parte del pacchetto '**gcc**'; essendo tale pacchetto disponibile per le piattaforme Windows, con il nome di '**cygwin**', è possibile usare as anche in tale ambiente.

I sorgenti scritti con questo linguaggio hanno per convenzione l'estensione «.s» ma è possibile scegliere qualsiasi altro tipo di nome.

2.1 Traduzione del sorgente con as

Per assemblare un sorgente si deve eseguire il comando:

```
$ as -a -o nome_oggetto.o  nome_sorgente.s
```

L'opzione '**-o**' serve ad indicare il nome del file oggetto ottenuto con la traduzione; essa però si conclude positivamente solo se non vengono riscontrati '**errori di sintassi**' dei quali eventualmente si riceve opportuna segnalazione (descrizione dell'errore e indicazione della riga in cui è avvenuto).

L'opzione '**-a**' non è obbligatoria e ha lo scopo di ottenere a video il sorgente affiancato dalla traduzione in codice macchina delle istruzioni e seguito dalle liste dei simboli (etichette dati e istruzioni); il fatto che tali informazioni vengano rese disponibili a video permette di poterle eventualmente ridirezionare su un file per un loro più comodo esame.

Altra opzione importante è '**--gstabs**' che costringe l'assemblatore a inserire maggiori informazioni all'interno del file oggetto; tali informazioni sono indispensabili se si vogliono usare programmi *debugger* come '**gdb**', che è il più usato in ambiente GNU/Linux.

Una volta concluso il collaudo del programma ed eliminati tutti gli errori, è però consigliabile assemblare il programma senza l'opzione '**--gstabs**' in modo da avere un file oggetto, e quindi un eseguibile, più snello e con migliori prestazioni.

Per avere dettagli sulle molte altre opzioni disponibili si può eseguire il comando:

```
$ man as
```

Se si lavora con una CPU a 64 bit ma si vuole ottenere un programma oggetto a 32 bit si deve aggiungere l'opzione '**--32**' al comando '**as**'.

2.2 Fase di linking e esecuzione

Per ottenere l'eseguibile occorre usare il programma per il linking '**ld**' nel modo seguente:

```
$ ld -o nome_eseguibile  nome_oggetto.o
```

L'opzione '**-o**' permette di indicare il nome del file eseguibile.

Anche questo comando prevede molte opzioni e inoltre offre la possibilità di fare il linking di più file oggetto contemporaneamente; nei prossimi capitoli sfrutteremo alcune sue potenzialità esaminando programmi sempre più complessi.

Ovviamente è anche qui possibile consultare il manuale in linea con:

```
$ man ld
```

Per eseguire il programma occorre il comando:

`$` ***./nome_eseguibile***

oppure:

`$` ***/percorso_per_raggiungere_eseguibile/nome_eseguibile***

2.3 Cenni all'uso del debugger '**gdb**'

L'esecuzione di un programma scritto in assembly non sempre offre risultati che possono essere apprezzati direttamente; questo accade soprattutto quando si inizia a usare il linguaggio e quindi non si usano istruzioni per l'input e l'output.

Può inoltre capitare che, in presenza di risultati visibili da parte del programma, questi non coincidano con quelli attesi oppure che il programma abbia comportamenti imprevisti e strani; in tal caso siamo in presenza di '**errori run-time**' o di '**errori logici**'.

In entrambe queste situazioni diventa indispensabile l'uso di un debugger come '**gdb**' in modo da poter inserire dei *breakpoint* che permettono poi l'esecuzione passo dopo passo, e da poter visionare il contenuto della memoria e dei principali registri.

Per attivarlo si deve dare il comando:

`$` **gdb** ***nome_eseguibile***

Ci troviamo in questo modo nell'ambiente interattivo di gdb cha ha un suo *prompt*:

```
(gdb)
```

e mette a disposizione una serie di comandi come:

- '**q**': esce dall'ambiente gdb;
- '**h**': aiuto sui comandi gdb;
- '**l**': lista alcune righe del sorgente;
- '**l**' ***num_riga***: lista le righe del sorgente subito prima e subito dopo quella indicata;
- '**info address**' ***var***: restituisce l'indirizzo di ***var***;
- '**info variables**': restituisce nome e indirizzo di tutte le variabili;
- '**info registers**': visualizza lo stato dei registri;
- '**breakpoint**' ***num_riga***: inserisce un blocco subito dopo la riga indicata;
- '**r**': esegue il programma fino al primo blocco;
- '**c**': riprende l'esecuzione;
- '**s**': esegue solo l'istruzione successiva.

Per maggiori dettagli si può consultare il manuale del comando gdb, che è comunque uno strumento abbastanza ostico anche se molto potente.

Per facilitare il compito ai programmatori assembly sono stati allora creati vari programmi che sono delle interfacce grafiche di gdb; esempi in tal senso possono essere '**ddd**' e '**insight**'.

Sono entrambi strumenti abbastanza semplici e intuitivi e per il loro uso dovrebbe essere sufficiente la consultazione dei rispettivi manuali on-line; di ddd vedremo esempi di utilizzo nei prossimi paragrafi.

Si ricorda in ogni caso che per poter usare questi programmi occorre avere assemblato il sorgente con l'opzione '**--gstabs**'.

2.4 Uso di gcc per i programmi assembly

Si può anche usare gcc in alternativa ad as per la traduzione del sorgente assembly; in questo caso il comando da lanciare è:

```
$ gcc -c -g -o nome_oggetto.o nome_sorgente.s
```

L'opzione '**-o**' ha il ruolo già illustrato in precedenza, l'opzione '**-c**' fa si che gcc si fermi alla creazione del file oggetto senza procedere con il linking e l'opzione '**-g**' (ovviamente facoltativa) ha lo stesso scopo di '**--gtabs**' quando si usa as.

Infine vediamo che è anche possibile fare tutto (traduzione e linking) con gcc con il comando:

```
$ gcc -g -o nome_eseguibile nome_sorgente.s
```

però in questo caso la parte esecutiva del programma deve iniziare con l'etichetta ***main:*** al posto di ***_start:***; questo aspetto sarà forse più chiaro al momento in cui vedremo la struttura del sorgente assembly AT&T (paragrafo 3.4):

Quando si compila un programma sorgente in linguaggio '**c**' con il comando:

```
$ gcc -o nome_eseguibile nome_sorgente.c
```

otteniamo come risultato il programma eseguibile (se non ci sono errori); in realtà però avviene un doppio passaggio realizzato da gcc:

1. viene creato un file oggetto temporaneo ***.o***;
2. viene creato l'eseguibile tramite il linker ld e viene rimosso il file oggetto temporaneo.

Il file oggetto temporaneo è un file in linguaggio macchina del tutto analogo a quello che si ottiene quando si assembla un sorgente ***.s***.
Se si vuole vedere il codice assembly corrispondente all'oggetto creato da gcc quando compila un sorgente '**c**', occorre eseguire il comando:

```
$ gcc -S nome_sorgente.c
```

e si ottiene come risultato il listato assembly in ***nome_sorgente.s***.

Se si lavora con una CPU a 64 bit ma si vuole ottenere un programma eseguibile a 32 bit si deve aggiungere l'opzione '**-m32**' al comando '**gcc**' (in tal caso deve essere installato il pacchetto '**gcc-multilib**').

Capitolo 3

Il linguaggio assembly con sintassi AT&T

Come già accennato, in questa sede non si vogliono considerare tutti gli aspetti del linguaggio assembly per GNU/Linux, ma esaminare le tecniche di base e le istruzioni fondamentali per poter gestire cicli, vettori, I/O, procedure e stack; questo verrà fatto usando prevalentemente esempi di programmi funzionanti che saranno opportunamente commentati.

Negli esempi vengono usate le istruzioni più comuni del linguaggio, un elenco delle quali è disponibile nell'appendice A.

3.1 Caratteristiche generali

Elenchiamo alcune caratteristiche fondamentali della sintassi e delle regole d'uso delle istruzioni dell'assembly con sintassi AT&T:

- per i nomi delle etichette dati o istruzioni, delle macro e delle procedure si possono usare lettere e cifre e qualche carattere speciale come «_», senza limiti di lunghezza; si ricordi però che non possono iniziare con una cifra e che il linguaggio è *case sensitive* e quindi c'è differenza tra maiuscole e minuscole;
- i commenti si possono inserire in tre modi diversi:
  - carattere «#»;
  - caratteri «//» (solo da inizio riga);
  - caratteri «/*» (inizio commento) e «*/» (fine commento); in questo modo si possono avere anche commenti su più righe.
- i valori numerici si possono rappresentare in esadecimale, prefissandoli con «0x», in binario, con il prefisso «0b» e in decimale, scrivendo i valori senza alcun prefisso;
- i valori costanti devono essere preceduti dal simbolo «$»;
- i nomi dei registri devono essere preceduti dal simbolo «%»;
- le istruzioni che coinvolgono registri e locazioni di memoria prevedono un suffisso che può essere:
  - ***b*** se l'istruzione è riferita a byte;
  - ***w*** se l'istruzione è riferita a word (due byte);
  - ***l*** se l'istruzione è riferita a doppia word (***l*** deriva da «long»);

  in verità l'assemblatore GAS permette l'omissione del suffisso deducendolo in base alla natura degli operandi; il sio uso è comunque consigliabile per aumentare la leggibilità dei sorgenti;
- il simbolo «$» può precedere anche un nome di etichetta (ad esempio ***dato1***); in tal caso serve a riferirsi al suo indirizzo in memoria (***$dato1***); in pratica c'è la stessa differenza che si ha in linguaggio '`c`' fra l'uso della varialbile ***dato1*** e l'applicazione ad essa dell'operatore «&» (***&dato1***);
- la maggior parte delle istruzioni dell'assembly AT&T prevede due operandi secondo questo modello:

```
codice_istr  operando1, operando2
```

è di fondamentale importanza osservare che nelle istruzioni gli operandi non possono essere entrambi riferimenti alla memoria (le etichette dati di cui si parla nel paragrafo 3.3).

3.2 I segmenti di un programma

Un programma assembly è suddiviso in parti, chiamate segmenti, in modo da rispecchiare l'architettura dei processori Intel x86 (riassunta nel paragrafo 1.1).

I segmenti fondamentali sono quattro:

- code segment, contenente le istruzioni del programma;
- data segment, contenente i dati;
- stack segment, contenente lo stack (vedere il paragrafo 3.14);
- extra segment; contenente eventuali dati aggiuntivi (usato molto raramente).

3.3 Tipi di istruzioni

Come in tutti i linguaggi assembly anche in quello con sintassi AT&T esistono tre tipi diversi di istruzioni in un sorgente:

- '**`direttive`**': non si tratta di istruzioni del linguaggio assembly ma di istruzioni rivolte all'assemblatore per fornirgli informazioni utili alla traduzione del programma (ad esempio ***.data*** e ***.text*** usate per indicare l'inizio del segmento dati e del segmento codice rispettivamente;
- '**`dichiarative`**': sono le istruzioni che servono a dichiarare le etichette;
- '**`esecutive`**': sono le vere e proprie istruzioni corrispondenti alle azioni che l'esecutore dovrà svolgere.

Le istruzioni direttive e dichiarative (che nel nostro caso sono ancora delle direttive usate per dichiarare le etichette dati) non vengono tradotte in linguaggio macchina, cosa che invece avviene per le istruzioni esecutive.

Fra queste ultime e le istruzioni macchina, a parte qualche eccezione, c'è una relazione «uno a uno».

Fra le etichette definite con le istruzioni dichiarative si possono distinguere:

- '**`etichette dati`**': inserite nel segmento dati, usate per definire dati in memoria; possono essere considerate equivalenti alle variabili presenti nei linguaggi di programmazione ad alto livello ('**`c`**', '**`pascal`**' ecc.);
- '**`etichette istruzioni`**': inserite nel segmento istruzioni, per poter effettuare i salti all'interno del programma (vedere il paragrafo 3.13).

3.4 Struttura del sorgente

Vediamo un primo esempio di programma in assembly allo scopo di illustrare la struttura generale di un sorgente; la numerazione delle righe non è presente in assembly ed è aggiunta qui al solo scopo di facilitare la descrizione del listato:[1]

```
1   /*
2   Programma:     modello.s
3   Autore:        FF
4   Data:          gg/mm/aaaa
5   Descrizione:   Modello di sorgente per assembly AT&T
6   */
7   .data
8   .text
9   .globl _start
10  _start:
11     nop
12  fine:
13     movl $1, %eax
14     int  $0x80
```

Le prime sei righe sono di commento e quindi non essenziali; è comunque consigliabile inserire sempre all'inizio righe come queste per fornire informazioni utili sul programma.

Alla riga 7 abbiamo la direttiva che denota l'inizio del segmento dati, che in questo esempio è vuoto; quindi subito dopo inizia il segmento codice in cui la prima istruzione è una direttiva (riga 9) che dichiara globale l'etichetta ***_start***.

A seguire, alla riga 10 è inserita tale etichetta con lo scopo di denotare l'inizio della parte esecutiva del programma; il suo uso, preceduto dalla relativa dichiarazione, è obbligatorio in tutti i programmi a meno che non si usi gcc per la traduzione e la produzione del programma eseguibile, nel qual caso deve essere usata (e dichiarata con ***.globl***) l'etichetta ***main***.

Questo programma non svolge alcuna elaborazione significativa e infatti contiene solo l'istruzione ***nop*** che, appunto, non fa niente.

Le ultime due istruzioni invece sono importanti e servono ad attivare l'interruzione software identificata con il valore $\boldsymbol{80_{16}}$ con la quale, in generale, si richiamano *routine* o servizi del sistema operativo (nel nostro caso Linux).

Il servizio richiamato è identificato dal valore che viene inserito nel registro ***eax*** prima di attivare l'interruzione; il valore ***1*** corrisponde alla *routine* di uscita dal programma e ritorno al sistema operativo.

Siccome ogni programma deve in qualche modo terminare e restituire il controllo al sistema operativo, queste due istruzioni saranno presenti in qualsiasi sorgente assembly.

L'etichetta istruzioni ***fine*** presente a riga 12 è inserita solo allo scopo di evidenziare ancora meglio queste operazioni finali e non è usata per effettuare alcun tipo di salto.

3.5 Primo esempio di programmazione

Il primo programma che esaminiamo svolge alcune somme e sottrazioni usando alcuni valori immediati (costanti) e i registri accumulatori opportunamente caricati con dei valori numerici.[2]

```
1  /*
2  * Descr.: esempio di somma e sottrazione con uso dei registri
3  * Autore: FF
4  * Data  : gg/mm/aaaa
5  * Note  : Assemblare con: as -o nome.o nome.s (eventuale
6  *         opzione --gstabs per poter usare gdb o ddd)
7  *         Linkare con ld -o nome nome.o
8  *         Eseguire con ./nome o (per il debug) con gdb nome o ddd nome
9  */
10 .data
11 .text
12 .globl _start
13 _start:
14    nop
15    movb $32, %al
16    movw $2048, %bx
17    movb $-1, %cl
18    movl $1000000000, %edx
19    subw %bx, %bx
20    movb $0x1, %bh
21    movl $3500000000, %ecx
22    addl %ecx, %edx
23    movb $1, %dh
24    movl $-0x1, %ecx
25    xorl %eax, %eax
26    movb $254, %al
27    addb $1, %al
28 fine:
29    movl $1, %eax
30    int $0x80
```

Le prime istruzioni, da riga 15 a riga 18, inseriscono dei valori nei registri accumulatori; si noti l'accordo fra il suffisso dell'istruzione ***mov*** e l'ampiezza del registro usato.

Alla riga 19 vediamo una sottrazione tra il registro ***bx*** e se stesso; è un modo per azzerare il contenuto del registro.

Stesso effetto si ottiene usando l'operatore di «or esclusivo» come mostrato a riga 25.

I valori costanti possono essere indicati anche in formato esadecimale (righe 20 e 24).

Alle righe 22 e 27 vediamo due operazioni di somma; a tale proposito si deve ricordare che in queste operazioni, come in tutte quelle simili con due operandi, il risultato viene immagazzinato nel secondo operando, il cui valore precedente viene quindi perso.

3.6 Il debugger '**ddd**'

Il debugger '**ddd**' è in pratica un'interfaccia grafica che facilita l'uso del debugger standard di GNU/Linux, cioè gdb.

Per usarlo occorre naturalmente avere installato l'omonimo pacchetto disponibile per tutte le distribuzioni di Linux.

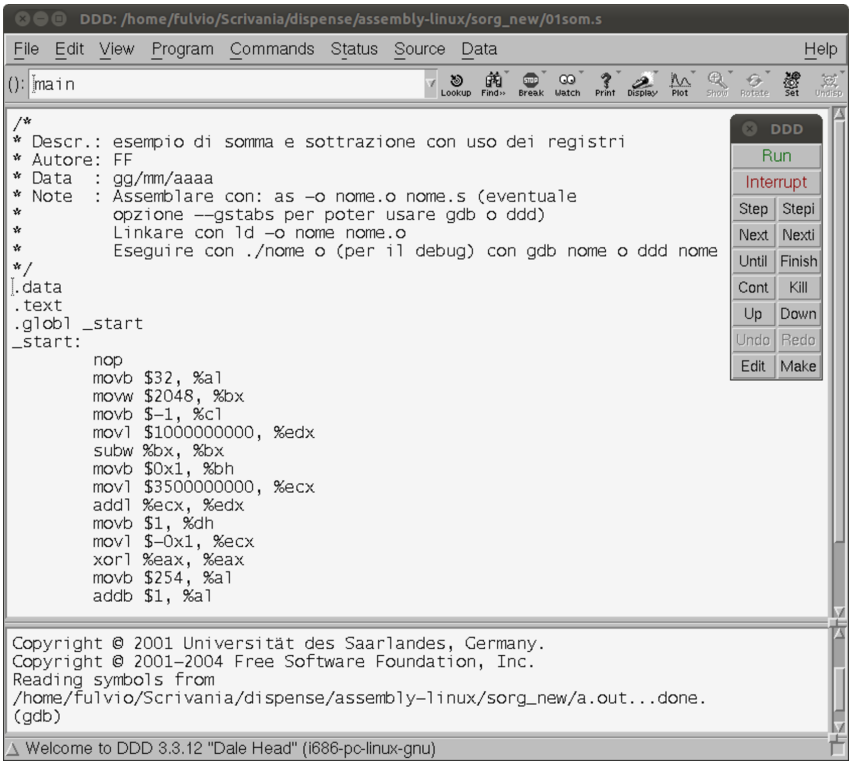
Vediamo le basi del suo utilizzo con alcune immagini tratte dall'esecuzione del programma illustrato nel paragrafo precedente.

Per attivare l'esecuzione con ddd occorre eseguire il comando:

```
$ ddd nome_eseguibile
```

Il programma si attiva mostrando la finestra visibile nella figura 3.4.

Figura 3.4.

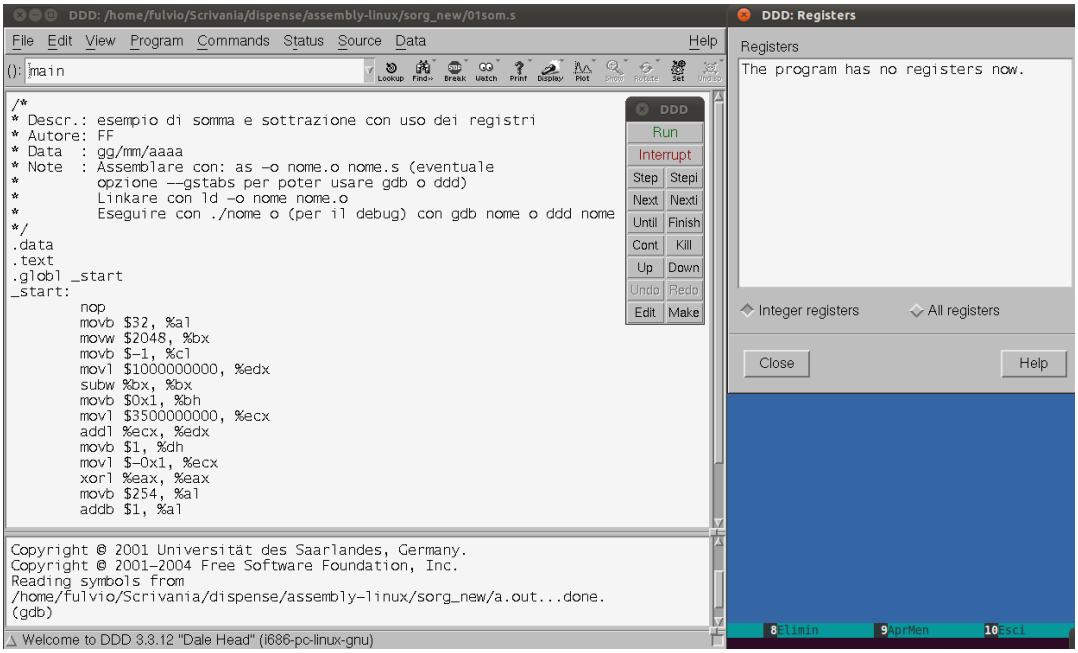


Bisogna aprire subito anche la finestra che mostra i valori dei registri attivando la voce «Registers» del menu «Status» e poi inserire un *breakpoint* all'inizio del programma (sulla seconda riga eseguibile, perché sulla prima talvolta non funziona).

Per fare questo occorre prima posizionarsi sulla riga desiderata e poi premere il pulsante «Break», riconoscibile grazie all'icona con il cartello di stop, presente nella barra degli strumenti.

Ci si trova così nella situazione mostrata nella figura 3.5.

Figura 3.5.

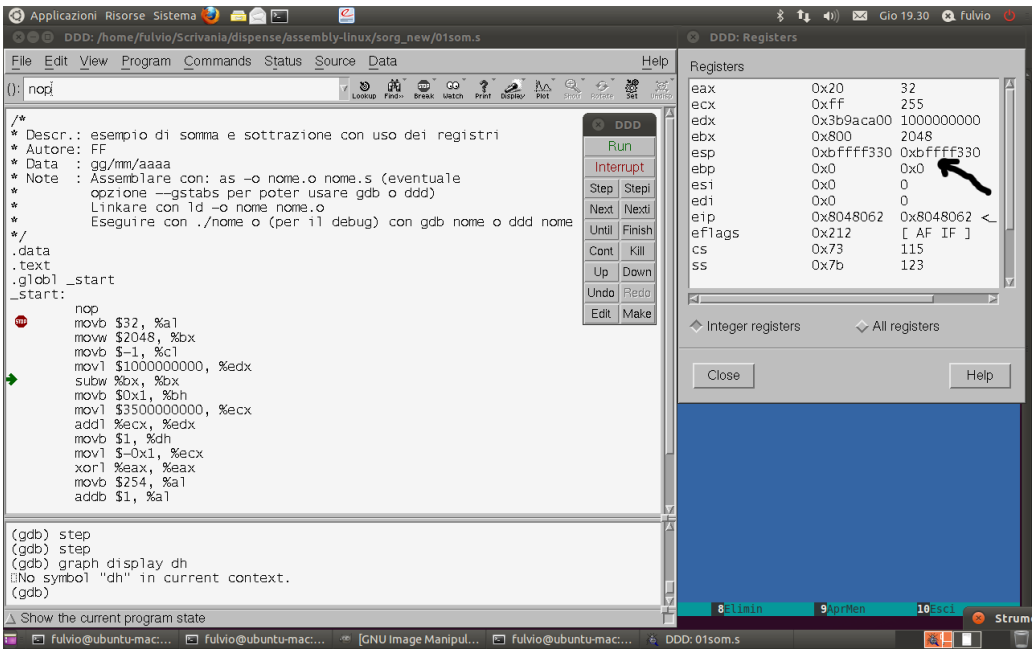


A questo punto si può *cliccare* sul pulsante «Run» nella finestrina «DDD» per provocare l'esecuzione fino alla riga con il *breakpoint* e successivamente sul pulsante «Step» per avanzare una istruzione alla volta; la freccia verde che appare sulla sinistra del listato indica la prossima istruzione che verrà eseguita all pressione del pulsante «Step».

Nella finestra dei valori dei registri si può seguire il variare del loro contenuto, espresso in esadecimale e in decimale, man mano che le istruzioni vengono eseguite.

Nella figura 3.6 si può vedere la situazione nei registri dopo le prime quattro istruzioni di caricamento (righe dalla 15 alla 18).

Figura 3.6.



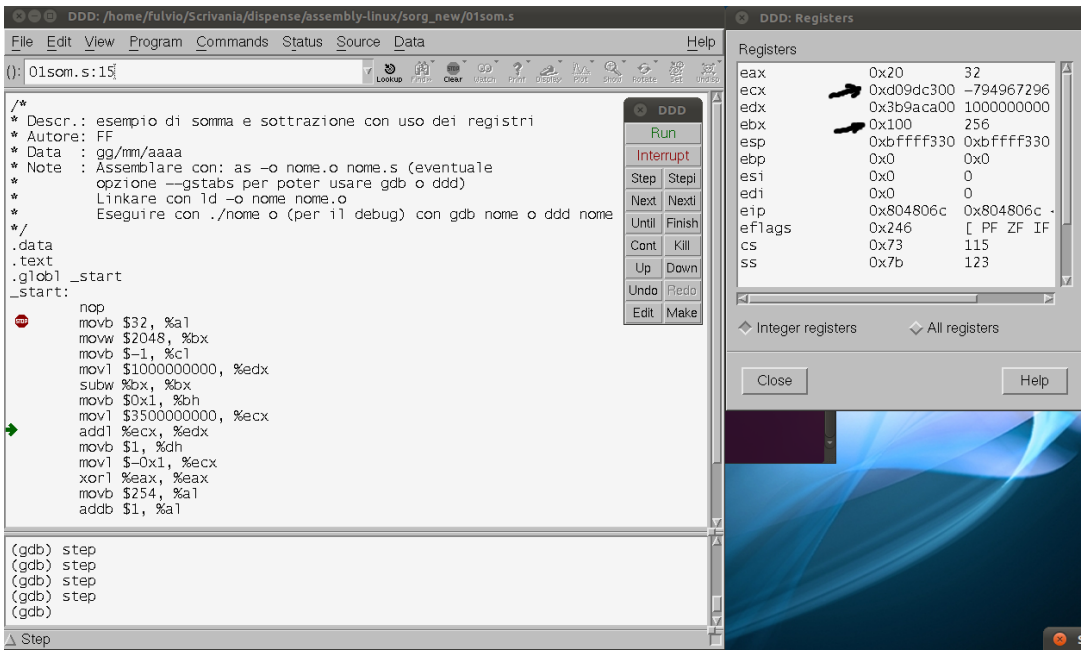
In particolare è interessante considerare il valore che appare nella parte bassa del registro ***ecx***: ***255*** (***0xff*** in esadecimale) e non ***-1***.

Sappiamo però, per le regole di rappresentazione dei valori interi in complemento a due (in questo caso su 8 bit in quanto abbiamo usato il registro ***cl***), che il valore ***-1*** si rappresenta proprio come ***0xff***.

Possiamo quindi notare come uno stesso insieme di bit memorizzato in un registro possa rappresentare sia un valore positivo che uno negativo (***255*** o ***-1*** nel nostro esempio); in base al contesto del programma e all'uso che di quel valore fa il programmatore, viene «deciso» quale delle due alternative è quella da prendere in considerazione.

La successiva figura 3.7 mostra i valori nei registri dopo le istruzioni delle righe,19, 20 e 21.

Figura 3.7.

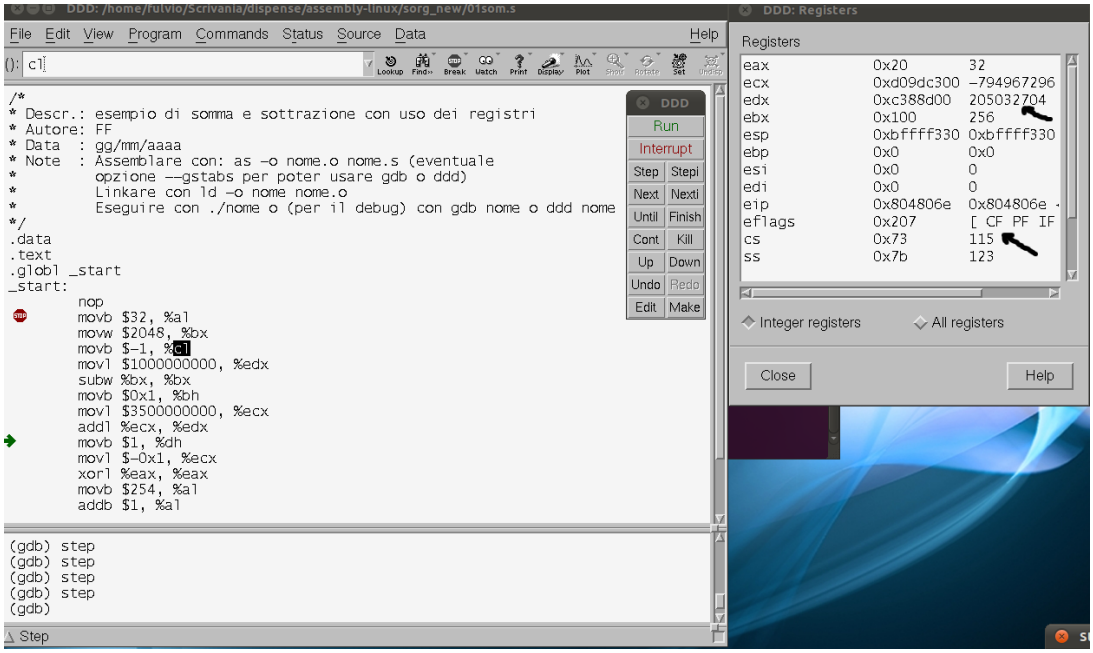


Possiamo vedere come l'aver posto una unità in ***bh*** fa assumere al registro ***bx*** il valore ***256***; se ragioniamo in decimale la cosa appare alquanto strana, molto meno se consideriamo la corrispondente rappresentazione esadecimale ***0x100*** in cui appare più evidente che gli 8 bit più bassi (corrispondenti a ***bl***) sono tutti a zero mentre gli altri 8 bit contengono il valore ***1***.

Altrettanto interessante appare il valore presente in ***cx*** che è negativo nella colonna dei valori decimali; come già osservato poco sopra dipende dal fatto che i bit corrispondenti all'esadecimale ***0xd09dc300*** rappresentano sia il valore ***3,500,000,000*** che ***-794,967,296***.

Proseguendo nell'esecuzione si arriva alla somma di riga 22; nella figura 3.8 ci concentriamo sul registro ***dx*** che dovrebbe contenere il risultato dell'operazione e invece contiene tutt'altro valore.

Figura 3.8.



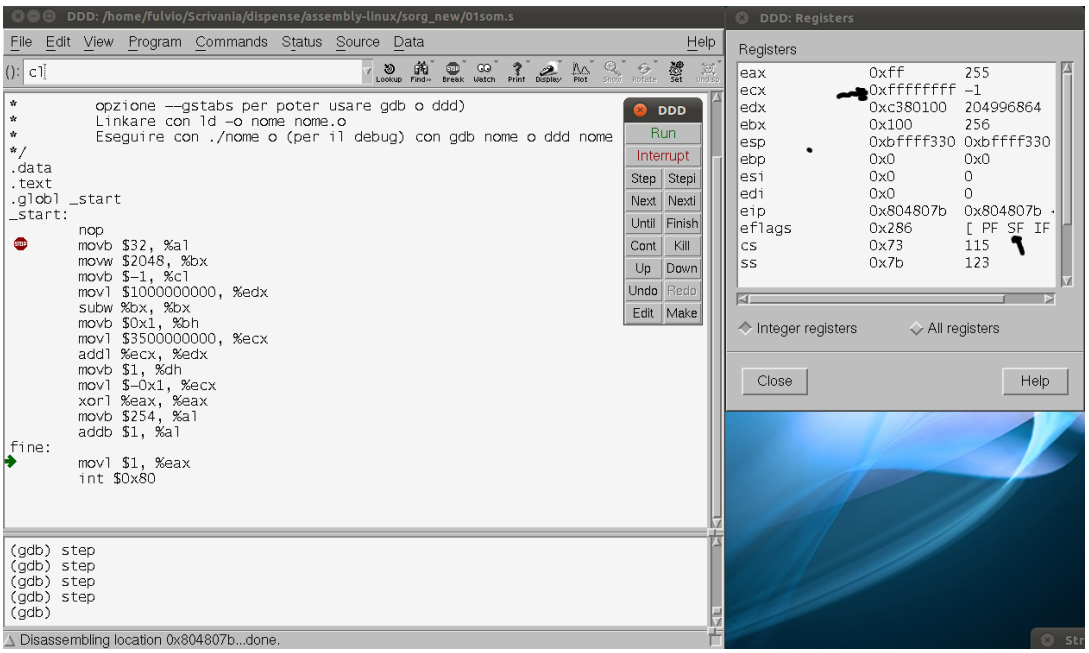
Anche in questo caso ci aiuta ricordare le regole di rappresentazione dei valori interi: abbiamo sommato 3,5 miliardi a 1 miliardo, ma il risultato supera il massimo valore rappresentabile con 16 bit (***4,294,967,295***); quello cha appare in ***dx*** è proprio il valore che è «traboccato» oltre il massimo possibile.

Conferma del traboccamento si ha anche esaminando il registro ***eflags*** che mostra l'attivazione del flag di carry o trabocco (CF) insieme a quello di parità (PF).

Ricordiamo che il settaggio dei bit del registro dei flag avviene solo dopo istruzioni aritmetico logiche che coinvolgono la ALU; quindi non si ha alcuna alterazione di tali bit dopo le istruzioni ***mov***.
Si può verificare questo provando ad azzerare un registro muovendoci il valore zero; in tal caso non si accende il bit di zero (ZF) che invece si attiva effettuando la ***sub*** o la ***xor*** di un registro su se stesso.

Concludiamo con la figura 3.9 in cui vediamo l'attivazione del bit di segno (SF) dopo la somma di una unità al registro ***al*** che conteneva ***254*** (riga 27 del programma); anche questo comportamento pare anomalo ma è giustificato dal fatto che ***255*** può essere anche interpretato come ***-1***.

Figura 3.9.



Nella stessa figura vediamo anche come il valore ***-1*** su 16 bit (nel registro ***cx***) corrisponda a ***0xffffffff*** e cioè «anche» al valore ***65,535***.

3.7 La situazione di overflow

Il successivo esempio, in cui si usano altre istruzioni molto banali come ***xchg***, ***inc***, ***dec***, ha un certo interesse in quanto mostra delle situazioni in cui si ottiene l'impostazione del bit di overflow.[3]

```
1   /*
2   * Descr.: esempio con operazioni varie (inc, dec, xchg ...)
3   * Autore: FF
4   * Data  : gg/mm/aaaa
5   * Note  : Assemblare con: as -o nome.o nome.s (eventuale
6   *          opzione --gstabs per poter usare gdb o ddd)
7   *          Linkare con ld -o nome nome.o
8   *          Eseguire con ./nome o (per il debug) con gdb nome o ddd nome
9   */
10  .data
11  .text
12  .globl _start
13  _start:
14     nop
15     movb $64, %al
16     movb $63, %bl
17     xchgb %al, %bl
18     addb %bl, %al
19     incb %al
20     decb %al
21  fine:
22     movl $1, %eax
23     int $0x80
```

I dati vengono posti nei registri ***al*** e ***bl*** che poi vengono scambiati alla riga 18; viene poi fatta la somma che fornisce un risultato «normale» senza causare l'impostazione di alcun bit di stato in particolare.

Quando, alla riga 20, viene incrementato di una unità il registro ***al***, si ottiene il risultato atteso (***128***) ma con la contemporanea accensione (tra gli altri) del bit di segno e del bit di overflow; ciò è del tutto normale se si ricorda che con otto bit il massimo intero rappresentabile è ***127*** e che la rappresentazione è «modulare» (gli interi sono immaginati disposti su una circonferenza e dopo il più alto positivo si trovano i valori negativi).

Con l'istruzione successiva, che riporta il valore del registro a ***127***, viene impostato il bit di overflow ma non quello di segno; si ha infatti un prestito dal bit in ottava posizione in una differenza tra valori considerabili con segno mentre il risultato ritorna ad essere positivo.

Stavolta non vengono mostrate le immagini che illustrano questi passaggi fatti con ddd; il lettore può facilmente provvedere autonomamente operando come mostrato nei casi precedenti.

3.8 Operazioni logiche

Nel successivo esempio prendiamo in esame alcune operazioni logiche.[4]

```
1   /*
2   Descrizione:   Esempio di operazioni logiche
3   Autore:        FF
4   Data:          gg/mm/aaaa
5   */
6   .data
7   .text
8   .globl _start
9   _start:
10     nop
11     movb $127, %bl
12     negb %bl
13     notb %bl
14     movb $15, %al
15     andb $35, %al
16     movb $15, %al
17     orb  $35, %al
18  fine:
19     movl $1, %eax
20     int  $0x80
```

Alla riga 12 si effettua la negazione (complemento a due) del valore contenuto in ***bl*** (***127***); il risultato che si ottiene è ***129***.

Di questo valore, alla riga successiva, viene fatta la ***not***, cioè il complemento a uno, che fornisce come risultato ***126***.

Anche in questo caso tralasciamo i dettagli dell'esecuzione con ddd.

Per accertarci della correttezza dei risultati consideriamo la rappresentazione in binario del valore ***127*** che è $\mathbf{01111111_2}$: facendo il complemento a due si ottiene $\mathbf{10000001_2}$, che corrisponde a ***129***; poi si passa al complemento a uno di quest'ultimo ottenendo $\mathbf{01111110_2}$, cioè ***126***.

Il programma prosegue proponendo, alle righe 15 e 17, le operazioni logiche di ***and*** e ***or*** fra i valori ***35*** e ***15***, con quest'ultimo memorizzato in ***al***; i risultati sono ***3*** e ***47*** rispettivamente.

Lasciamo al lettore la verifica di tali risultati partendo dalle rappresentazioni binarie di ***15*** (***00001111_2***) e ***35*** (***00100011_2***) tramite le facili operazioni di ***and*** e ***or*** bit a bit.

3.9 Operazioni di moltiplicazione e divisione

Le operazioni di moltiplicazione e divisione (da intendere come divisione tra interi che fornisce un quoziente e un resto) prevedono un solo operando.

Tale operando rappresenta il moltiplicatore nella moltiplicazione e il divisore nella divisione; il moltiplicando e il dividendo devono essere precaricati in appositi registri prima di svolgere l'operazione.

Si tenga anche presente che l'operando delle moltiplicazioni e divisioni non può essere un valore immediato.

Più in dettaglio il funzionamento della istruzione ***mul*** è il seguente:

- ***mulb***: operando a 8 bit, l'altro valore deve essere in ***al*** e il risultato viene posto in ***ax***;
- ***mulw***: operando a 16 bit, l'altro valore deve essere in ***ax*** e il risultato viene posto in ***dx:ax***;
- ***mull***: operando a 32 bit, l'altro valore deve essere in ***eax*** e il risultato viene posto in ***edx:eax***.

A seguito di una moltiplicazione viene impostato il flag ***cf*** per segnalare se è usato il registro contenente la parte alta del risultato (***dx*** o ***edx***).

Per quanto riguarda l'istruzione ***div*** abbiamo invece:

- ***divb***: operando (divisore) a 8 bit, il dividendo a 16 bit deve essere in ***ax***; il quoziente viene posto in ***al*** e il resto in ***ah***;
- ***divw***: operando (divisore) a 16 bit, il dividendo a 32 bit deve essere in ***dx:ax***; il quoziente viene posto in ***ax*** e il resto in ***dx***;
- ***divl***: operando (divisore) a 32 bit, il dividendo a 64 bit deve essere in ***edx:eax***; il quoziente viene posto in ***eax*** e il resto in ***edx***;

A seguito di una divisione si può verificare una condizione di overflow che, al momento dell'esecuzione, causa una segnalazione di errore del tipo: «Floating point exception» («Eccezione in virgola mobile»); questo avviene se il quoziente è troppo grande per essere contenuto nel registro previsto come nella seguente porzione di codice:

```
movw $1500, %ax
movb $3, %bl
divb %bl
```

Nel listato seguente vengono effettuate alcune operazioni di moltiplicazione e divisione. [5]

```
1  /*
2  * Descrizione:   Esempio di moltiplicazioni e divisioni
3  * Autore:        FF
4  * Data:          gg/mm/aaaa
```

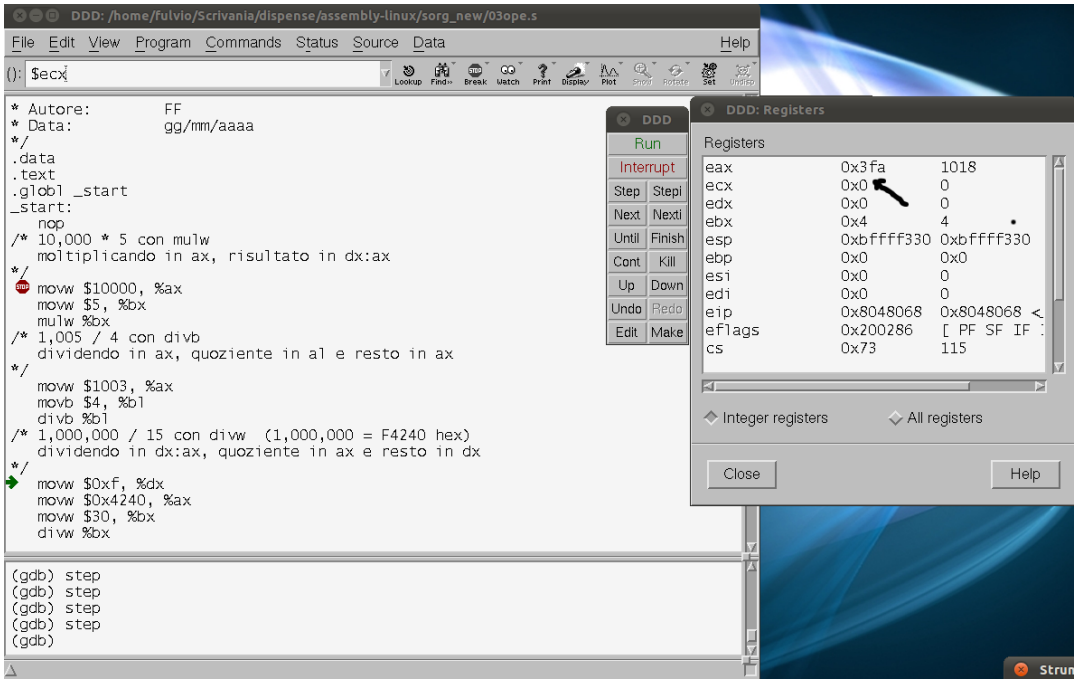
```
5   */
6   .data
7   .text
8   .globl _start
9   _start:
10     nop
11  /* 10,000 * 5 con mulw
12     moltiplicando in ax, risultato in dx:ax
13  */
14     movw $10000, %ax
15     movw $5, %bx
16     mulw %bx
17  /* 1,003 / 4 con divb
18     dividendo in ax, quoziente in al e resto in ax
19  */
20     movw $1003, %ax
21     movb $4, %bl
22     divb %bl
23  /* 1,000,000 / 15 con divw  (1,000,000 = F4240 hex)
24     dividendo in dx:ax, quoziente in ax e resto in dx
25  */
26     movw $0xf, %dx
27     movw $0x4240, %ax
28     movw $30, %bx
29     divw %bx
30  /* 1,000,000,000 * 15 con mull
31     moltiplicando in eax, risultato in edx:eax
32  */
33     movl $15, %eax
34     movl $1000000000, %ebx
35     mull %ebx
36  /* ris. precedente / 10 con divl
37     dividendo in edx:eax, quoziente in eax e resto in edx
38  */
39     movl $10, %ecx
40     divl %ecx
41  fine:
42     movl $1, %eax
43     int  $0x80
```

Nel listato sono presenti dei commenti che spiegano le operazioni svolte.

La prima moltiplicazione (righe da 14 a 16) non necessita di grosse osservazioni in quanto agisce su operandi residenti su singoli registri e fornisce il risultato atteso nel registro ***ax*** sufficiente ad accogliere il valore ***50,000***.

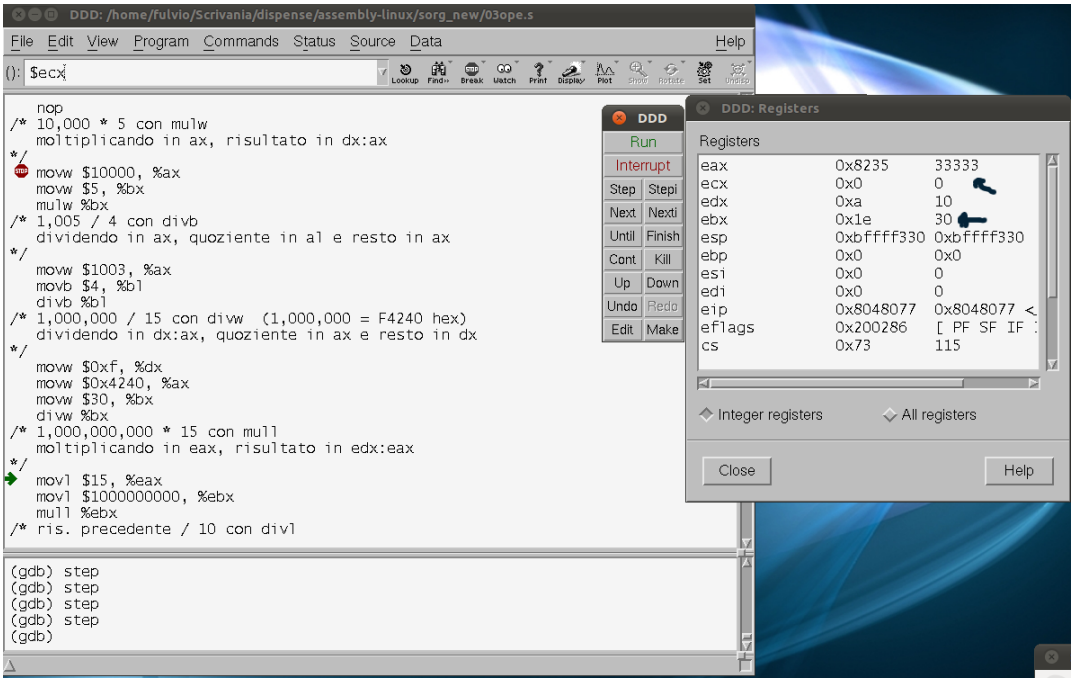
Più interessante è il risultato della prima divisione (righe da 20 a 22) che appare (come al solito) incomprensibile se visto in decimale, più chiaro in esadecimale: vediamo infatti nella figura 3.14 come in ***al*** sia presente il quoziente ***0xfa*** (***250***) e in ***ah*** il resto ***3***.

Figura 3.14.



Nella successiva divisione fatta con ***divw*** (righe da 26 a 29) il dividendo viene posto in parte nel registro ***dx*** e in parte nel registro ***ax*** in quanto è troppo grande per essere contenuto solo in quest'ultimo; dopo la divisione troviamo il quoziente in ***ax*** e il resto in ***dx*** come previsto (vedere figura 3.15).

Figura 3.15.

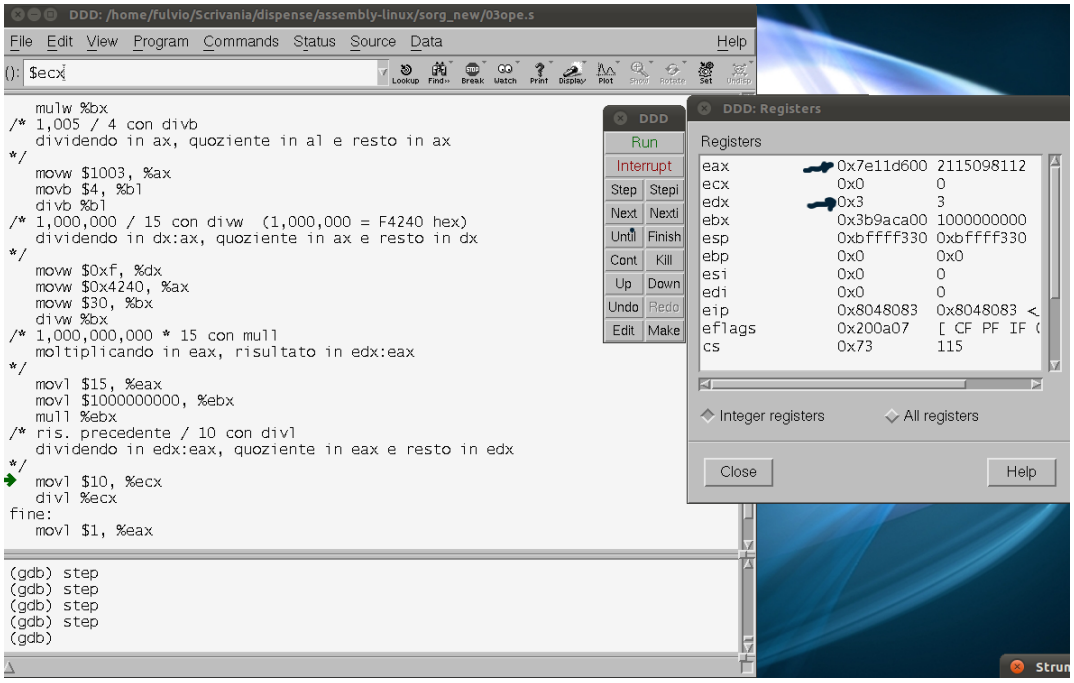


Proseguendo vediamo la moltiplicazione del valore ***1,000,000,000*** caricato in ***ebx*** con ***eax*** che contiene ***15*** (righe da 33 a 35): il risultato non può essere contenuto nei 32 bit del registro ***eax***

(il massimo valore possibile con 32 bit senza segno è circa quattro miliardi), quindi i suoi bit più significativi sono memorizzati in ***edx*** e viene settato il flag ***cf***.

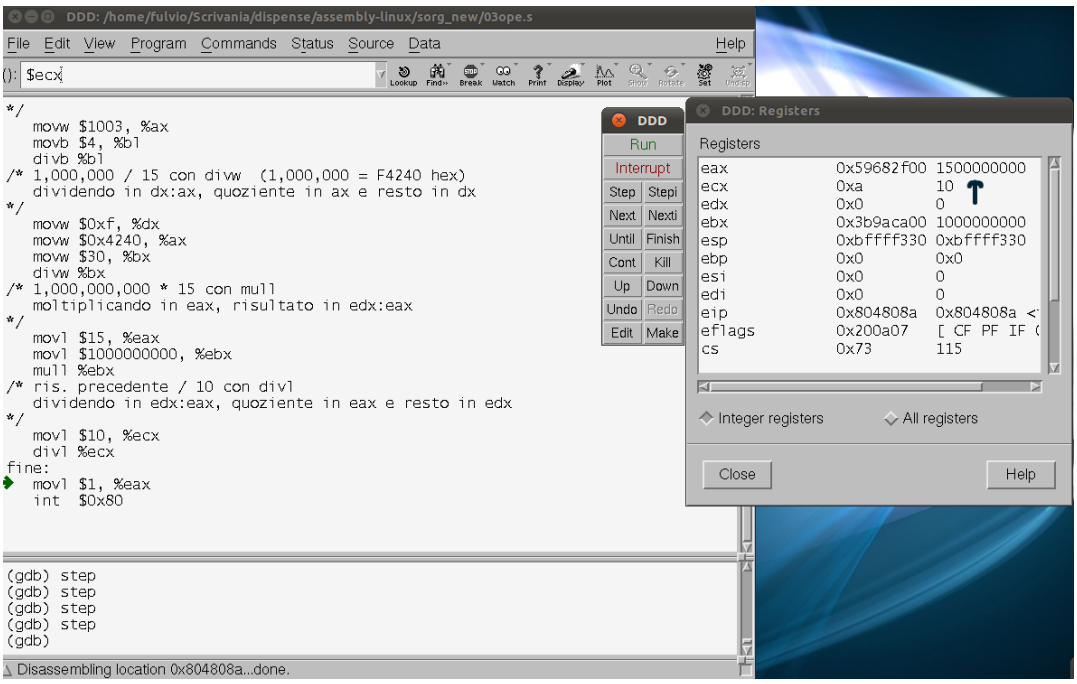
Osservando i valori contenuti in ***edx*** e ***eax*** dopo l'operazione (vedere figura 3.16) sembra che il risultato sia privo di senso, ma se consideriamo il valore espresso nella forma ***edx:eax*** e traduciamo le cifre esadecimali contenute nei due registri, e cioè $\boldsymbol{37e11d600_{16}}$, in binario otteniamo $\boldsymbol{1101111110000100011101011000000000_2}$ corrispondente al valore decimale ***15,000,000,000***, che è il risultato corretto.

Figura 3.16.



Ne abbiamo conferma anche dopo l'esecuzione della divisione con ***ecx***, caricato nel frattempo con il valore ***10*** (righe 39 e 40), che fornisce come risultato ***1,500,000,000*** (vedere figura 3.17).

Figura 3.17.



3.10 Uso della memoria

Iniziamo con questo paragrafo a considerare l'uso dei dati in memoria, basato sulla definizione delle etichette dati all'interno del segmento ***.data***.

Le etichette dati possono essere assimilate, per il loro ruolo, alle variabili che si usano nei linguaggi di programmazione ad alto livello: sono caratterizzate da un nome, da un tipo e da un contenuto.

Nell'esempio seguente vediamo la definizione di due etichette dati con nome ***dato1*** e ***dato2***, entrambe di tipo ***.long*** (cioè intere a 32 bit) e contenenti rispettivamente i valori iniziali ***100,000*** e ***200,000***.

```
.data
dato1:   .long 100000
dato2:   .long 200000
```

Nell'appendice A.0 è disponibile l'elenco delle direttive utili alla definizione dei vari tipi di dati possibili (interi più corti, reali, stringhe ecc.).

Nel caso si vogliano definire delle etichette dati con valore iniziale zero (come in alcuni dei prossimi esempi) si può anche usare il segmento «speciale», ***.bss*** dedicato proprio ai dati non inizializzati.

L'etichetta dati può avere un nome a piacere purché non coincidente con una parola riservata del linguaggio e possibilmente «significativo»; al momento della definizione l'etichetta deve essere seguita da «:».

Per usare il valore contenuto in una etichetta dati si deve racchiuderne il nome fra parantesi tonde; se invece si vuole fare riferimento all'indirizzo di memoria dell'etichetta si deve anteporre il simbolo «$» al nome.

Nell'esempio seguente viene mostrata la sintassi da usare in questi due casi con accanto l'equivalente operazione in linguaggio '**c**'.

```
.data
var1:   .byte 0
var2:   .long 0
.text
.globl _start
_start:
   movb $15, (var1)    # a=32;
   movl $var1, (var2) # c=&a; // con c definito come puntatore
```

Si ricordi che, nel caso di istruzioni assembly con due operandi, questi non possono essere entrambi etchette dati.

Questa limitazione dipende dall'esigenza di non avere istruzioni esecutive che debbano fare più di due accessi complessivi alla memoria.

Consideriamo le seguenti due istruzioni assembly in cui supponiamo che le due etichette siano definite come ***.byte***:

```
   addb %bl, (var1)
   addb (var1), (var2)
```

La prima è corretta in quanto l'esecutore deve compiere solo due accessi alla memoria: uno per leggere il contenuto di ***var1*** e uno per scriverci il risultato dopo aver sommato il valore precedente con quanto presente nel registro ***bl***.

La seconda invece è errata perché imporrebbe tre accessi alla memoria: due per leggere i valori contenuti nelle celle corrispondenti alle due etichette e uno per scrivere il risultato in ***var2***.

L'errore viene segnalato dall'assemblatore in sede di traduzione con un messaggio come questo: «Error: too many memory references for 'add'».

3.11 Lettura del contenuto della memoria con '**ddd**'

Vediamo adesso come usare ddd per leggere il contenuto dei dati in memoria.

A questo scopo riprendiamo il programma già esaminato nel paragrafo 3.8 a cui apportiamo una piccola modifica: il valore ***127*** non viene caricato nel registro ***bl*** ma nella posizione di memoria identificata dall'etichetta dati ***var1***; naturalmente anche le successive operazioni vengono effettuate su tale etichetta e non più sul registro [6]

```
1   /*
2   Descrizione:   Esempio di operazioni logiche con uso memoria
3   Autore:        FF
4   Data:          gg/mm/aaaa
5   */
6   .data
7   var1:     .byte 127
8   .text
9   .globl _start
10  _start:
11     nop
12     negb (var1)
```

```
13      notb (var1)
14      movb $15, %al
15      andb $35, %al
16      movb $15, %al
17      orb  $35, %al
18   fine:
19      movl $1, %eax
20      int  $0x80
```

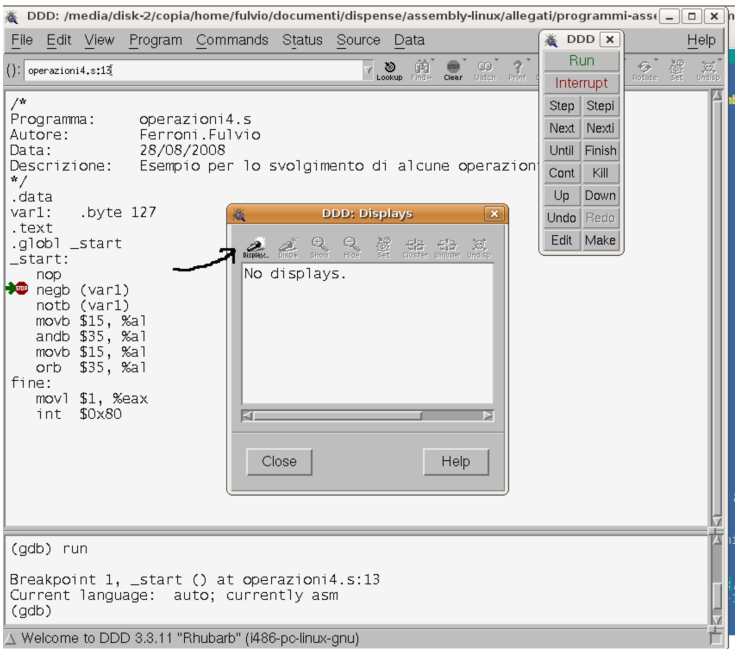
Le operazioni da svolgere sono le seguenti:

- eseguire:

  `$` **`ddd`** ***nome_eseguibile***

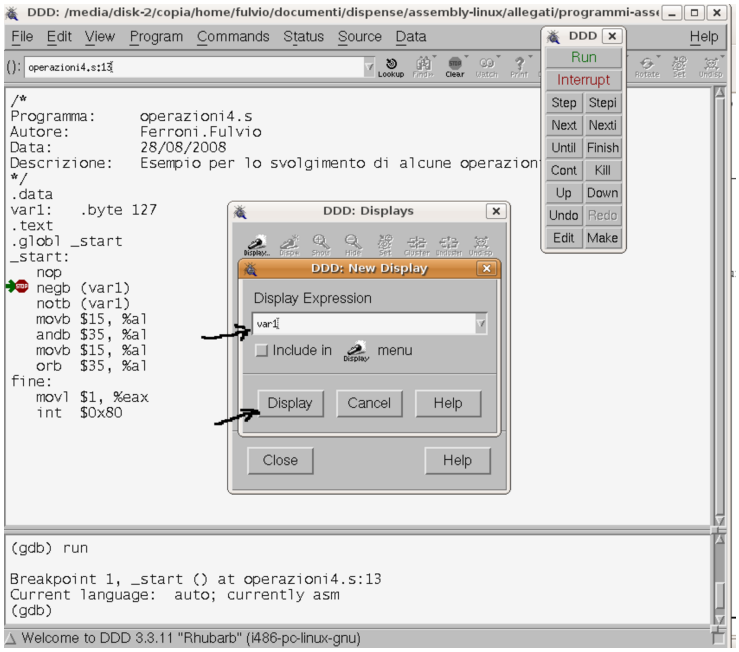
- posizionare il *breakpoint* sulla seconda riga eseguibile;
- lanciare il «Run» dalla finestrina «DDD» come al solito;
- selezionare la voce «Displays» dal menu «Data», ottenendo quanto mostrato nella figura 3.22;

  Figura 3.22.



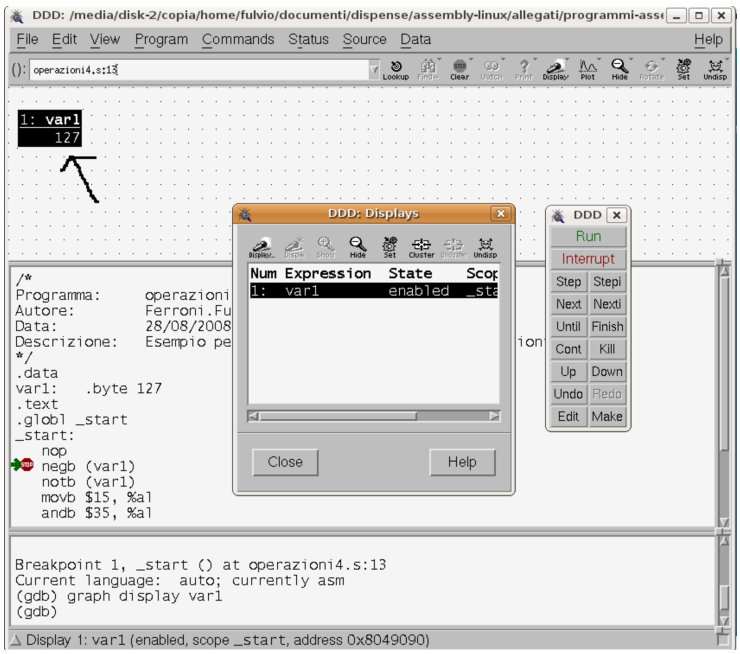
- *cliccare* sul pulsantino «Display» della finestra appena aperta in modo da ottenere una ulteriore finestra di dialogo in cui si deve scrivere il nome dell'etichetta dati da ispezionare, confermando poi con [*Invio*] o con il pulsante «Display» subito sotto (vedere la figura 3.23);

Figura 3.23.



- si apre in questo modo la sezione di visualizzazione dei dati contenente la variabile scelta come mostrato nella figura 3.24; naturalmente, in caso di bisogno, si possono aggiungere altre etichette da ispezionare seguendo lo stesso procedimento;

Figura 3.24.



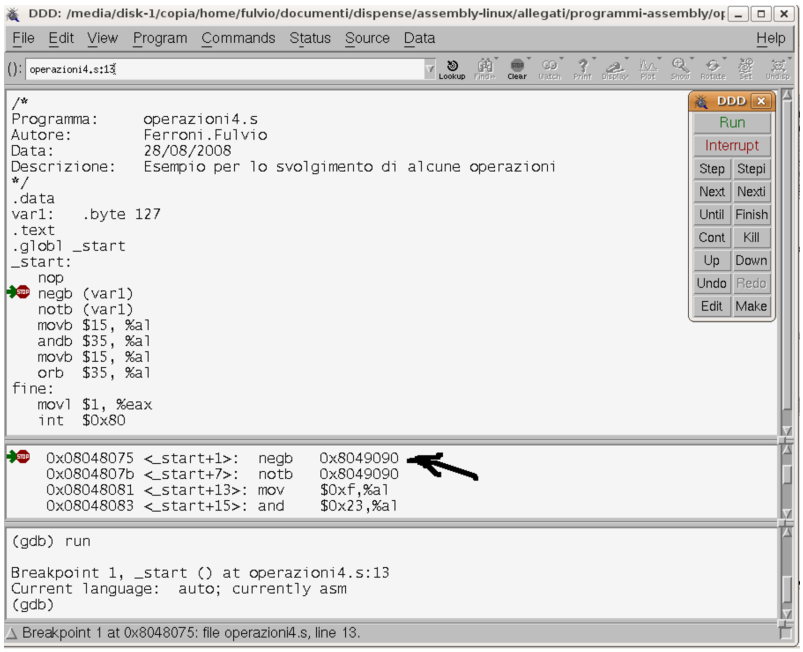
Lo stesso risultato si ottiene, in modo più rapido e agevole, facendo un [*doppio click*] sul nome della variabile che si vuole tenere sotto controllo nel listato mostrato da ddd.

- A questo punto continuando l'esecuzione con il pulsante «Step» si può vedere come cambia il valore della variabile prescelta in base alle operazioni svolte su di essa.

Non entriamo nei dettagli dei calcoli svolti dal programma in quanto li abbiamo già esaminati nel paragrafo 3.8.

Vediamo invece ulteriori potenzialità del programma ddd: attiviamo la finestra «Data Machine Code» in modo da avere visione degli indirizzi esadecimali corrispondenti alle etichette dati (figura 3.25).

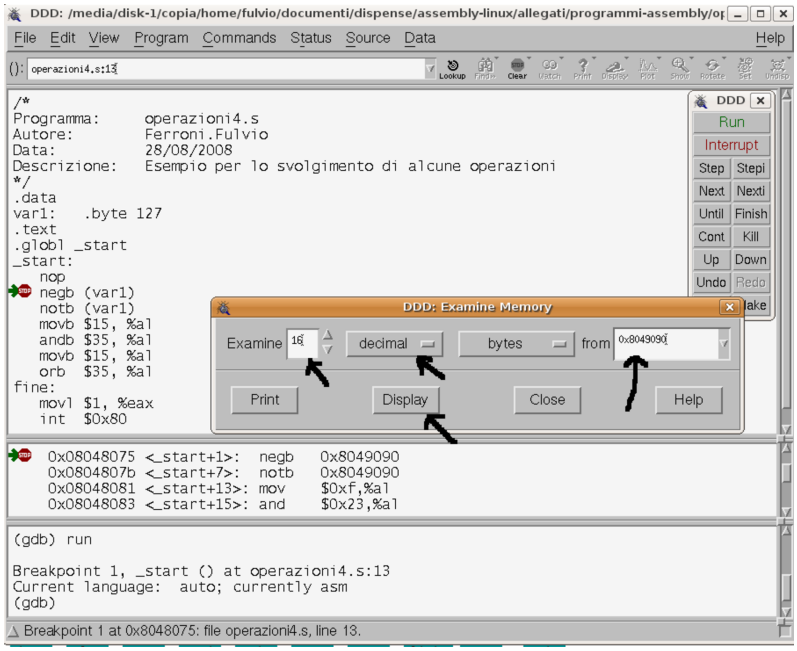
Figura 3.25.



Se si vuole ispezionare il contenuto della memoria a partire, ad esempio, dall'indirizzo assegnato all'etichetta ***var1***, che risulta essere $\boldsymbol{8049090_{16}}$ occorre operare come segue:

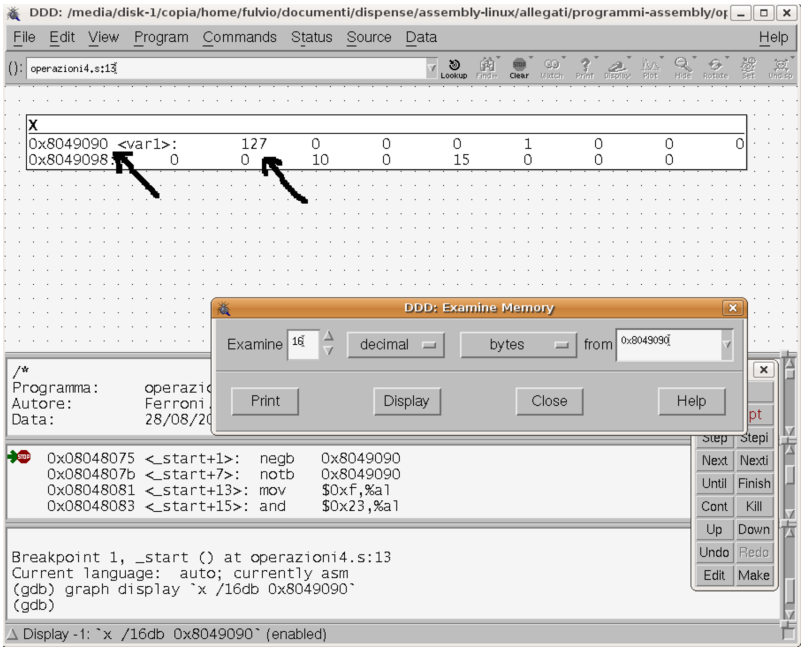
- attivare il *breakpoint* e lanciare il programma come di consueto con il pulsante «Run»;
- selezionare la voce «Memory» dal menu «Data», ottenendo quanto mostrato nella figura 3.26;

Figura 3.26.



- inserire la quantità di posizioni di memoria da visualizzare, la modalità (ad esempio in decimale) e l'indirizzo di partenza (in esadecimale con il prefisso «0x»), quindi premere il bottone «Display»;
- in questo modo si apre la finestra dei dati (per chiuderla si agisce sul menu «View») in cui viene visualizzato il contenuto delle celle di memoria selezionate, come mostrato nella figura 3.27;

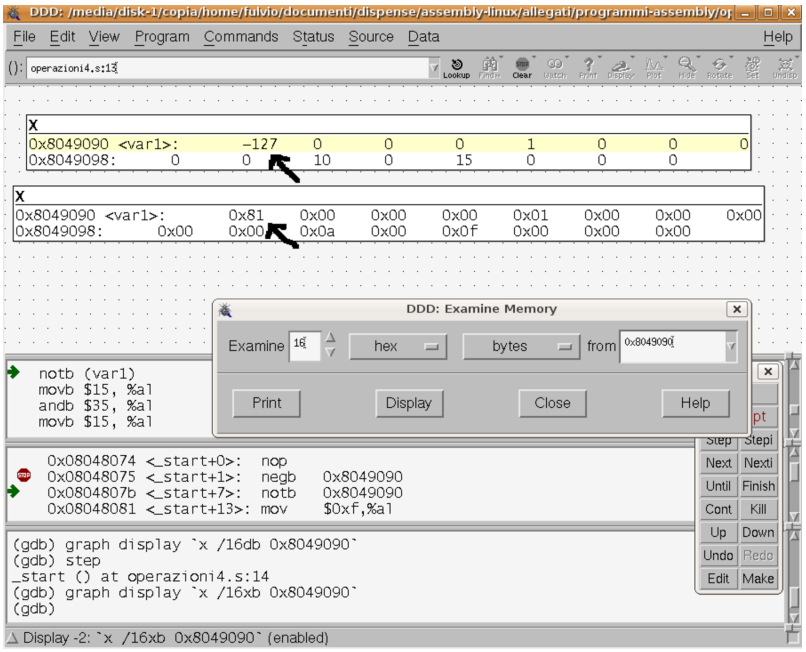
Figura 3.27.



- Naturalmente si possono aprire in modo analogo altre sezioni di visualizzazione di altre zone di memoria, o della stessa in modalità diverse; inoltre si ricordi che il contenuto delle etichette si può visionare anche con il metodo illustrato precedentemente.

- Continuando poi l'esecuzione con il pulsante «Step» si può vedere come cambiano i valori delle posizioni di memoria prescelte; ad esempio nella figura 3.28 vediamo la situazione dopo l'esecuzione dell'istruzione ***negb*** sia in decimale che in esadecimale.

Figura 3.28.



3.12 Somma con riporto e differenza con prestito

In questo paragrafo esaminiamo le operazioni di somma con riporto e sottrazione con prestito, usando dati definiti in memoria.[7]

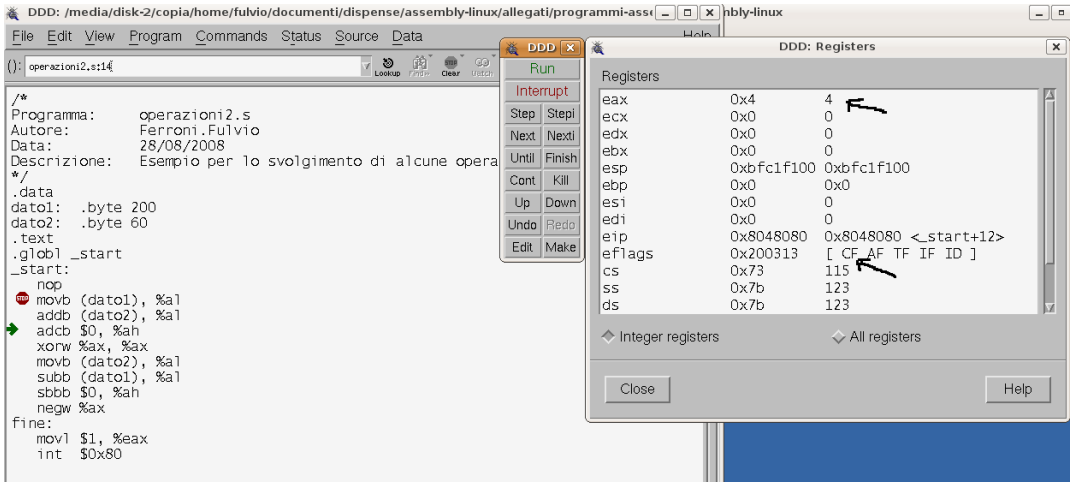
```
1   /*
2   Descrizione:   Esempio contenente somma con riporto
3                  e differenza con prestito
4   Autore:        FF
5   Data:          gg/mm/aaaa
6   */
7   .data
8   dato1:  .byte 200
9   dato2:  .byte 60
10  .text
11  .globl _start
12  _start:
13     nop
14     movb (dato1), %al
15     addb (dato2), %al
16     adcb $0, %ah
17     xorw %ax, %ax
18     movb (dato2), %al
19     subb (dato1), %al
20     sbbb $0, %ah
21     negw %ax
22  fine:
23     movl $1, %eax
24     int  $0x80
```

Anche stavolta il segmento dati non è vuoto ma contiene alle righe 8 e 9 le definizioni delle etichette di memoria con i valori su cui operare,

Nelle righe da 14 a 16 si effettua la somma tra i due dati; è necessaria l'operazione ***adcb*** dopo la somma perché il risultato (***260***) esce dall'intervallo dei valori senza segno rappresentabili con otto bit e quindi si deve sommare il riporto nei bit del registro immediatamente a sinistra e cioè in ***ah***.

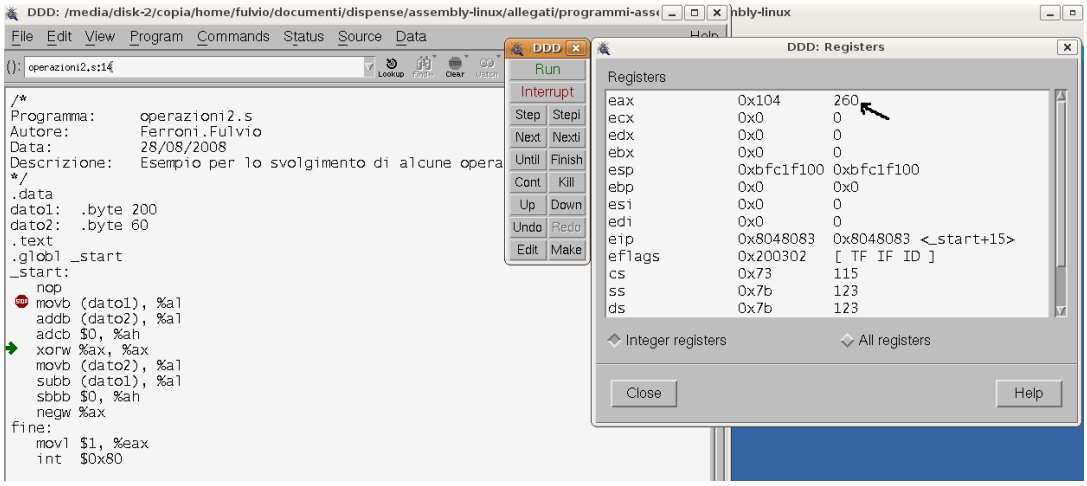
La figura 3.22 mostra il risultato, in apparenza errato, e la situazione dei bit di stato dopo la somma presente a riga 15.

Figura 3.30.



Nella figura 3.31 invece il risultato appare corretto dopo la somma con riporto fatta a riga 16.

Figura 3.31.



A riga 17 vediamo l'azzeramento di un registro grazie ad una operazione di ***xor*** su se stesso; altre possibilità per ottenere lo stesso risultato, anche se in maniera meno «elegante», possono essere: ***subw %ax, %ax*** oppure ***movw $0, %ax***.

Con le righe da 18 a 20 viene sottratto ***dato1*** da ***dato2***; qui è necessario tenere conto del prestito sui bit immediatamente a sinistra, e cioè in ***ah***, con l'operazione ***sbbb***.

Anche in questo caso vediamo, nella figura 3.32, la situazione dopo la sottrazione con il risultato apparentemente errato e poi, nella figura 3.33, il risultato corretto, con il bit di segno attivato.

Figura 3.32.

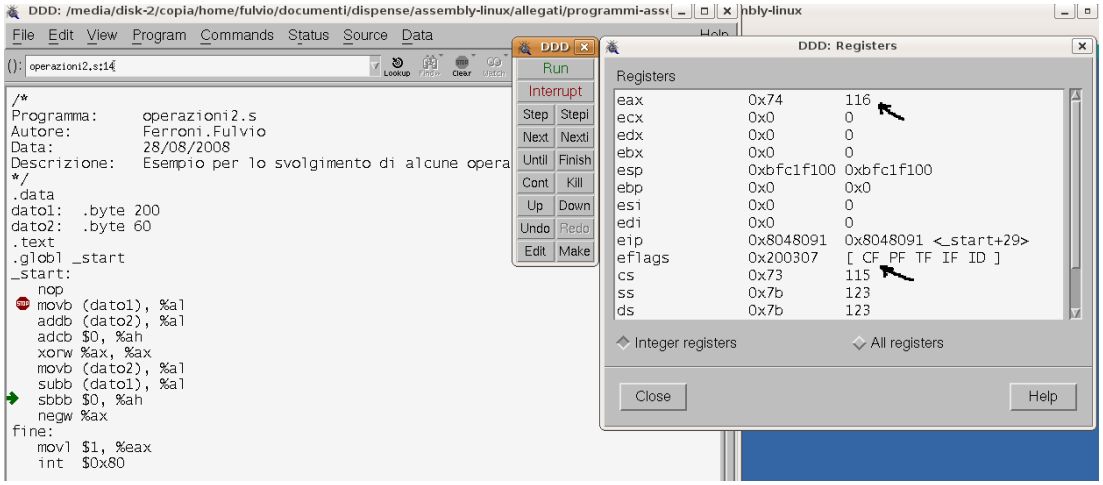
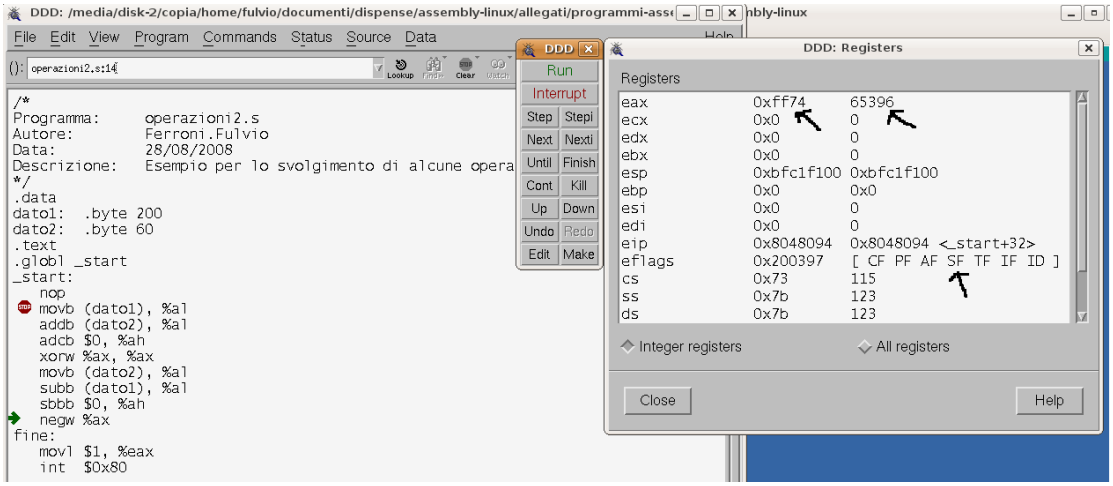


Figura 3.33.

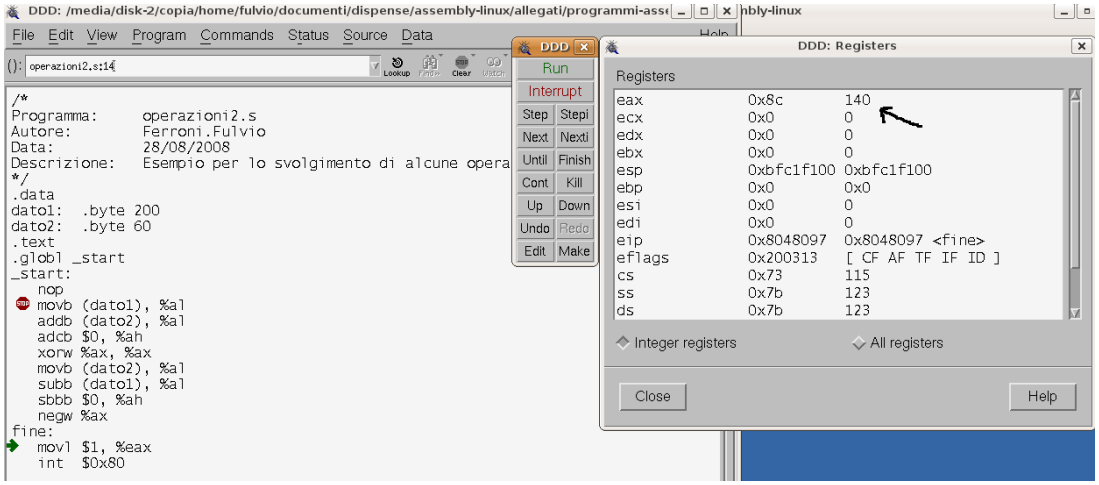


In realtà il valore che appare nel registro ***eax*** pare ben diverso da quello che ci saremmo aspettati (***-140***) ma, ricordando le modalità di rappresentazione dei valori interi nell'elaboratore, abbiamo:

$\mathbf{ff74_{16}} = \mathbf{1111111101110100_2}$ = complemento a due di $\mathbf{0000000010001100_2}$ che vale ***140*** e quindi il valore rappresentato è ***-140***.

Per convincerci ancora di più della correttezza del risultato osserviamo, nella figura 3.34, l'effetto dell'operazione di negazione (o di complemento a due) fatta sul registro ***ax*** alla riga 21.

Figura 3.34.



3.13 Salti incondizionati e condizionati

I salti servono ad alterare la normale sequenza di esecuzione delle istruzioni all'interno di un programma; il salto avviene sempre con «destinazione» un'etichetta istruzioni che deve essere definita (una volta sola) nel programma in qualsiasi posizione, indifferentemente prima o dopo l'istruzione di salto.

L'etichetta istruzioni può avere un nome a piacere purché non coincidente con una parola riservata del linguaggio e possibilmente «significativo»; al momento della definizione l'etichetta deve essere seguita da «:».

Come mostrato più avanti, nei programmi assembly i salti si usano per poter costruire le strutture di programmazione di selezione iterazione per le quali il linguaggio non mette a disposizione istruzioni apposite.

Il salto incondizionato, realizzabile con l'istruzione ***jmp***, viene effettuato in ogni caso, a prescindere da qualsiasi condizione si sia venuta a creare a causa delle operazioni precedenti; i salti condizionati, dei quali esistono varie versioni elencate nell'appendice A, sono eseguiti solo se si è verificata la condizione richiesta dal tipo si salto.

Nell'esempio seguente viene mostrato un programma che non compie alcuna elaborazione significativa ma che contiene un'istruzione di salto incondizionato, al fine di mostrarne il comportamento con l'uso del debugger.[8]

```
1  /*
2  Programma:     salto_inc.s
3  Autore:        FF
4  Data:          gg/mm/aaaa
5  Descrizione:   Esempio con salto incondizionato
6  */
7  .data
8  var1:    .byte 127
9  .text
10 .globl _start
11 _start:
12    nop
13    negb (var1)
14    notb (var1)
```

```
15      jmp fine
16      nop
17      nop
18  fine:
19      movl $1, %eax
20      int  $0x80
```

Come detto il programma non serve ad alcun scopo elaborativo; dopo un paio di operazioni sul valore ***var1*** c'è un salto incondizionato all'etichetta ***fine:*** e quindi le due istruzioni ***nop*** alle righe 16 e 17 non vengono eseguite (non che cambi molto, visto che sono istruzioni «vuote»).

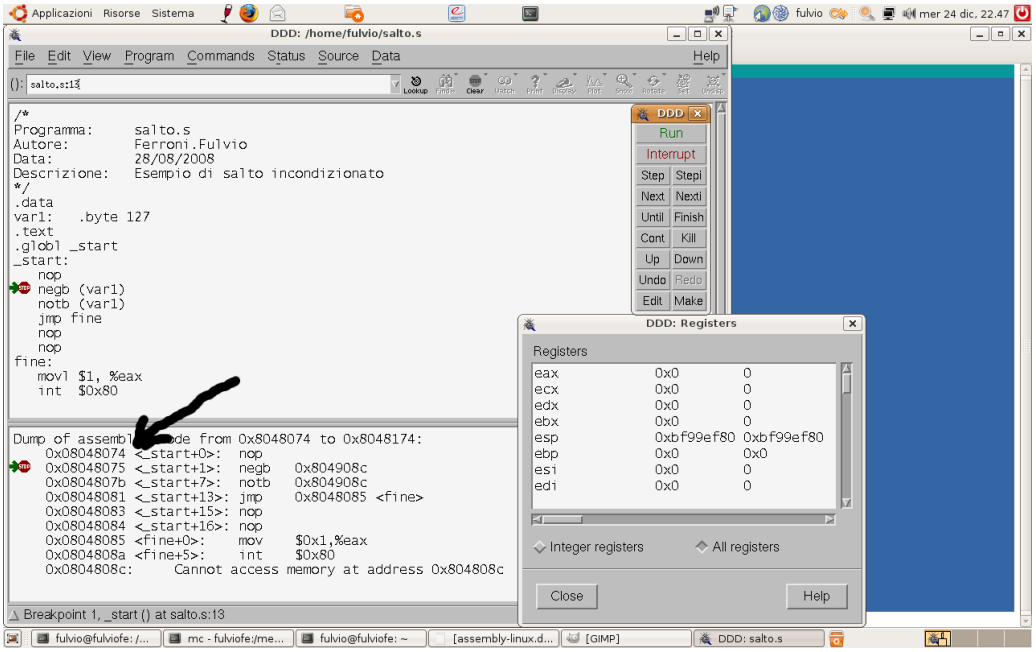
Vale però la pena osservare l'esecuzione con il debugger ddd inserendo il *breakpoint* come mostrato in precedenza e attivando, oltre alla finestra dei registri anche quella del «Data Machine Code» dal menu «View».

In questa finestra è disponibile il listato in una forma più vicina a quella del linguaggio macchina, dove tutti i valori sono espressi in esadecimale e i riferimenti alla memoria sono indicati tramite gli indirizzi e non con le etichette.

Nella figura 3.36, vediamo, proprio nella finestra del codice macchina, gli indirizzi in cui sono memorizzate le istruzioni del nostro programma; dal fatto che la «distanza» in memoria fra di esse è variabile riceviamo conferma che le istruzioni non sono tutte della stessa lunghezza.

Questo è un fatto normale per i processori di tipo CISC (*Complex Instruction Set Computing*) come gli Intel e compatibili; nel nostro esempio notiamo che l'istruzione ***nop*** è lunga un byte, la ***jmp*** due byte, la ***movl*** cinque byte, la ***negb*** e la ***notb*** sei byte.

Figura 3.36.



Nelle successive due figure 3.37 e 3.38 si può notare il valore del registro contatore di programma ***eip*** subito prima e subito dopo l'esecuzione dell'istruzione di salto incondizionato; da come cambiano i valori di tale registro si può capire che il modo con cui la macchina esegue un'istruzione

di salto è molto banale: viene «semplicemente» inserito nel contatore di programma l'indirizzo dell'istruzione a cui saltare invece che quello della prossima istruzione nella sequenza del programma (nel nostro esempio sarebbe la ***nop*** a riga 16).

Figura 3.37.

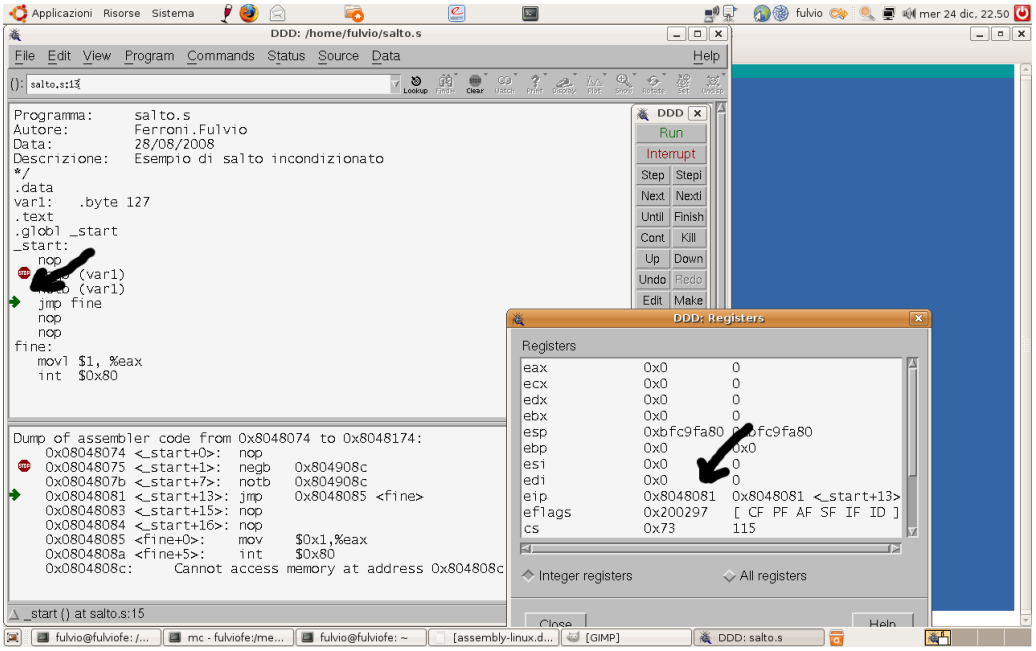
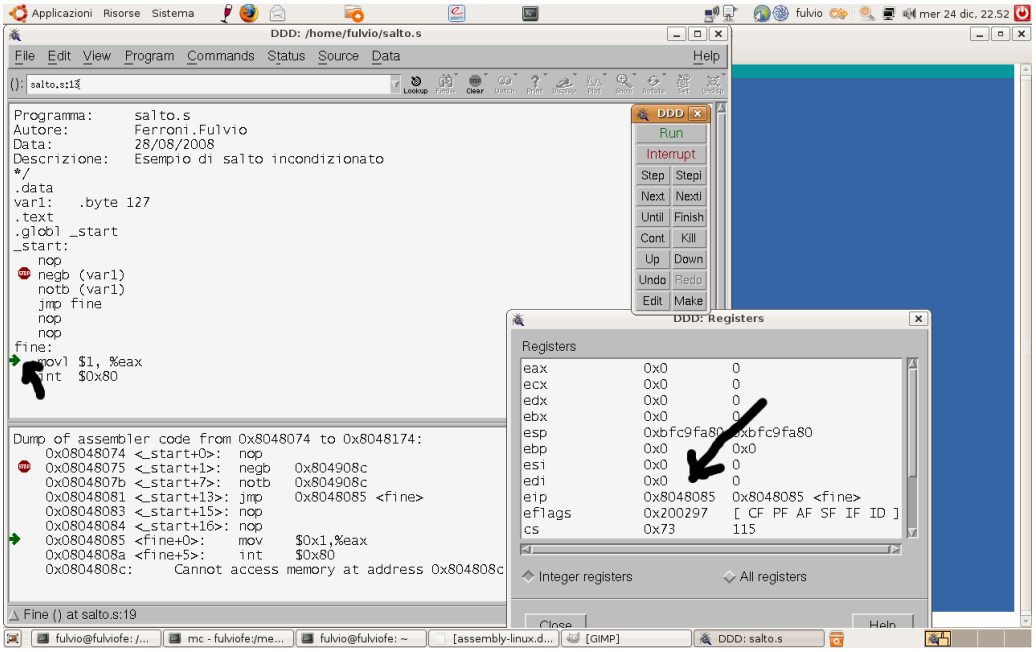


Figura 3.38.



Il salto incondizionato può a tutti gli effetti essere assimilato alla «famigerata» istruzione ***goto*** presente in molti linguaggi di programmazione ad alto livello, specialmente fra quelli di vecchia concezione.

L'uso di tale istruzione, anche nei linguaggi che la prevedono, è solitamente sconsigliato a vantaggio delle tecniche di programmazione strutturata.

L'approfondimento di questi concetti esula dagli scopi di queste dispense; ricordiamo solo che secondo i dettami della programmazione strutturata un algoritmo (e quindi il programma che da esso deriva) deve contenere solo tre tipi di strutture di controllo:

- '**`sequenza`**': sempre presente in quanto qualsiasi algoritmo è per sua natura «una '**`sequenza`**' di operazioni da svolgere su dei dati di partenza per ottenere dei risultati»;
- '**`selezione (a una via o a due vie)`**': permette di eseguire istruzioni diverse al verificarsi o meno di una certa condizione; nei linguaggi ad alto livello si realizza di solito con il costrutto ***if***;
- '**`iterazione con controllo in testa o con controllo in coda`**': permette di realizzare i cicli grazie ai quali si ripete l'esecuzione di un certo insieme di istruzioni; nei linguaggi ad alto livello ci sono vari tipi di costrutti utili a questo scopo, come la ***for***, il ***while***, il ***do ... while*** e altri ancora a seconda del linguaggio considerato.

Per quanto riguarda la programmazione in assembly non ci sarebbe alcun impedimento tecnico riguardo l'uso del salto incondizionato come se fosse un ***goto***; dal punto di vista concettuale è però sicuramente opportuno, e anche molto istruttivo, cercare di rispettare i dettami della programmazione strutturata anche in questo ambito.

La strategia migliore è quindi quella di stendere sempre in anticipo un algoritmo strutturato relativo alla soluzione del problema in esame e poi convertirlo in un sorgente assembly.

Questo procedimento ha senso a patto che il problema non sia davvero banale o risolvibile con una semplice sequenza di istruzioni (come nel caso degli esempi del paragrafo precedente) e implica l'uso di tecniche come i '**`diagrammi di flusso`**' o la '**`pseudo-codifica`**'.

In queste dispense la fase preliminare verrà quasi del tutto trascurata anche perché la capacità di risolvere problemi per via algoritmica è da considerare un prerequisito per avvicinarsi alla programmazione a basso livello.

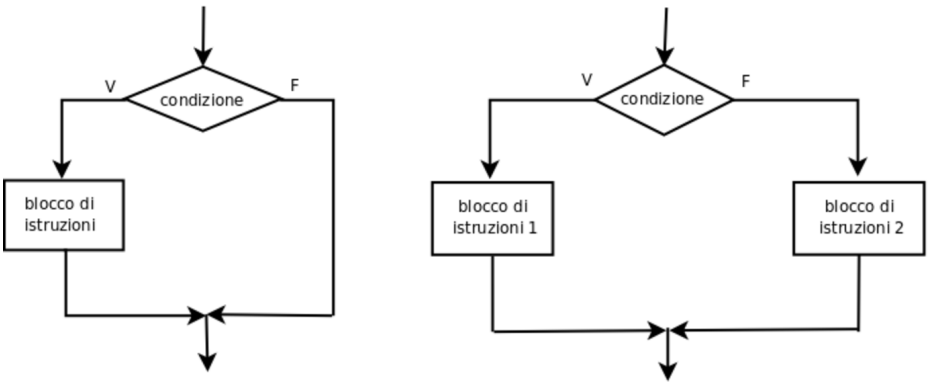
Purtroppo scrivere programmi strutturati in assembly non è semplice perché il linguaggio non mette a disposizione alcuna istruzione assimilabile ad una selezione o ad una iterazione (con l'eccezione dell'istruzione ***loop***, che vedremo tra breve).
Si devono quindi realizzare tali costrutti «combinando» opportunamente l'uso di salti condizionati e incondizionati.

3.13.1 Realizzazione di strutture di selezione con i salti

Consideriamo la struttura di selezione nelle sue due varianti: «a una via» e «a due vie».

Esse si possono rappresentare nei diagrammi di flusso come mostrato nella figura 3.39.

Figura 3.39.



Per la loro realizzazione in linguaggio assembly occorre servirsi dei salti condizionati e incondizionati; nei prossimi listati, in cui si fa uso di istruzioni espresse in modo sommario, usando la lingua italiana, viene mostrata la logica da seguire.

Per la selezione a una via:

```
   istruzioni prima della selezione

   confronto
   salto condizionato a finese

      istruzioni del corpo della selezione

finese:

   istruzioni dopo la selezione
```

Per la selezione a due vie:

```
   istruzioni prima della selezione

   confronto
   salto condizionato a altrimenti

      istruzioni di un ramo della selezione
      salto incondizionato a finese

altrimenti:

      istruzioni dell'altro ramo della selezione

finese:

   istruzioni dopo la selezione
```

Come esempio di uso di selezione a una via consideriamo la divisione tra i due numeri ***num1*** e ***num2*** da effettuare solo se il secondo è diverso da zero.

Siccome non abbiamo ancora le nozioni indispensabili per gestire le fasi di input e output dei dati in assembly, il programma non presenta alcun risultato a video e quindi, per sincerarsi del suo corretto funzionamento, occorre eseguirlo con il debugger; questo ovviamente vale anche per i prossimi esempi fino al momento in cui illustreremo le modalità di visualizzazione dei dati.

Non essendo (al momento) possibile fornire dati al programma durante l'esecuzione e riceverne i risultati, i due valori di input vengono inseriti fissi nel sorgente e il risultato viene depositato nel registro ***cl***.[9]

```
1   /*
2   Programma:     selezione1.s
3   Autore:        FF
4   Data:          gg/mm/aaaa
5   Descrizione:   Esempio con selezione a una via
6   */
7   .data
8   num1:   .byte  125
9   num2:   .byte  5
10  .text
11  .globl _start
12  _start:
13     movb $0, %bl
14     cmpb (num2), %bl
15     je finese
16     xorw %ax, %ax
17     movb (num1), %al
18     divb (num2)
19     movb %al, %cl
20  finese:
21  fine:
22     movl $1, %eax
23     int  $0x80
```

Il programma è molto banale: alla riga 14 effettua il confronto tra il secondo numero e il registro ***bl*** dove in precedenza ha caricato zero; alla riga 15 si ha un salto condizionato all'etichetta ***fine:*** se i valori risultano uguali.

In caso contrario alle righe da 16 a 19 si effettua la divisione con divisore ***num2***, ponendo il dividendo ***num1*** in ***ax*** e il risultato ottenuto in ***cl***.

Come accennato in precedenza, l'istruzione ***cmp*** viene «tradotta» in una differenza tra i due valori da confrontare; per questo motivo il sistema decide se effettuare il salto condizionato ***je*** testando il flag di zero (che deve risultare a valore uno) del registro di stato: infatti se tale flag vale 1 significa che la differenza tra i valori, fatta per realizzarne il confronto, ha fornito risultato zero e quindi essi sono uguali.

Per mostrare l'uso della selezione a due vie consideriamo la differenza tra due numeri in valore assoluto (fatta cioè in modo che il risultato sia sempre positivo); anche in questo caso i dati di input sono fissi a programma mentre il risultato viene posto in memoria usando l'etichetta ***ris***.[10]

```
/*
Programma:     selezione2.s
Autore:        FF
```

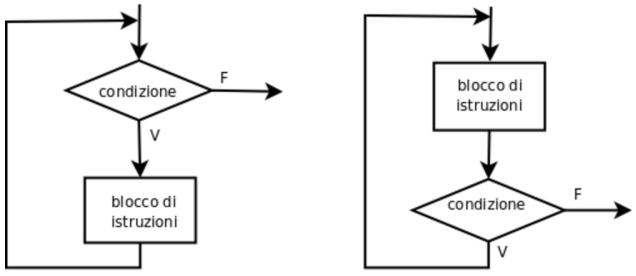
```
Data:          gg/mm/aaaa
Descrizione:   Esempio con selezione a due vie
*/
.data
num1:   .byte  247
num2:   .byte  210
ris:    .byte  0
.text
.globl _start
_start:
   movb (num2), %al
//iniziose
   cmpb (num1), %al
   jl altrimenti     # se %al (cioe' num2) piu' piccolo salta
   movb (num1), %cl
   subb %cl, %al     # sottrae %cl (cioe' num1) da %al (cioe' num2)
   jmp  finese
altrimenti:
   movb (num1), %cl
   subb (num2), %cl  # sottrae num2 da %cl (cioe' num1)
finese:
   movb %cl, (ris)
fine:
   movl $1, %eax
   int  $0x80
```

In questo caso non illustriamo le istruzioni del programma, reputando sufficienti, data la sua estrema semplicità, i commenti inseriti direttamente nel listato.

3.13.2 Realizzazione di strutture di iterazione con i salti

Anche per l'iterazione esistono due varianti: «con controllo in testa» e «con controllo in coda»; le possiamo rappresentare nei diagrammi di flusso come mostrato nella figura 3.44.

Figura 3.44.



Vediamo la logica da seguire per realizzare i due tipi di ciclo usando le istruzioni di salto.

Per il ciclo con controllo in testa:

```
   istruzioni prima del ciclo

iniziociclo:
   confronto
   salto condizionato a fineciclo:

      istruzioni del corpo del ciclo
      salto incondizionato a iniziociclo

fineciclo:

   istruzioni dopo il ciclo
```

Per il ciclo con controllo in coda:

```
   istruzioni prima del ciclo

iniziociclo:

      istruzioni del corpo del ciclo

   confronto
   salto condizionato a iniziociclo

   istruzioni dopo il ciclo
```

Come esempi di uso dei cicli mostriamo due programmi: il primo per il calcolo del prodotto tra due numeri maggiori o uguali a zero, svolto tramite una successione di somme; il secondo per il calcolo della differenza tra due numeri maggiori di zero, ottenuta sottraendo ciclicamente una unità finché il più piccolo si azzera.

In entrambi i casi la spiegazione delle istruzioni svolte viene fatta attraverso i commenti contenuti nei listati e non è quindi necessaria la loro numerazione.

Nel primo programma i due numeri da moltiplicare sono ***num1*** e ***num2*** assegnati nel programma, il risultato viene depositato nel registro ***bx***.[11]

```
/*
Programma:     iterazione1.s
Autore:        FF
Data:          gg/mm/aaaa
Descrizione:   Esempio con iterazione con controllo in testa:
               prodotto come successione di somme
*/
.data
num1:   .byte  50
num2:   .byte  6
.text
.globl _start
_start:
   xorb %cl, %cl       # azzero %cl (contatore)
   xorw %bx, %bx       # azzero %bx (accumulatore)
```

```
iniziociclo:
   cmpb (num2), %cl
   je  fineciclo       # se %cl = num2 il ciclo è finito
      addb (num1), %bl # accumulo il valore di num1 in %bl
      adcb $0, %bh     # considero eventuale riporto
      incb %cl         # incremento del contatore
      jmp iniziociclo
fineciclo:
fine:
   movl $1, %eax
   int  $0x80
```

Nel secondo programma i valori da sottrarre sono ***val1*** e ***val2*** e, sebbene siano fissi nel programma, si suppone di non conoscere a priori quale sia il minore; il risultato viene depositato in ***ris***.[12]

```
/*
Programma:     iterazione2.s
Autore:        FF
Data:          gg/mm/aaaa
Descrizione:   Esempio con iterazione con controllo in coda:
               differenza tra numeri positivi sottraendo ciclicamente 1
*/
.data
val1:   .byte  5
val2:   .byte  26
ris:    .byte  0
.text
.globl _start
_start:
// con una selezione a due vie pone in %al il valore maggiore e in %bl il minore
   movb (val2), %dl    # val2 va in %dl, non si fanno confronti tra etichette
   cmpb (val1), %dl
   jl altrimenti
      movb %dl, %al
      movb (val1), %bl
      jmp finese
altrimenti:
      movb (val1), %al
      movb %dl, %bl
finese:
   xorb %cl, %cl
iniziociclo:
      decb %al       # tolgo 1
      decb %bl
   cmpb %bl, %cl
   jne iniziociclo   # se %bl non zero il ciclo continua
   movb %al, (ris)   # valore residuo di %al = differenza
fine:
   movl $1, %eax
   int  $0x80
```

3.13.3 Iterazioni con l'istruzione *loop*

L'istruzione ***loop*** permette di creare «iterazioni calcolate» in cui viene usato un contatore gestito automaticamente dal sistema; si ottiene qualcosa di simile all'uso del costrutto ***for*** presente in tutti i linguaggi di programmazione ad alto livello.

Il criterio di funzionamento è il seguente:

```
   istruzioni prima del ciclo

   impostazione del numero di iterazioni in %cx

iniziociclo:

      istruzioni del corpo del ciclo
      loop iniziociclo

   istruzioni dopo il ciclo
```

In pratica l'istruzione ***loop*** svolge automaticamente le seguenti due operazioni:

- decrementa di uno %cx (***dec %cx***);
- salta se non zero all'etichetta di inizio ciclo (***jnz iniziociclo***).

Come esempio vediamo il programma per il calcolo del quadrato di un numero positivo ***n*** ottenuto come somma dei primi ***n*** numeri dispari.

Il valore ***n*** è, come al solito fisso nel programma mentre il risultato viene posto in ***q***; anche in questo caso i commenti sono inseriti direttamente nel listato.[13]

```
/*
Programma:     iterazione3.s
Autore:        FF
Data:          gg/mm/aaaa
Descrizione:   Esempio con iterazione calcolata (loop):
               quadrato di n come somma dei primi n numeri dispari
*/
.data
n:   .word  5
q:   .word  0
.text
.globl _start
_start:
   movw (n), %cx
   movw $1, %ax     # imposta la variabile per scorrere i numeri dispari
   xorw %bx, %bx    # azzera registro dove accumulare i valori
iniziociclo:
      addw %ax, %bx
      addw $2, %ax  # prossimo valore dispari
      loop iniziociclo
   movw %bx, (q)
fine:
   movl $1, %eax
   int  $0x80
```

Anche per l'istruzione ***loop*** è prevista la presenza dei suffissi (ad eccezione del suffisso ***b***); abbiamo quindi:

- ***loopw***, che utilizza il registro %cx;
- ***loopl***, che utilizza il registro %ecx.

Come abbiamo notato in precedenza, anche se, in assenza del suffisso, l'assemblatore è in grado di utilizzare la versione appropriata dell'istruzione in base al contesto di utilizzo, è sempre bene farne uso per una maggiore leggibilità dei sorgenti .

3.14 Gestione dello *stack*

Lo *stack* o '**`pila`**' è un'area di memoria, a disposizione di ogni programma assembly, che viene gestita in modalità LIFO (*Last In First Out*).

Questo significa che gli inserimenti e le estrazioni di elementi nella pila avvengono alla stessa estremità detta *top* o '**`cima`**' e quindi i primi elementi a uscire sono gli ultimi entrati (proprio come in una pila di piatti).

Lo stack ha un'importanza fondamentale soprattutto quando si usano le procedure in assembly, come vedremo nel paragrafo 3.19 ad esse dedicato; può essere utile però anche in tutti quei casi in cui serva memorizzare temporaneamente dei dati: ad esempio per «liberare» dei registri da riutilizzare in altro modo.

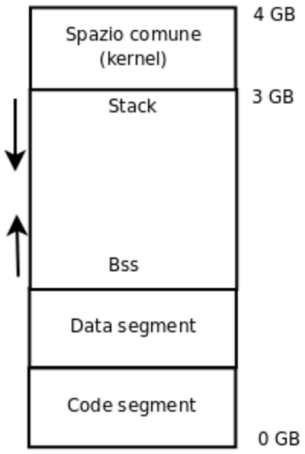
Ci sono tre registri che riguardano questa area di memoria:

- ***ss*** (*stack segment*): contiene l'indirizzo iniziale del segmento stack (o il relativo selettore se parliamo di processori IA-32) e non viene manipolato direttamente nei programmi assembly;
- ***sp*** (*stack pointer*): contiene l'indirizzo della cima dello stack facendo riferimento al registro ***ss***, è cioè l'offset rispetto all'inizio del segmento;
- ***bp*** (*base pointer*): è un altro registro di offset per il segmento stack e viene usato per gestire i dati nella pila senza alterare il valore del registro ***sp*** (vedremo che questo sarà importante soprattutto nell'uso delle procedure); il registro ***bp*** in realtà può essere usato come offset di qualsiasi altro segmento (il segmento stack è la scelta predefinita) ma in tal caso occorre specificarlo scrivendo l'indirizzo nella forma completa ***regseg:bp*** o ***selettore:bp***.

Una osservazione importante deve essere fatta riguardo all'organizzazione degli indirizzi nella pila: i dati sono inseriti a partire dall'indirizzo finale del segmento e il riempimento avviene scendendo ad indirizzi più bassi; il registro ***sp***, quindi, ha inizialmente il valore più alto possibile e decresce e aumenta ad ogni inserimento e estrazione di dati.

Nella figura 3.51 viene mostrata l'organizzazione della memoria virtuale di 4 GB assegnata a ogni processo.

Figura 3.51.



Ricordiamo però che, come detto nel paragrafo 1.2 e come emerge anche dallo schema proposto, solo i primi 3 GB sono davvero a disposizione del processo.

Le frecce significano che l'area di memoria che ospita il segmento ***bss*** cresce verso l'alto, mentre lo stack cresce verso il basso.

Le istruzioni per la gestione dello stack sono:

- ***pop***: per prelevare un dato dallo stack e porlo nel registro usato come operando;
- ***push***: per inserire un dato nello stack prelevandolo dal registro usato come operando.

Entrambe le istruzioni sono disponibili solo per dati di 16 o 32 bit (suffissi ***w*** e ***l***).

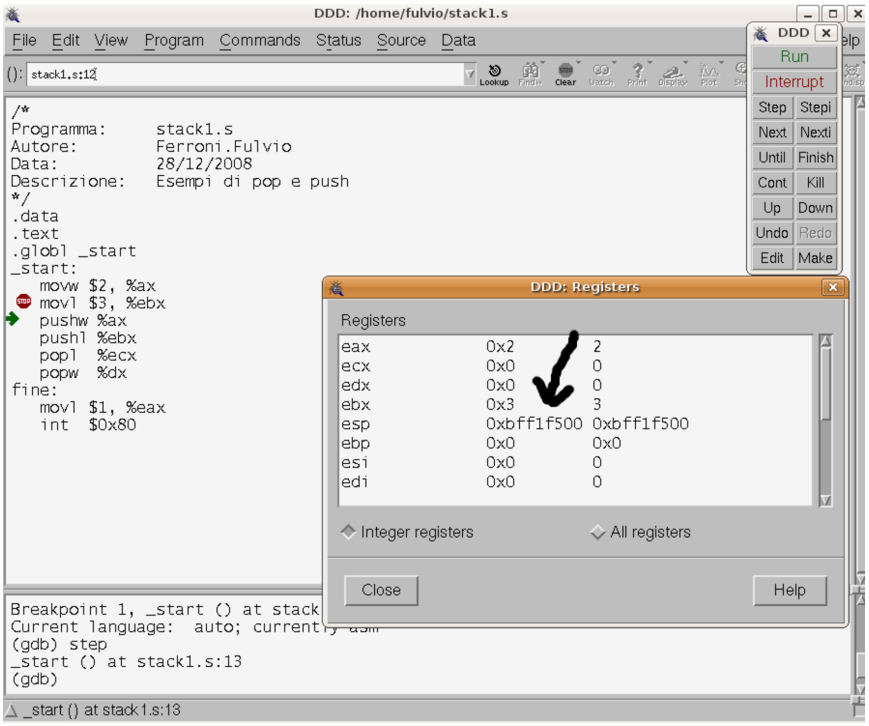
Vediamo un piccolo esempio in cui evidenziamo (usando il debugger) le variazioni che subisce il registro ***sp*** ad ogni inserimento o estrazione di dati; il programma non ha alcuno scopo concreto e contiene solo alcune ***pop*** e ***push***. [14]

```
/*
Programma:     stack1.s
Autore:        FF
Data:          gg/mm/aaaa
Descrizione:   Esempi di pop e push
*/
.data
.text
.globl _start
_start:
   movw $2, %ax
   movl $3, %ebx
   pushw %ax
   pushl %ebx
   popl  %ecx
   popw  %dx
fine:
   movl $1, %eax
   int  $0x80
```

Il programma esegue solo due inserimenti nello stack e poi preleva i dati inseriti; è ovvio che se l'ultimo inserito è una dato ***long***, tale deve essere anche il primo estratto; in caso contrario non si riceve alcuna segnalazione di errore ma i dati estratti saranno errati.

Nella figura 3.53 vediamo la situazione dei registri prima del primo inserimento nella pila.

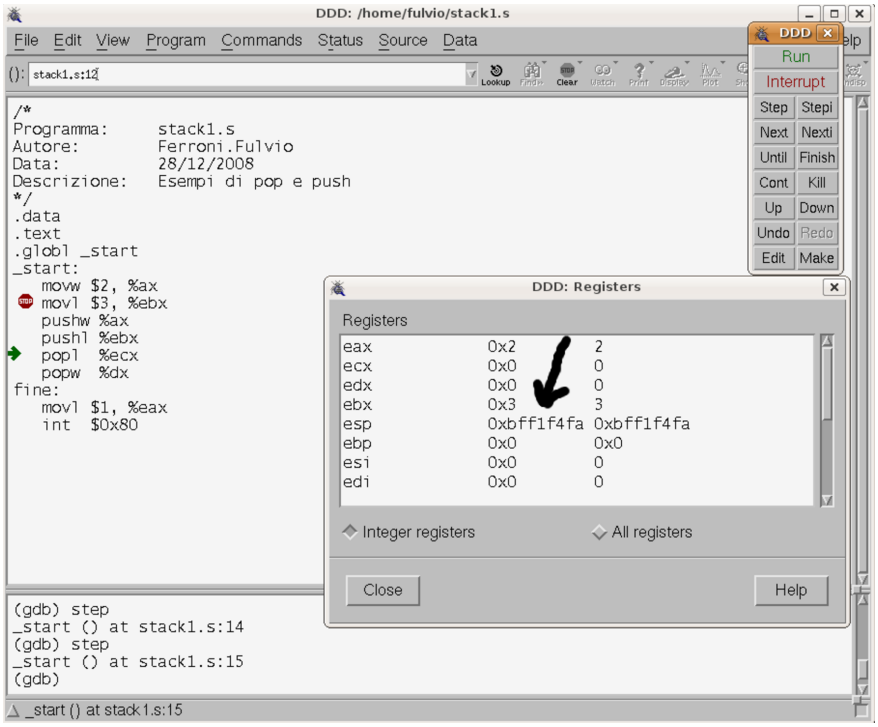
Figura 3.53.



In particolare notiamo il valore del registro ***esp*** che è di poco superiore a tre miliardi; questo ci conferma che lo stack è posizionato verso la fine dello spazio virtuale di 3 GB disponibile per il programma.

Nella figura 3.54 ci spostiamo alla situazione dopo i due inserimenti.

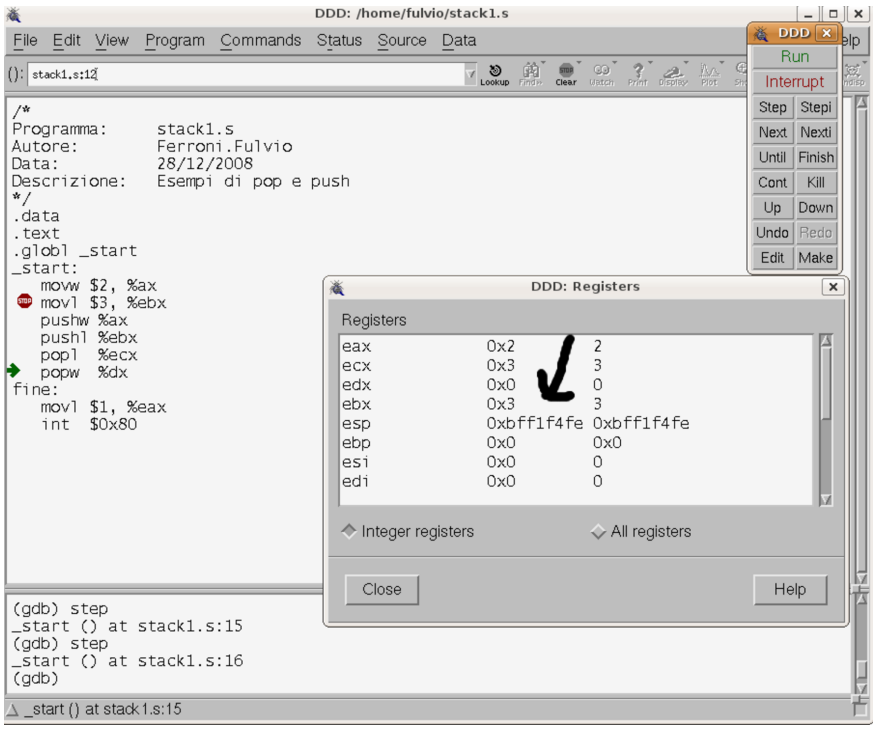
Figura 3.54.



Il valore del puntatore alla pila è sceso di sei e ciò rispecchia il fatto che abbiamo inserito un dato lungo due byte ed uno lungo quattro.

Infine nella figura 3.55 vediamo la situazione dopo la prima estrazione con il puntatore alla pila che è risalito di quattro unità (abbiamo estratto un dato lungo quattro byte).

Figura 3.55.



Vediamo adesso un esempio di uso concreto dello stack per la gestione di due cicli annidati composti da un numero di iterazioni minore di dieci (i valori delle iterazioni sono fissi nel sorgente);

il programma stampa a video i valori degli indici dei due cicli saltando a riga nuova per ogni iterazione del ciclo più esterno.[15]

```
1  /*
2  Programma:     stack2.s
3  Autore:        FF
4  Data:          gg/mm/aaaa
5  Descrizione:   Gestione di cicli annidati con lo stack
6  */
7  .bss
8  cifra:     .string ""         # per stampa delle cifre
9  .data
10 iteraz1:   .long   7          # iterazioni ciclo esterno
11 iteraz2:   .long   5          # iterazioni ciclo interno
12 spazio:    .string " "        # per spaziare le cifre
13 acapo:     .string "\n"       # a capo
14 .text
15 .globl _start
16 .macro stampa_spazio
17    movl $4, %eax
18    movl $1, %ebx
19    movl $spazio, %ecx
20    movl $1, %edx
21    int  $0x80
22 .endm
23 .macro stampa_cifra
24    movl $4, %eax
25    movl $1, %ebx
26    movl $cifra, %ecx
27    movl $1, %edx
28    int  $0x80
29 .endm
30 _start:
31 // imposta ciclo esterno
32    movl (iteraz1), %esi
33 ciclo_esterno:
34    movl %esi, %eax
35    addl $0x30, %eax              # somma 48 per avere ascii del valore
36    movl %eax, (cifra)
37 // stampa indice esterno
38    stampa_cifra
39 // stampa spazio due volte
40    stampa_spazio
41    stampa_spazio
42 // imposta ciclo interno
43    pushl %esi
44    movl (iteraz2), %esi
45  ciclo_interno:
46       movl %esi, %ebx
47       addl $0x30, %ebx           # somma 48 per avere ascii del valore
48       movl %ebx, (cifra)
49  // stampa indice interno
50       stampa_cifra
51  // stampa spazio
52       stampa_spazio
```

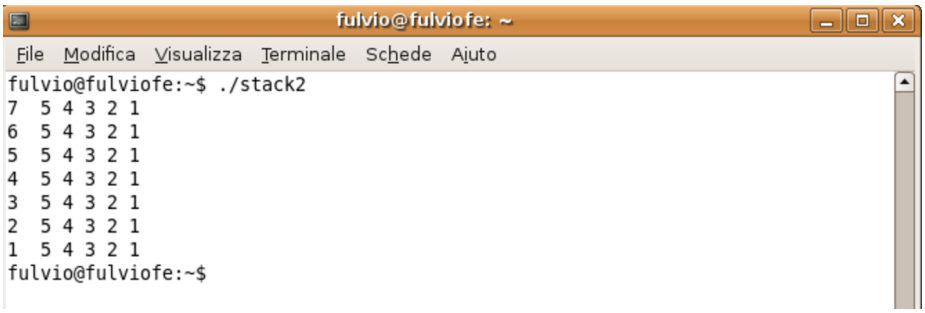
```
53      // controllo ciclo interno
54           decl %esi
55           jnz ciclo_interno
56      // fine ciclo interno: recupera indice esterno e stampa a capo
57         popl %esi
58         movl $4, %eax
59         movl $1, %ebx
60         movl $acapo, %ecx
61         movl $1, %edx
62         int  $0x80
63      // controllo ciclo esterno
64         decl %esi
65         jnz ciclo_esterno
66      fine:
67         movl $1, %eax
68         int  $0x80
```

Il programma usa lo stesso indice ***esi*** per gestire entrambe le iterazioni; le istruzioni più «interessanti» sono:

- alla riga 43, dove salva nella pila il valore dell'indice del ciclo esterno prima di reimpostarlo per il ciclo interno;
- alla riga 57, dove recupera dalla pila il valore dell'indice per proseguire le iterazioni del ciclo esterno.

Nella figura 3.57 vediamo gli effetti dell'esecuzione del programma.

Figura 3.57.



3.15 Uso di funzioni di linguaggio '`c`' nei programmi assembly

Quando si scrivono programmi in assembly in GNU/Linux, è possibile utilizzare in modo abbastanza comodo le funzioni del linguaggio '`c`' al loro interno.

Questa possibilità è molto allettante perché ci permette di usare le funzioni ***scanf*** e ***printf*** per l'input e l'output dei dati evitando tutti i problemi di conversione dei valori, da stringhe a numerici e viceversa, che si dovrebbero affrontare usando le *routine* di I/O native dell'assembly (a tale proposito si può consultare l'appendice C).

Il richiamo delle funzioni citate dai sorgenti assembly è abbastanza semplice e prevede le seguenti operazioni:

- inserimento nello stack dei parametri della funzione, uno alla volta in ordine inverso rispetto a come appaiono nella sintassi della stessa in linguaggio '`c`';
- richiamo della funzione con l'istruzione: ***call nome_funzione***;
- ripristino del valore del registro ***esp*** al valore precedente agli inserimenti nello stack.

Come esempio riportiamo una porzione di listato (non è un programma completo) con una chiamata a ***scanf*** e una a ***printf*** in linguaggio '`c`', commentate, seguite poi dalle sequenze di istruzioni assembly da eseguire per ottenere le stesse chiamate.

```
// scanf("%d",&val1); // input di una variabile intera di nome val1

pushl $val1      # val1 etichetta di tipo .long - $val1 e' il suo indirizzo
pushl $formato   # formato etichetta .string contenente "%d"
call scanf
addl $8, %esp    # due pushl hanno spostato %esp di 8 byte indietro

// printf("Valore del %d elem. = %f\n",el,ris); # stampa stringa con 2 val.

pushl (ris)      # ris etichetta di tipo .long
pushl (el)       # el etichetta di tipo .long
pushl $stri      # stri etichetta .string contente "Valore del %d elem. = %f\n"
call printf
addl $12, %esp   # tre pushl hanno spostato %esp di 12 byte indietro
```

Il prossimo listato è invece un programma completo, simile a quello visto in precedenza, in cui si chiedono due valori da tastiera, si esegue un calcolo (stavolta una somma) e si visualizza il risultato; grazie all'uso delle funzioni '`c`', non sono più necessarie le conversioni e il programma è molto più semplice, oltre che più breve.[16]

Si presti la massima attenzione al fatto che l'esecuzione delle funzioni ***printf*** e ***scanf*** avviene con l'uso, da parte del processore, dei registri accumulatori; essi sono quindi «sporcati» da tali esecuzioni e, nel caso contengano valori utili all'elaborazione, devono essere salvati in opportune etichette di appoggio.

```
/*
Programma:     io-funzc.s
Autore:        FF
Data:          gg/mm/aaaa
Descrizione:   Input e output con funzioni del c
*/
.bss
val1:      .long   0                        # primo valore di input
val2:      .long   0                        # secondo valore di input
ris:       .long   0                        # risultato
.data
invito:    .string "Inserire un valore: "   # stringa terminata con \0
formato:   .string "%d"                     # formato input
risult:    .string "Somma = %d\n"           # stringa per risultato
.text
```

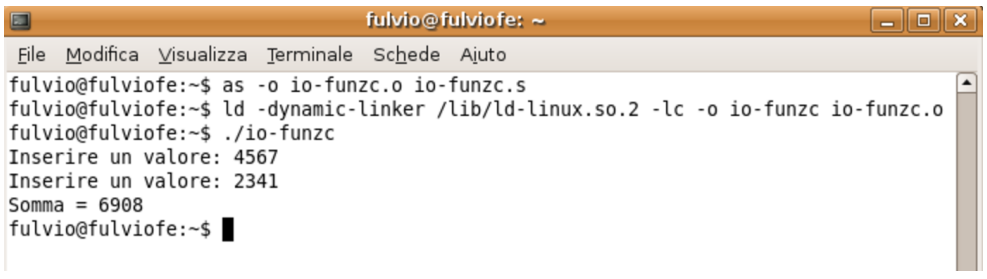
```
.globl _start
_start:
// input primo valore
   pushl $invito
   call printf
   addl $4, %esp
   pushl $val1
   pushl $formato
   call scanf
   addl $8, %esp
// input secondo valore
   pushl $invito
   call printf
   addl $4, %esp
   pushl $val2
   pushl $formato
   call scanf
   addl $8, %esp
// calcolo di val1 + val2
   movl (val2), %eax
   addl (val1), %eax
   movl %eax, (ris)
// stampa risultato
   pushl (ris)
   pushl $risult
   call printf
   addl $8, %esp
fine:
   movl $1, %eax
   int  $0x80
```

La fase di assemblaggio del programma non prevede cambiamenti; nella fase di linking invece devono essere aggiunte le seguenti opzioni:

- ***-lc***: significa che si devono collegare le librerie del '`c`';
- ***-dynamic-linker /lib/ld-linux.so.2***: serve ad usare il linker dinamico indicato per caricare dinamicamente le librerie del '`c`'.

Nella figura 3.60 vediamo i comandi per la traduzione e il linking e gli effetti dell'esecuzione del programma.

Figura 3.60.



Per ottenere più facilmente lo stesso risultato si può utilizzare lo script ***gcc*** per la traduzione del sorgente; in questo caso basta eseguire l'unico comando:

```
$ gcc -g -o nome_eseguibile nome_sorgente.s
```

a patto di avere sostituito nel sorgente le righe:

```
.globl _start
_start:
```

con:

```
.globl main
main:
```

Questa esigenza è dovuta al fatto che il '**gcc**' si aspetta di trovare l'etichetta ***main*** nel file oggetto di cui effettuare il link.

Ricordiamo che l'opzione '**-g**' nel comando, serve solo si ha intenzione di eseguire il programma con il debugger.

3.16 Rappresentazione dei valori reali

In questo paragrafo, servendoci di un semplice programma e del debugger, ci soffermiamo su alcune considerazioni riguardanti la rappresentazione dei valori numerici, soprattutto reali, all'interno del sistema di elaborazione, completando quanto mostrato nel paragrafo 3.5 e confermando le nozioni teoriche che il lettore dovrebbe possedere su questo argomento (e che sono comunque fruibili nelle dispense «Rappresentazione dei dati nell'elaboratore» segnalate all'inizio di questo testo).[17]

```
1  /*
2  Programma:     valori_num.s
3  Autore:        FF
4  Data:          gg/mm/aaaa
5  Descrizione:   Prove sulla rappr. di valori numerici reali
6  */
7  .data
8  val1:      .long     2147483643   # max intero - 4
9  val2:      .float    0.0625       # reale singola prec.
10 val3:      .double   -7.125       # reale doppia prec.
11 st_int:    .string   "Valore = %d\n"  # per stampa intero
12 st_real:   .string   "Valore = %f\n"  # per stampa reale
13 .text
14 .globl main
15 main:
16    nop
17    movl $val1, %eax
18    movl (val1), %ebx
19    movl (val2), %ecx
20    movl (val3), %edx
21 // stampe a video
22    pushl (val1)
23    pushl $st_int
24    call printf
```

```
25      addl $8, %esp
26      pushl (val3)+4
27      pushl (val3)
28      pushl $st_real
29      call printf
30      addl $12, %esp
31  fine:
32      movl $1, %eax
33      int  $0x80
```

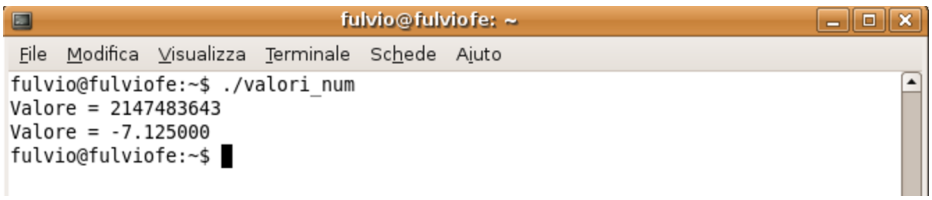
Il programma nelle prime righe, dalla 17, alla 20 esegue degli spostamenti che servono solo per visionare i dati con il debugger.

Successivamente, dalla riga 22 alla 25, stampa il valore intero e, dalla riga 26 alla 30 il valore reale in doppia precisione.

Soffermiamoci in particolare sulle righe 26 e 27 con le quali si pongono nello stack prima i 32 bit «alti» del valore ***var3*** e poi i 32 bit «bassi»; tale valore è infatti composto da 64 bit e quindi una sola istruzione ***pushl*** non sarebbe sufficiente.

Nella figura 3.64 vediamo il risultato dell'esecuzione del programma.

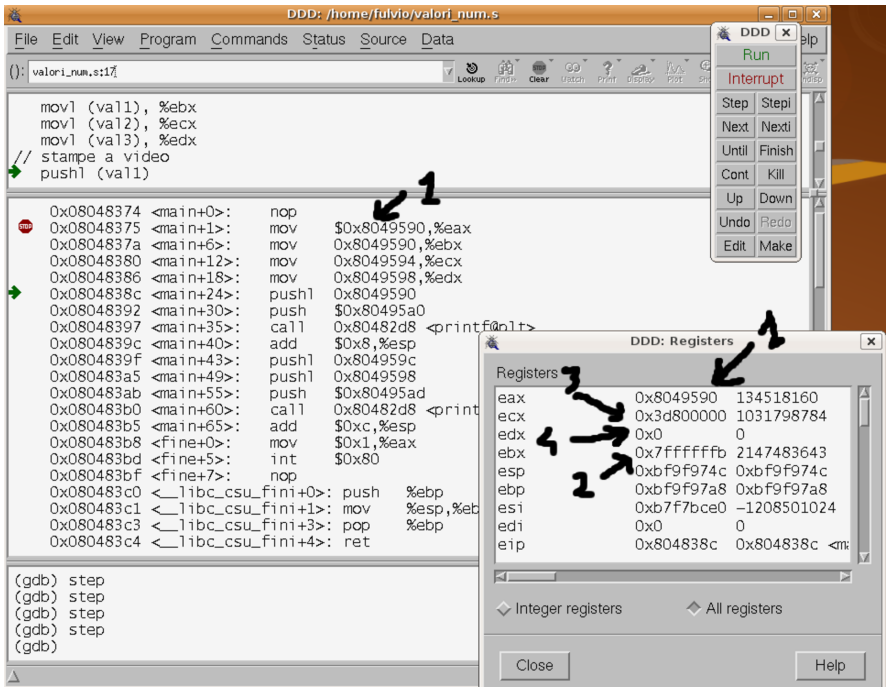
Figura 3.64.



Forse è però più interessante seguirne almeno i primi passaggi con il debugger.

Nella figura 3.65 vediamo la situazione dopo le istruzioni di spostamento.

Figura 3.65.



Il primo spostamento pone in ***eax*** l'indirizzo dell'etichetta ***val1*** ed in effetti possiamo constatarlo visionando il contenuto del registro e il valore dell'indirizzo nella finestra del codice macchina.

Il secondo spostamento porta in ***ebx*** il contenuto dell'etichetta ***val1*** che è $\mathbf{7fffffff_{16}}$ tradotto in esadecimale dal binario in complemento a due corrispondente al valore ***2,147,483,643***.

Il terzo spostamento porta in ***ecx*** il contenuto dell'etichetta ***val2*** che è $\mathbf{3d800000_{16}}$ tradotto in esadecimale dal binario in standard IEEE-754 singola precisione corrispondente al valore ***0.0625***.

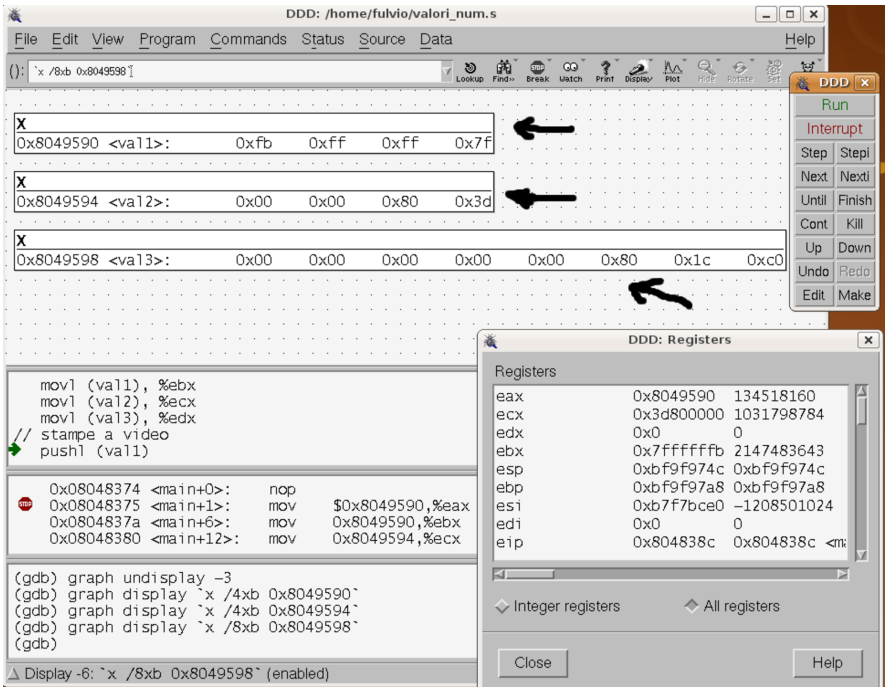
Il quarto spostamento, che dovrebbe avere portato in ***edx*** il contenuto dell'etichetta ***val3*** pare non sia riuscito; il motivo è che il valore è lungo 64 bit e il registro solo 32.

Se però andiamo a visionare direttamente gli indirizzi di memoria delle varie etichette, possiamo constatare che i valori sono tutti corretti.

Nella figura 3.66 vediamo appunto la presenza dei giusti valori assegnati alle tre etichette in esadecimale e memorizzati «al contrario» secondo il metodo '**`little-endian`**' di gestione della memoria.

In particolare il valore di ***val3*** è $\mathbf{c01c800000000000_{16}}$ cioè la traduzione in esadecimale della rappresentazione binaria in standard IEEE-754 doppia precisione di ***-7.125***.

Figura 3.66.



3.17 Modi di indirizzamento

Quando ci si riferisce a dei dati in memoria occorre specificare l'indirizzo dei dati stessi, indicandone solo l'offset (il registro di segmento o il selettore corrispondono infatti quasi sempre a ***ds***).

Ci sono varie maniere per fare questa operazione corrispondenti a vari '**`modi di indirizzamento`**':

- '**`diretto`**' o '**`assoluto`**': è il modo più comune e anche quello più utilizzato negli esempi finora esaminati; si indica direttamente in un operando il nome della cella di memoria interessata all'operazione;
- '**`indiretto`**': in tal caso l'indirizzo viene indicato in un registro ed esso deve essere racchiuso tra parantesi tonde; ad esempio: ***movl $1, (%ebx)***;
- '**`indicizzato`**' o '**`indexato`**': in questo caso l'offset si ottiene sommando ad un '**`indirizzo base`**' (etichetta o registro base) il valore di un '**`registro indice`**'; ad esempio: ***movl %eax, base(%esi)***;
- '**`base/scostamento`**': l'offset viene ottenuto sommando a un indirizzo base una costante; ad esempio: ***pushl (val1)+4*** oppure ***8(%ebp)***;
- '**`base/indice/scostamento`**': è una generalizzazione dei precedenti e lo illustriamo più avanti con un esempio;
- '**`immediato`**': lo inseriamo in questo elenco ma non è un vero modo di indirizzamento in quanto indica l'uso di un operando che contiene un valore, detto appunto immediato; esempio ***movl $4, %eax***.

Occorre subito chiarire che con il processore 8086 ci sono delle limitazioni nell'uso dei registri per l'indirizzamento indiretto: possono essere solo ***bx***, ***si***, ***di*** relativamente al segmento ***ds*** e ***bp*** per il segmento ***ss***.

Anche per l'indirizzamento con gli indici ci sono regole abbastanza rigide: i registri base possono essere solo ***bx*** e ***bp*** mentre gli indici sono solo ***si*** e ***di***.

Tutte queste limitazioni non esistono invece nei processori IA-32, con i quali si possono usare indistintamente tutti i registri estesi a 32 bit.

Torniamo ora brevemente sull'indirizzamento '**`base/indice/scostamento`**' o '**`base + indice * scala + scostamento`**' ('**`base + index * scale + disp`**'):

```
movw var(%ebx,%eax,4), %cx
```

Significa che vogliamo spostare in ***cx*** il contenuto della cella di memoria il cui indirizzo è ottenuto sommando quello di ***var*** (base), al contenuto di ***ebx*** (scostamento) e al prodotto fra ***4*** (scala) e il contenuto di ***eax*** (indice); nei nostri esempi una modalità così complessa di indirizzamento non è necessaria.

3.18 Vettori

Come noto un vettore o *array* è una struttura dati costituita da un insieme di elementi omogenei che occupano locazioni di memoria consecutive.

La dichiarazione di un vettore è molto semplice; sotto sono mostrati due esempi:

```
vet1:   .byte 1,2,5,3,9,6,7
vet2:   .fill,50,1,0
```

Nel primo caso si definisce un vettore in cui ogni elemento è un byte (ma è ovviamente possibile usare ***.word***, ***.long*** ecc.) e le cui celle contengono i valori elencati a fianco; nel secondo caso abbiamo un vettore di ***50*** elementi grandi un byte, tutti contenenti il valore zero.

Notiamo che le stringhe possono essere proficuamente gestite come vettori di caratteri (come avviene in alcuni linguaggi, fra i quali il '**`C`**'); vediamo un paio di definizioni «alternative» di stringhe:

```
stringa1: .byte 'C','i','a','o',' ','a',' ','t','u','t','t','i'
stringa2: .fill,50,1,'*'
```

Sicuramente nel primo caso è molto più comodo usare la direttiva ***.string***; quando invece si deve definire una stringa contenente ripetizioni di uno stesso carattere, come nel secondo caso, è più conveniente usare ***.fill***.

Come esempio di uso di un vettore vediamo un programma che individua e stampa a video (usando la chiamata ***printf***) il valore massimo contenuto in un vettore; tale vettore è predefinito all'interno del programma.

Il sorgente contiene già tutti i commenti che dovrebbero permettere la comprensione della sua logica elaborativa.[18]

```
/*
Programma:      vettore.s
Autore:         FF
Data:           gg/mm/aaaa
Descrizione:    Ricerca max in un vettore predefinito
*/
.data
```

```
vet:       .byte 12,34,6,4,87,3
msgout:    .string "Max = %d\n"
.text
.globl main
main:
   movl $5, %ebx        # ultimo indice vettore
   xorl %eax, %eax      # pulisco %eax (max)
   movl $0, %esi
   movb vet(%esi), %al  # inizialmente metto in max il primo el del vettore
   movl $1, %esi        # imposto indice per il ciclo
ciclo:
   cmpb vet(%esi), %al  # confronto el del vettore con max
   jg avanti            # se max è maggiore non faccio niente
   movb vet(%esi), %al  # se max minore aggiorno max
avanti:
   incl %esi            # incremento indice
   cmpl %esi, %ebx       # controllo di fine ciclo
   jne ciclo
// stampa max trovato
   pushl %eax           # stampo con funzione printf
   pushl $msgout
   call printf
   addl $8, %esp
fine:
   movl $1, %eax
   int  $0x80
```

3.19 Procedure

Le procedure sono i sottoprogrammi del linguaggio assembly.

Nella sintassi AT&T si dichiarano semplicemente con un nome di etichetta e si chiudono con l'istruzione ***ret***; il richiamo avviene con l'istruzione ***call*** seguita dal nome dell'etichetta.

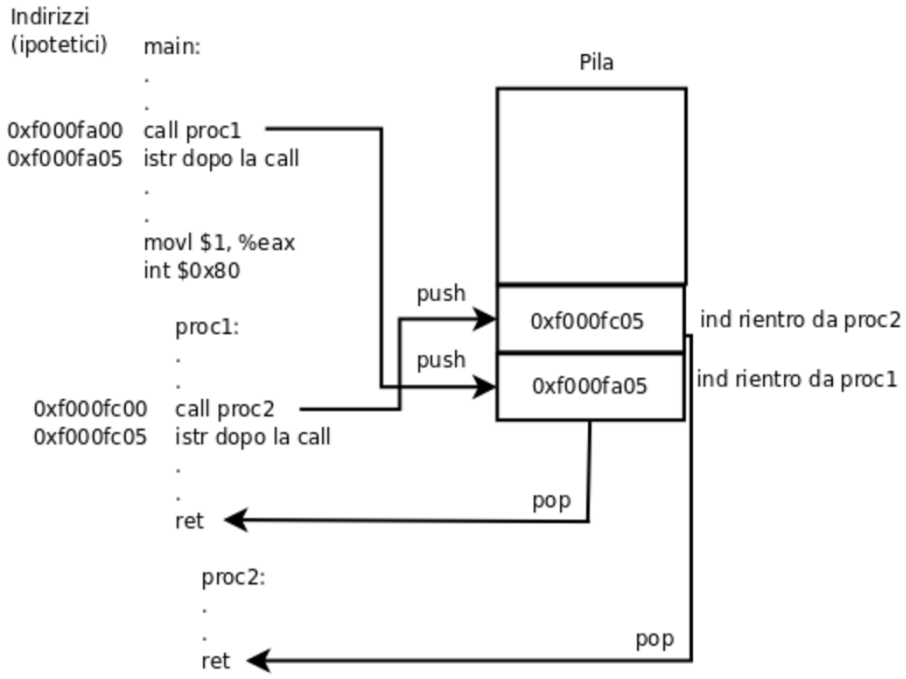
La gestione dell'esecuzione di una procedura avviene, da parte del sistema, mediante queste operazioni:

- viene salvato l'indirizzo dell'istruzione successiva alla ***call*** che in quel momento è contenuto in ***eip***; tale indirizzo è detto '**`indirizzo di rientro`**';
- viene effettuato un salto all'indirizzo corrispondente al nome della procedura;
- quando essa termina (istruzione ***ret***) viene recuperato dallo stack l'indirizzo di rientro e memorizzato in ***eip***, in modo che l'esecuzione riprenda il suo flusso originario nel programma chiamante.

Il meccanismo permette di gestire chiamate nidificate e anche la ricorsione (chiamata a se stessa da parte di una procedura), sfruttando la modalità di accesso allo stack in modo da «impilare» i relativi indirizzi di rientro.

Lo schema della figura 3.71 rappresenta quanto appena detto.

Figura 3.71.



Ribadiamo che tutte queste operazioni vengono svolte automaticamente dal sistema senza che il programmatore debba preoccuparsene.

Se però di devono passare dei parametri ad una procedura, l'automatismo non è più sufficiente e occorre gestire tale passaggio usando opportunamente lo stack, senza interferire con l'uso che ne fa il sistema per la chiamata alla procedura e il successivo rientro al chiamante.

La sequenza delle operazione da compiere è:

- depositare nello stack, prima della chiamata alla procedura, i valori dei parametri da passarle, eventualmente anche quelli di ritorno, che essa provvederà a valorizzare e aggiornare nello stack;
- all'interno del sottoprogramma provvedere poi ad estrarre, usare e eventualmente reinserire nello stack, tali valori usando il registro ***ebp*** e non l'istruzione ***pop*** in quanto quest'ultima estrarrebbe i valori a partire dall'ultimo inserito che non è uno dei «nostri» parametri, ma l'indirizzo di rientro inserito automaticamente dal sistema;
- nel programma chiamante, al rientro dalla procedura, effettuare le opportune estrazioni dallo stack per «ripulirlo» e/o per usare eventuali parametri di ritorno dal sottoprogramma.

Altre operazioni che potrebbero rivelarsi necessarie sono:

- il salvataggio nello stack del registro ***ebp***, da fare all'inizio della procedura, in modo che il suo uso non interferisca con quello fatto da altre procedure nel caso di chiamate nidificate; il vecchio valore del registro deve essere poi ripristinato, estraendolo dallo stack, prima dell'istruzione ***ret***;
- il salvataggio nello stack, fatto all'inizio della procedura, dei registri usati dal chiamante, nell'eventualità che essi siano usati anche all'interno del sottoprogramma, e il relativo loro recupero prima della conclusione dello stesso; in questo caso si possono proficuamente usare le istruzioni ***pusha*** e ***popa***.

Possiamo riassumere tutto il procedimento in linguaggio informale nel modo seguente:

```
nel chiamante:

- salvataggio nello stack dei parametri da passare alla procedura
- call della procedura
- (inserimento nello stack dell'indirizzo di rientro) fatto
  automaticamente dal sistema

nella procedura:

- eventuale salvataggio nello stack del registro ebp
- valorizzazione di ebp = esp
- eventuale salvataggio nello stack dei registri usati dal
  chiamante o di tutti i registri (pusha)
- uso di ebp con indirizzamento base/scostamento per leggere i param. dallo stack
- eventuale aggiornamento nello stack dei parametri di ritorno
- eventuale ripristino dei registri precedentemente salvati o di tutti i
  registri (popa)
- eventuale ripristino del valore precedente di ebp prelevandolo dallo stack
- ret
- (estrazione dell'indirizzo di rientro e valorizzazione di eip) fatta
  automaticamente dal sistema

nel chiamante:

- estrazione dallo stack dei parametri
```

Come primo esempio di uso di una procedura consideriamo un programma molto semplice che è suddiviso in un *main* che accetta due valori in input (usando la chiamata ***scanf***), richiama una procedura di calcolo e stampa a video il risultato (con la chiamata ***printf***); il calcolo consiste nella somma tra i due valori ricevuti come parametri dal sottoprogramma insieme al risultato (inizialmente pari a zero), che sarà il valore di ritorno.[19]

```
/*
Descrizione:   Chiamata a una procedura con parametri
Autore:        FF
Data:          gg/mm/aaaa
*/
.bss
val1:       .long 0
val2:       .long 0
ris:        .long 0
.data
msgin:      .string "Inserire un valore: "
formatin:   .string "%d"
msgout:     .string "Somma = %d\n"
.text
.globl main
main:
// fase di input
   pushl $msgin
   call printf
   addl $4, %esp
```

```
   pushl $val1
   pushl $formatin
   call scanf
   addl $8, %esp
   pushl $msgin
   call printf
   addl $4, %esp
   pushl $val2
   pushl $formatin
   call scanf
   addl $8, %esp
// chiamata a procedura
   pushl (ris)      # in stack il risultato (valore di ritorno)
   pushl (val2)     # in stack primo parametro
   pushl (val1)     # in stack secondo parametro
   call somma
   popl  %eax       # estraggo secondo parametro (non serve piu')
   popl  %eax       # estraggo primo parametro (non serve piu')
   popl  (ris)      # estraggo risultato
// stampa risultato
   pushl (ris)
   pushl $msgout
   call printf
   addl $8, %esp
fine:
   movl $1, %eax
   int  $0x80
// procedura
somma:
   movl %esp, %ebp      # %ebp = cima della pila
   movl 4(%ebp), %ebx   # salto i primi 4 byte dove e' ind. di rientro
                        # e leggo il primo parametro
   movl 8(%ebp), %eax   # leggo secondo parametro
   addl %ebx, %eax
   movl %eax, 12(%ebp)  # scrivo il risultato sul terzo parametro
   ret
```

Come ulteriore esempio vediamo invece un programma un po' più impegnativo, soprattutto per la gestione dello stack che impone: il calcolo del fattoriale di un numero naturale con uso di funzione ricorsiva.

Anche in questo caso c'è un programma principale che si cura dell'input e della visualizzazione del risultato e che richiama la procedura di calcolo; come nel precedente esempio, non è inserita la numerazione per la successiva illustrazione delle istruzioni, in quanto ci sono abbondanti commenti che dovrebbero essere sufficienti per la comprensione del programma.[20]

```
/*
Descrizione:   Fattoriale con procedura ricorsiva
Autore:        FF
Data:          gg/mm/aaaa
*/
.bss
num:       .long 0
ris:       .long 0
```

```
.data
msgin:      .string "Inserire il valore: "
formatin:   .string "%d"
msgout:     .string "Fattoriale = %d\n"
.text
.globl main
main:
// fase di input
   pushl $msgin
   call printf
   addl $4, %esp
   pushl $num
   pushl $formatin
   call scanf
   addl $8, %esp
// chiamata a procedura
   pushl (ris)      # in stack il risultato (valore di ritorno)
   pushl (num)      # in stack il parametro
   call fatt        # eseguo fatt(num)
   popl  %eax       # estraggo parametro (non serve piu')
   popl  (ris)      # estraggo risultato
// stampa risultato
   pushl (ris)
   pushl $msgout
   call printf
   addl $8, %esp
fine:
   movl $1, %eax
   int  $0x80
// procedura fatt(x)
fatt:
   pushl %ebp           # salvo %ebp
   movl %esp, %ebp      # %ebp = cima della pila
   movl 8(%ebp), %ebx   # salto i primi 8 byte dove ci sono %ebp e
                        # ind. di rientro e leggo il parametro x
   cmpl $1, %ebx        # se x = 1 esco dalla ricorsione (fatt = 1)
   je noricor
   pushl %ebx           # metto da parte questo parametro x
   decl %ebx            # x = x-1
   pushl (ris)          # in stack il risultato (valore di ritorno)
   pushl %ebx           # in stack il parametro (x-1)
   call fatt            # eseguo fatt(x)
   popl  %eax           # estraggo parametro (non serve piu')
   popl  %eax           # estraggo risultato ( fatt(x-1) )
   popl  %ebx           # estraggo %ebx precedente (x)
   mull  %ebx           # calcolo %eax = x*fatt(x-1)
   movl  %eax, 12(%ebp) # scrivo risultato
   jmp fine_proc
noricor:
   movl %ebx, 12(%ebp)  # scrivo il risultato senza ricorsione %ebx = 1
fine_proc:
   popl %ebp            # ripristino %ebp
   ret
```

[1] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/modello.s*⟩.

[2] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/01som.s*⟩.

[3] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/02ope.s*⟩.

[4] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/ope_logiche.s*⟩.

[5] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/03ope.s*⟩.

[6] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/ope_logiche_mem.s*⟩.

[7] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/04ope.s*⟩.

[8] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/salto_inc.s*⟩.

[9] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/selezione1.s*⟩.

[10] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/selezione2.s*⟩.

[11] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/iterazione1.s*⟩.

[12] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/iterazione2.s*⟩.

[13] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/iterazione3.s*⟩.

[14] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/stack1.s*⟩.

[15] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/stack2.s*⟩.

[16] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/io-funzc.s*⟩.

[17] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/valori_num.s*⟩.

[18] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/vettore.s*⟩.

[19] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/proc1.s*⟩.

[20] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/proc2.s*⟩.

Appendici

Appendice A

Istruzioni dichiarative, direttive ed esecutive dell'assembly AT&T

A.1 Istruzioni dichiarative

Le istruzioni dichiarative servono a dichiarare le etichette dati e le etichette istruzioni nei programmi assembly.

Il ruolo e l'importanza di tali etichette è già stato illustrato nei vari esempi di queste dispense; ricordiamo solo che i nomi delle etichette sono «liberi» ma non devono coincidere con parole riservate del linguaggio, devono essere il più possibile significativi e devono essere seguiti dal simbolo «:».

A.2 Istruzioni direttive

Riguardo alle direttive all'assemblatore vediamo un piccolo elenco delle più usate, con una breve descrizione del loro ruolo:

- ***.ascii*** o ***.asciz*** o ***.string***: queste direttive permettono di dichiarare una o più stringhe separate da virgole; con ***.ascii*** le stringhe non sono terminate, con le altre le stringhe sono terminate con l'apposito carattere '\0'; esempio: ***messaggio: .string "Ciao\n"***;
- ***.byte***: permette di dichiarare un dato lungo un byte; esempio: ***carattere: .byte 'a'***;
- ***.word***: permette di dichiarare un dato lungo due byte; esempio: ***val: .word 1234***;
- ***.int*** o ***.long***: permette di dichiarare un dato lungo quattro byte; esempio: ***val: .long 1000000***;
- ***.float*** e ***.double***: permettono di dichiarare un dato reale macchina in singola precisione (lungo quattro byte) e in doppia precisione (lungo otto byte); a tale proposito occorre osservare che i dati reali vengono gestiti dal '`coprocessore matematico`' mediante otto registri interni lunghi 80 bit (viene usata la precisione estesa) denominati ***st0***, ***st1***, .. ***st7*** (in queste dispense non vengono forniti ulteriori approfondimenti sul funzionamento del coprocessore matematico e sulla gestione di questi registri);
- ***.data***: definisce l'inizio del segmento dati; segnaliamo che, a differenza di quanto avviene con altri assembly, i registri di segmento vengono caricati automaticamente e il programmatore non deve preoccuparsi di tale aspetto;
- ***.bss***: (*Block Starded by Symbol*) è il segmento contenente i dati non inizializzati, o meglio che vengono allocati in memoria al momento dell'esecuzione e non al momento della traduzione del programma;
- ***.text***: definisce l'inizio del segmento istruzioni;
- ***.extern***: è una direttiva disponibile ma ignorata; tutti i simboli non definiti sono considerati automaticamente esterni, cioè definiti in un altro modulo sorgente (che dovrà essere uniti a quello corrente nella fase di *linking*);
- ***.global*** o ***.globl***: definiscono i simboli globali (esportati ad altri moduli); ognuna prende per argomento il simbolo da rendere globale; esempio: ***.globl _start***;
- ***.include***: include un file nella posizione corrente; prende come argomento il nome del file tra virgolette; esempio: ***.include "stdio.inc"***;

- ***.rept*** e ***.endr***: permette di ripetere le dichiarazioni inserite fino alla direttiva ***.endr*** il numero di volte indicato come argomento di ***.rept***; esempio:

```
v:
.rept 100
.long 0
.endr
```

dichiara una etichetta di nome ***v*** costituita da 100 elementi di tipo ***.long*** ognuno con valore zero.

A.3 Istruzioni esecutive

Suddividiamo le istruzioni esecutive per tipologia:

- *istruzioni di uso generale*:
  - ***int***: attivazione di un interrupt software; si usa per richiamare delle *routine* del sistema operativo e ha come operando il valore numerico che identifica l'interrupt;
  - ***mov***: effettua lo spostamento di valore tra due operandi;
  - ***nop***: nessuna operazione, non ha operandi;
  - ***xchg***: scambia il valore tra i due operandi;
- *istruzioni aritmetiche*:
  - ***add***: somma il primo operando al secondo, il risultato sostituisce il secondo operando;
  - ***adc***: somma con carry, cioè tenendo conto del riporto dell'operazione precedente;
  - ***inc***: incremento di una unità dell'operando;
  - ***sub***: sottrae il primo operando al secondo, il risultato sostituisce il secondo operando;
  - ***sbb***: sottrazione con borrow, cioè tenendo conto del prestito dell'operazione precedente;
  - ***dec***: decremento di una unità dell'operando;
  - ***neg***: negazione dell'operando (complemento a due);
  - ***cmp***: confronto tra due operandi; è considerata istruzione aritmetica perché il confronto viene fatto con una sottrazione tra i due operandi e verificando se il risultato è zero;
  - ***mul***: moltiplicazione tra valori naturali; prevede un solo operando;
  - ***imul***: moltiplicazione tra interi; vale quanto detto per la ***mul***;
  - ***div***: divisione tra valori naturali; vale quanto detto per la ***mul***;
  - ***idiv***: divisione tra interi; vale quanto detto per la ***mul***;
- *istruzioni logiche*:
  - ***not***: operazione di negazione; fa il complemento a uno dei bit dell'operando;
  - ***and***: effettua l'operazione di «and» tra i bit corrispondenti dei due operandi;
  - ***or***: effettua l'operazione di «or» tra i bit corrispondenti dei due operandi;
  - ***xor***: effettua l'operazione di «or esclusivo» tra i bit corrispondenti dei due operandi;

- *istruzioni per gestire i bit del registro di stato*:
  - ***cli***: azzera il bit if per mascherare le interruzioni;
  - ***sti***: setta il bit if per abilitare le interruzioni;
  - ***clc***: azzera il flag ***cf***;
  - ***stc***: setta il flag ***cf***;
- *istruzioni di I/O*:
  - ***in***: per leggere dati da una porta di I/O; ha due operandi, la porta e il registro;
  - ***out***: per scrivere dati su una porta di I/O; ha due operandi, il registro e la porta;
- *istruzioni di salto*; hanno tutte un operando che è l'etichetta istruzioni a cui saltare e devono seguire immediatamente (a parte quella di salto incondizionato) una istruzione il cui effetto sui flag del registro di stato condiziona l'avvenire o meno del salto:
  - ***jmp***: salto incondizionato;
  - ***ja***: salta se maggiore per confronti tra numeri naturali;
  - ***jae***: salta se maggiore o uguale per confronti tra numeri naturali;
  - ***jb***: salta se minore per confronti tra numeri naturali;
  - ***jbe***: salta se minore o uguale per confronti tra numeri naturali;
  - ***jc***: salta se il bit ***cf*** è settato;
  - ***je***: salta se valori uguali;
  - ***jg***: salta se maggiore per confronti tra numeri interi;
  - ***jge***: salta se maggiore o uguale per confronti tra numeri interi;
  - ***jl***: salta se minore per confronti tra numeri interi;
  - ***jle***: salta se minore o uguale per confronti tra numeri interi;
  - ***jnc***: salta se il flag ***cf*** non è settato;
  - ***jne***: salta se valori diversi;
  - ***jno***: salta se non è avvenuto overflow;
  - ***jns***: salta se il risultato è positivo;
  - ***jnz***: salta se il risultato non è zero;
  - ***jo***: salta se è avvenuto overflow;
  - ***js***: salta se il risultato è negativo;
  - ***jz***: salta se il risultato è zero;
  - ***loop***: esegue un ciclo (vedere paragrafo 3.13.3);
- *istruzioni per l'uso delle procedure*:
  - ***call***: chiamata a una procedura di cui va indicato il nome;
  - ***ret***: ritorno da una procedura; non ha operandi;
- *istruzioni per la gestione dello stack*:
  - ***pop***: per prelevare un dato dallo stack e porlo nel registro usato come operando;
  - ***popa***: per prelevare dallo stack i valori di tutti i registri (non ha operandi);
  - ***push***: per inserire un dato nello stack prelevandolo dal registro usato come operando.

– ***pusha***: per inserire nello stack i valori di tutti i registri (non ha operandi).

Si ricordi che tutte le istruzioni che coinvolgono registri o operandi di memoria si differenziano grazio ad un opportuno suffisso (***b***, ***w***, ***l***), il cui uso non è obbligatorio ma «fortemente consigliato».

Appendice B

Confronto tra sintassi AT&T e sintassi Intel

Vediamo un confronto sommario tra la sintassi AT&T e la sintassi Intel che è maggiormente usata essendo adottata da un numero maggiore di tipi di assembly.

Fra questi citiamo:

- MASM (*Macro Assembler*), prodotto proprietario della Microsoft e disponibile per le piattaforme DOS/Windows;
- TASM (*Turbo Assembler*), prodotto proprietario della Borland e disponibile per le piattaforme DOS/Windows;
- NASM (*Netwide Assembler*), prodotto libero e gratuito e disponibile sia per DOS/Windows che per Unix/Linux;
- AS86, prodotto libero e gratuito e disponibile per Unix/Linux.

Segnaliamo che non tutti gli assembly sono *case sensitive* come GAS; alcuni lo sono (NASM), altri no (MASM).

Notiamo inoltre che, nel caso della sintassi Intel, solo le etichette istruzioni sono seguite da «:» mentre in GAS tale simbolo segue qualunque etichetta.

Vediamo adesso le altre differenze più rilevanti tra i due tipi di sintassi riferendosi, per quella Intel, all'assembly NASM.

Come abbiamo visto, nella sintassi AT&T i nomi dei registri sono preceduti dal simbolo «%», mentre i valori immediati sono preceduti dal simbolo «$»; questo non avviene invece nella sintassi Intel.

Sappiamo che il simbolo «$» nella sintassi AT&T può precedere anche un nome di etichetta e in quel caso serve a riferirsi al suo indirizzo in memoria; per lo stesso scopo, nella sintassi Intel, basta usare il nome dell'etichetta.

Con la sintassi Intel i valori esadecimali si indicano inserendo il suffisso «h» e il prefisso «0» se iniziano con una lettera, oppure utilizzando il prefisso «0x» (come nella sintassi AT&T).

Nella sintassi Intel non ci sono i suffissi delle istruzioni; allo stesso scopo, esistono le direttive ***byte ptr***, ***word ptr***, ***dword ptr***, ma sono poco usate in quanto è la dimensione degli operandi a stabilire la natura dell'istruzione usata.

Ribadiamo che anche anche GAS si comporta in modo simile permettendo l'omissione del suffisso che poi viene assegnato in automatico dall'assemblatore in base alla natura degli operandi; ciò però è possibile solo se nell'operazione non sono coinvolte locazioni di memoria ed inoltre è sempre consigliabile usare i suffissi per aumentare la leggibilità dei sorgenti.

Per riassumere le differenze appena illustrate usiamo la tabella B.1 in cui sono inserite istruzioni equivalenti, scritte secondo le regole di sintassi delle due tipologie di assembly.

Tabella B.1

Intel	AT&T
mov eax, 5	movl $5, %eax
mov ecx, 0ffh	movl $0xff, %ecx
int 80h	int $0x80
mov cl, dh	movb %dh, %cl
mov ax, bx	movw %bx, %ax
mov ebx, dword ptr [ecx]	movl (%ecx), %ebx

Come si evince dagli esempi della tabella precedente, il ruolo degli operandi è opposto nei due casi: si scrive prima la destinazione nella sintassi Intel e prima la sorgente nella sintassi AT&T.

Nella sintassi Intel i commenti si inseriscono in una sola maniera: facendoli precedere dal carattere «;».

Quando si fa riferimento a locazioni di memoria, nella sintassi Intel si usano le parantesi quadre invece che quelle tonde della sintassi AT&T.

Nella tabella B.2 vediamo alcuni esempi a tale proposito.

Tabella B.2

Intel	AT&T
mov eax, [ebx] ; commento	movl (%ebx), %eax # commento
mov ecx, dato	movl $dato, %ecx
mov eax, [ebx+3]	movl 3(%ebx), %eax

Nel primo caso si muove nel registro ***eax*** il contenuto della cella puntata da ***ebx***; nel secondo caso si muove in ***ecx*** l'indirizzo dell'etichetta ***dato***; il terzo è un caso particolare dell'indirizzamento '**`base + index * scale + disp`**' ('**`base + indice * scala + scostamento`**') di cui daremmo qualche cenno nel paragrafo 3.17: si sposta in ***eax*** il contenuto della cella il cui indirizzo è dato dal valore di ***ebx*** più tre.

Appendice C

Le operazioni di I/O con l'assembly AT&T

L'ingresso e uscita di dati nel linguaggio assembly riguardano esclusivamente stringhe; in caso si debbano inserire o visualizzare valori numerici occorre operare le necessarie conversioni da programma (più avanti verranno mostrati esempi a tale proposito).

Per gestire l'input e l'output si deve ricorrere a delle funzioni del sistema operativo, dette «chiamate di sistema» o *syscall*.

Per la lettura e la scrittura si usano rispettivamente le chiamate di sistema corrispondenti alle funzioni '**`read`**' e '**`write`**' del linguaggio '**`c`**' (si ricordi che il sistema operativo GNU/Linux è scritto in '**`c`**', quindi le chiamate di sistema hanno una stretta corrispondenza con le funzioni a basso livello di tale linguaggio).

Per avere maggiori dettagli circa queste due funzioni si può usare il manuale in linea di GNU/Linux, ad esempio con il comando:

```
$ man read
```

Un esempio di chiamata di sistema molto usata (forse la più usata) è quella per l'uscita dal programma, attivata ponendo il valore ***1*** nel registro ***eax*** prima del richiamo dell'interruzione software 80_{16} (istruzione: ***int $0x80***).

Per attivare la chiamata di sistema per la lettura occorre ovviamente richiamare la stessa interruzione software, valorizzando però i registri in modo più articolato:

- ***eax***: deve contenere il valore ***3*** corrispondente appunto alla funzione di lettura;
- ***ebx***: deve contenere il numero identificativo del file da cui leggere; nel nostro caso si tratta del file speciale '**`stdin`**' corrispondente alla tastiera e associato al valore ***0***;
- ***ecx***: deve contenere l'indirizzo di partenza della stringa che accoglierà l'input;
- ***edx***: deve contenere il numero di caratteri da leggere.

Dopo l'esecuzione della chiamata, nel registro ***eax***, viene ritornata la lunghezza effettiva della stringa letta, comprensiva del carattere «invio» (***'\n'***) con cui si conclude la sua immissione.

Per la scrittura a video la situazione è simile e si devono usare:

- ***eax***: deve contenere il valore ***4*** corrispondente alla funzione di scrittura;
- ***ebx***: deve contenere il numero identificativo del file su cui scrivere; nel nostro caso si tratta del file speciale '**`stdout`**' corrispondente al video e associato al valore ***1***;
- ***ecx***: deve contenere l'indirizzo di partenza della stringa da visualizzare;
- ***edx***: deve contenere il numero di caratteri da scrivere.

Dopo l'esecuzione della chiamata, nel registro ***eax***, viene ritornata la lunghezza effettiva della stringa visualizzata.

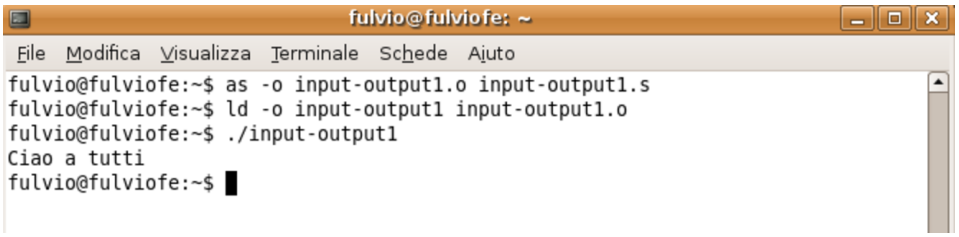
Come primo esempio vediamo un programma che stampa a video il messaggio «Ciao a tutti».[1]

```
/*
Programma:      input-output1.s
Autore:         FF
Data:           gg/mm/aaaa
Descrizione:    Esempio di scrittura a video
```

```
*/
.data
messaggio:   .string "Ciao a tutti\n" # \n per andare a capo
.text
.globl _start
_start:
   movl $4, %eax
   movl $1, %ebx
   movl $messaggio, %ecx   # $messaggio contiene ind. di inizio della stringa
   movl $13, %edx
   int  $0x80
fine:
   movl $1, %eax
   int  $0x80
```

Essendo questo un programma che prevede una qualche interazione durante la sua esecuzione, nella figura C.2 possiamo vedere, dopo i comandi per la sua traduzione e linking, l'effetto della sua esecuzione con la stampa a video della stringa.

Figura C.2.



Nel prossimo esempio vogliamo invece stampare a video una stringa immessa da tastiera.[2]

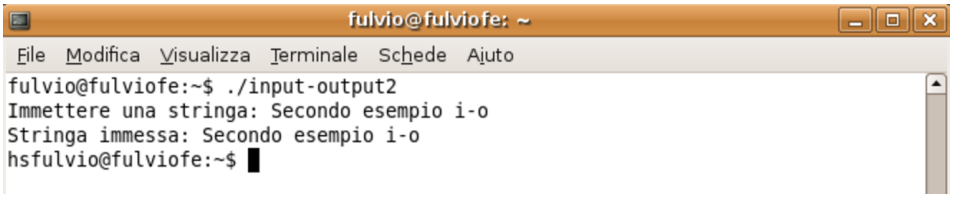
```
/*
Programma:     input-output2.s
Autore:        FF
Data:          gg/mm/aaaa
Descrizione:   Esempio di lettura da tastiera e scrittura a video
*/
.bss
stringa:   .string ""                          # stringa di output
.data
invito:    .string "Immettere una stringa: "   # senza a capo
msgout:    .string "Stringa immessa: "         # senza a capo
app:       .long   0                           # appoggio per lunghezza effettiva
.text
.globl _start
_start:
// messaggio di invito
   movl $4, %eax
   movl $1, %ebx
   movl $invito, %ecx
   movl $23, %edx
   int  $0x80
// lettura stringa da tastiera
   movl $3, %eax
```

```
    movl $0, %ebx
    movl $stringa, %ecx
    movl $100, %edx     # 100 = lunghezza massima stringa
    int  $0x80
    movl %eax, (app)    # lunghezza effettiva in app per la successiva stampa
// stampa a video messaggio preliminare
    movl $4, %eax
    movl $1, %ebx
    movl $msgout, %ecx
    movl $18, %edx
    int  $0x80
// stampa a video stringa
    movl $4, %eax
    movl $1, %ebx
    movl $stringa, %ecx
    movl (app), %edx
    int  $0x80
fine:
    movl $1, %eax
    int  $0x80
```

Notiamo l'utilizzo del segmento ***bss***, nel quale viene dichiarata l'etichetta da usare per l'output.

Nella figura C.4 vediamo il risultato dell'esecuzione del programma.

Figura C.4.



Come detto, in assembly l'input e l'output riguardano solo stringhe: se inseriamo un valore numerico questo viene acquisito come una stringa composta dai simboli corrispondenti alle cifre da cui è costituito; viceversa se vogliamo visualizzare un valore dobbiamo trasformarlo nella corrispondente stringa.

Nel prossimo esempio vengono mostrati entrambi questi procedimenti in un programma che accetta da tastiera due valori interi, ne calcola il prodotto ed emette a video il risultato.

In questo caso ripristiniamo la numerazione delle righe del listato perché i procedimenti di conversione non sono banali e meritano qualche spiegazione più accurata.[3]

```
1   /*
2   Programma:      input-output3.s
3   Autore:         FF
4   Data:           gg/mm/aaaa
5   Descrizione:    Acquisizione e stampa a video di valori numerici
6   */
7   .bss
8   stringa:    .string ""                          # stringa per input
9   .data
10  invito:     .string "Immettere un valore: "     # senza a capo
```

```
11  msgout:    .string "Risultato del prodotto: " # senza a capo
12  lun:       .long   0                          # lunghezza input
13  val1:      .byte   0                          # primo valore di input
14  val2:      .byte   0                          # secondo valore di input
15  strout:    .string "     \n"                  # stringa per output (5 car)
16  .text
17  .globl _start
18  _start:
19  // input primo valore
20  // messaggio di invito
21     movl $4, %eax
22     movl $1, %ebx
23     movl $invito, %ecx
24     movl $21, %edx
25     int  $0x80
26  // lettura stringa da tastiera
27     movl $3, %eax
28     movl $0, %ebx
29     movl $stringa, %ecx
30     movl $4, %edx       # 4 = lun. max stringa (3 + 1 per invio)
31     int  $0x80
32     movl %eax, (lun)    # lunghezza effettiva immessa
33  // converti primo input
34     mov $0, %esi
35     xorw %ax, %ax
36     movw $0xa, %bx
37     movl (lun), %ecx
38     decl %ecx            # per non contare anche invio
39  ciclo1:
40     mulw %bx             # esegue %ax = %ax * %bx ( = 10)
41     movb stringa(%esi), %dl
42     subb $0x30, %dl
43     addb %dl, %al
44     adcb $0, %ah
45     inc  %esi
46     loopl ciclo1
47     movb %al, (val1)
48  // input secondo valore
49  // messaggio di invito
50     movl $4, %eax
51     movl $1, %ebx
52     movl $invito, %ecx
53     movl $21, %edx
54     int  $0x80
55  // lettura stringa da tastiera
56     movl $3, %eax
57     movl $0, %ebx
58     movl $stringa, %ecx
59     movl $4, %edx       # 3 = lun. max stringa (3 + 1 per invio)
60     int  $0x80
61     movl %eax, (lun)    # lunghezza effettiva immessa
62  // converti secondo input
63     mov $0, %esi
64     xorw %ax, %ax
```

```
65      movw $0xa, %bx
66      movl (lun), %ecx
67      decl %ecx            # per non contare anche invio
68    ciclo2:
69      mulw %bx             # esegue %ax = %ax * %bx ( = 10)
70      movb stringa(%esi), %dl
71      subb $0x30, %dl
72      addb %dl, %al
73      adcb $0, %ah
74      inc  %esi
75      loopl ciclo2
76      movb %al, (val2)
77    // elaborazione
78      movb (val1), %al
79      mulb (val2)         # il risultato in %ax
80    // converte valore del ris in stringa
81      movw $0xa, %bx
82      movl $4, %esi
83    converti:
84      xor %dx, %dx
85      divw %bx             # divido ris per 10 il resto va in %dl
86      addb $0x30, %dl      # sommo 48 per avere cod. ascii
87      movb %dl, strout(%esi)
88      dec %esi
89      orw %ax, %ax        # se quoziente %ax zero - fine
90      jne converti
91    // stampa a video messaggio preliminare
92      movl $4, %eax
93      movl $1, %ebx
94      movl $msgout, %ecx
95      movl $25, %edx
96      int  $0x80
97    // stampa a video della stringa del risultato
98      movl $4, %eax
99      movl $1, %ebx
100     movl $strout, %ecx
101     movl $6, %edx
102     int  $0x80
103   fine:
104     movl $1, %eax
105     int  $0x80
```

Nelle righe da 21 a 25 il programma visualizza un messaggio con la richiesta di inserimento di un valore; tale inserimento avviene grazie alle righe da 27 a 31, mentre alla riga 32 si salva la lunghezza effettiva della stringa immessa (compreso l'invio).

Il numero immesso in ***stringa*** è, appunto, una stringa e deve essere convertito nel corrispondente valore; il procedimento usato può essere così riassunto:

- si parte dalla cifra più a sinistra e si somma ogni cifra a un totalizzatore;
- prima di passare alla cifra successiva si moltiplica il totalizzatore per ***10***;
- arrivati all'ultima cifra a destra, nel totalizzatore abbiamo il valore corrispondente alla stringa di cifre di partenza.

Le righe da 34 a 46 realizzano quanto appena descritto; in dettaglio:

- alla riga 34 si azzera il registro indice ***esi***;
- alle righe 35 e 36 si azzera il totalizzatore ***ax*** e si pone il valore ***10*** in ***bx*** per le successive moltiplicazioni;
- alle righe 37 e 38 si pone in ***cx*** la quantità si caratteri della stringa, meno uno per non considerare anche il carattere invio;
- a riga 39 inizia il ciclo da ripetere tante volte quante sono le cifre della stringa, grazie al controllo di riga 46;
- alla riga 40 si moltiplica il totalizzatore per ***10***; la prima volta sarebbe inutile in quanto ***ax*** in quel momento vale zero;
- alla riga 41 si sposta una cifra della stringa (quella indicizzata da ***esi***) in ***dl***; in questa istruzione si gestisce la stringa come un vettore come mostrato nel paragrafo 3.18;
- alla riga 42 si sottrae 30_{16} da ***dl*** per ottenere il valore corrispondente a quella cifra;
- alle righe 43 e 44 si somma tale valore nel totalizzatore, tenendo conto di eventuale riporto;
- infine il ciclo si chiude con l'incremento dell'indice a riga 45.

A questo punto in ***ax*** abbiamo il valore da spostare in ***val1*** (riga 47).

Dalla riga 48 alla riga 76 tutto il procedimento (messaggio, input, conversione) viene ripetuto per il secondo valore con l'ovvia differenza che stavolta esso viene spostato in ***val2***.

Le righe 78 e 79 svolgono la moltiplicazione tra i due valori; il risultato, che è in ***ax***, deve essere convertito in stringa.

Per fare ciò si usa il seguente algoritmo:

- si divide ciclicamente il valore per ***10***;
- ad ogni divisione, il resto rappresenta una cifra da porre nella stringa (partendo da destra);
- il ciclo si arresta quando il quoziente della divisione è zero.

Le righe da 81 a 90 eseguono quanto appena illustrato; in dettaglio:

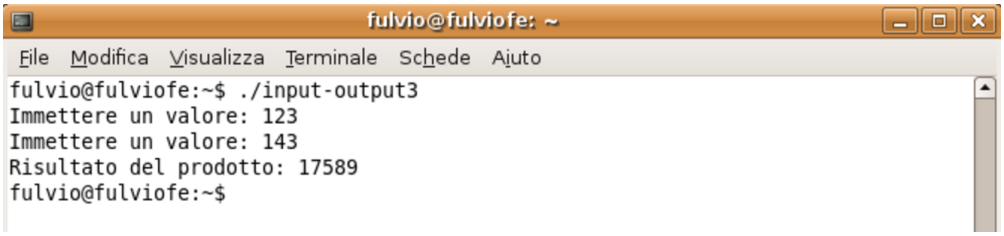
- alla riga 81 si pone il valore ***10*** in ***bx*** per le successive divisioni;
- alla riga 82 si imposta l'indice per il riempimento della stringa di output sulla posizione prevista più a destra; nel nostro caso è la quinta posizione (quella di indice quattro);
- il ciclo di conversione inizia alla riga 83 ed è con controllo in coda;
- alle righe 84 e 85 si divide ***dx:ax*** per ***10***; siccome il nostro valore da dividere è solo in ***ax***, ***dx*** lo azzeriamo;
- la divisione pone il quoziente in ***ax*** e il resto in ***dx*** (in realtà in ***dl*** visto che è minore di dieci); alla riga 86 sommiamo 30_{16} per avere in ***dl*** il codice ASCII della cifra corrispondente al valore del resto;
- alla riga 87 tale cifra viene inserita nel vettore ***strout*** nella posizione indicizzata da ***esi***;

- alla riga 88 si decrementa l'indice, mentre alla riga 89 si verifica se ***ax*** vale zero; se tale condizione è falsa, grazie alla riga 90 si passa alla prossima iterazione.

Conclusa la conversione del valore da visualizzare, il programma emette un opportuno messaggio (righe da 92 a 96) e poi la stringa contenente il risultato (righe da 98 a 102).

Nella figura C.6 vediamo gli effetti dell'esecuzione del programma.

Figura C.6.



[1] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/input-output1.s* ⟩.

[2] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/input-output2.s* ⟩.

[3] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/input-output3.s* ⟩.

Appendice D

Le macro dell'assembly AT&T

Nell'ultimo esempio dell'appendice C ci sono parti del codice che si ripetono in modo identico; ciò avviene nella parte iniziale per l'input dei due valori da moltiplicare.

Una cosa simile è però il più possibile da evitare, sia per eliminare la noia di dover scrivere più volte le stesse istruzioni, sia per non correre il rischio di dimenticare qualche gruppo di istruzioni nel momento che si apportano delle modifiche.

A tale proposito si può allora ricorrere alle '**`macro`**' che sono gruppi di istruzioni che si scrivono una volta sola e si «richiamano» dove occorre.

Chiariamo subito che il concetto di «richiamo» di una macro non ha niente a che vedere con quello analogo relativo ai sottoprogrammi; quando l'assemblatore incontra il riferimento ad una macro non fa altro che copiare fisicamente le relative istruzioni in quel punto del programma (si parla anche di «espansione» della macro).
Per questo motivo in una macro non possono esserci delle etichette, in quanto altrimenti comparirebbero più volte nel sorgente «espanso», causando errore da parte dell'assemblatore.

Nell'assembly con sintassi AT&T le macro si dichiarano, solitamente all'inizio del sorgente, con la direttiva ***.macro*** seguita da un nome, e si chiudono con la direttiva ***.endm***; il richiamo si effettua invece scrivendo semplicemente il nome.

Usando le macro nell'esempio dell'appendice C si evita di scrivere due volte buona parte delle istruzioni relative alle fasi di input (purtroppo non tutte, perché, come detto, le parti contenenti le etichette dei cicli non possono essere inserite nelle macro), ottenendo il listato seguente.[1]

```
/*
Programma:     io-macro.s
Autore:        FF
Data:          gg/mm/aaaa
Descrizione:   Acquisizione e stampa con macro
*/
.bss
stringa:   .string ""                          # stringa per input
.data
invito:    .string "Immettere un valore: "     # senza a capo
msgout:    .string "Risultato del prodotto: " # senza a capo
lun:       .long   0                           # lunghezza input
val1:      .byte   0                           # primo valore di input
val2:      .byte   0                           # secondo valore di input
strout:    .string "     \n"                   # stringa per output (5 car)
.text
.globl _start
.macro messaggio_invito
   movl $4, %eax
   movl $1, %ebx
   movl $invito, %ecx
   movl $21, %edx
   int  $0x80
.endm
.macro lettura_stringa
   movl $3, %eax
   movl $0, %ebx
   movl $stringa, %ecx
```

```
   movl $4, %edx       # 4 = lun. max stringa (3 + 1 per invio)
   int  $0x80
   movl %eax, (lun)    # lunghezza effettiva immessa
.endm
.macro prepara_ciclo
   mov $0, %esi
   xorw %ax, %ax
   movw $0xa, %bx
   movl (lun), %ecx
   decl %ecx            # per non contare anche invio
.endm
.macro corpo_ciclo
   mulw %bx             # esegue %ax = %ax * %bx ( = 10)
   movb stringa(%esi), %dl
   subb $0x30, %dl
   addb %dl, %al
   adcb $0, %ah
   inc  %esi
.endm
_start:
// input primo valore
   messaggio_invito
   lettura_stringa
// converti primo input
   prepara_ciclo
ciclo1:
   corpo_ciclo
   loopl ciclo1
   movb %al, (val1)
// input secondo valore
   messaggio_invito
   lettura_stringa
// converti secondo input
   prepara_ciclo
ciclo2:
   corpo_ciclo
   loopl ciclo2
   movb %al, (val2)
// elaborazione
   movb (val1), %al
   mulb (val2)         # il risultato in %ax
// converte valore del ris in stringa
   movw $0xa, %bx
   movl $4, %esi
converti:
   xor %dx, %dx
   divw %bx             # divido ris per 10 il resto va in %dl
   addb $0x30, %dl      # sommo 48 per avere cod. ascii
   movb %dl, strout(%esi)
   dec %esi
   orw %ax, %ax        # se quoziente %ax zero - fine
   jne converti
// stampa a video messaggio preliminare
   movl $4, %eax
```

```
    movl $1, %ebx
    movl $msgout, %ecx
    movl $25, %edx
    int  $0x80
// stampa a video della stringa del risultato
    movl $4, %eax
    movl $1, %ebx
    movl $strout, %ecx
    movl $6, %edx
    int  $0x80
fine:
    movl $1, %eax
    int  $0x80
```

[1] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨*allegati/programmi-assembly/io-macro.s* ⟩.

Appendice E

Assembly AT&T e linguaggio '**c**'

In questa appendice vediamo come far convivere codice assembly e codice '**c**' in uno stesso programma eseguibile.

Il motivo per cui questo può essere importante è che certe funzioni possono essere scritte in assembly al fine di sfruttare in pieno le caratteristiche dell'hardware e di ottimizzare al massimo le prestazioni, mentre possono rimanere in '**c**' tutte quelle operazioni, come l'input e l'output, in cui esso è di utilizzo molto più semplice.

In parte questa convivenza l'abbiamo già sperimentata usando chiamate a funzioni del '**c**' in alcuni programmi assembly (vedi paragrafo 3.15); adesso esploriamo altre possibilità date da: '**assembly inline**' e richiamo di funzioni esterne assembly da un programma '**c**'.

E.1 L'assembly inline

In molti ambienti (e GNU/Linux è uno di questi) è possibile inserire istruzioni assembly in un programma '**c**'; si parla in tal caso di '**assembly inline**'.

La sintassi da utilizzare è la seguente:

```
__asm__ (
istruzioni assembly
: operandi di output (opzionali)
: operandi di input (opzionali)
: lista dei registri modificati (opzionale)
);
```

All'inizio dobbiamo scrivere ***asm*** preceduta e seguita da due *underscore*.

Il primo parametro, è costituito dall'insieme delle istruzioni assembly, ognuna chiusa da «;»; tali istruzioni devono essere ognuna tra virgolette a meno che non si scrivano sulla stessa riga.

Il secondo parametro è costituito dagli operandi che ospiteranno i relativi risultati (opzionali perché la nostra *routine* può non prevedere valori di output).

Il terzo parametro sono i valori di input (opzionali in quanto possono essere assenti).

L'ultimo parametro è la lista dei registri modificati (*clobbered*) nell'ambito delle istruzioni assembly (opzionale perché potremmo non modificarne alcuno o gestire il loro salvataggio e ripristino con lo stack).

Vediamo subito un piccolo esempio in cui inseriamo la numerazione delle righe per la successiva descrizione.[1]

```
1   /*
2   Programma:     inline1.c
3   Autore:        FF
4   Data:          gg/mm/aaaa
5   Descrizione:   Primo esempio di assembly inline
6   */
7   #include <stdio.h>
8   int main()
9   {
10     int val1, val2;
11     printf("Inserire val1: ");
12     scanf("%d",&val1);
```

```
13      printf("Inserire val2: ");
14      scanf("%d",&val2);
15      __asm__("movl %0, %%eax; movl %1, %%ebx;"
16         "xchgl %%eax, %%ebx;"
17         : "=r" (val1),
18           "=r" (val2)
19         : "r" (val1),
20           "r" (val2)
21         : "%eax", "%ebx"
22      );
23      printf ("Valori scambiati val1=%d val2=%d\n",val1,val2);
24      return 0;
25  }
```

Le righe fino alla 14 e da 23 a 25 contengono «normali» istruzioni '`c`' che non richiedono commenti.

Alle righe 15 e 16 troviamo le istruzioni assembly; notiamo che per fare uso dei registri occorre raddoppiare la quantità di «%» nel prefisso in quanto un solo «%» viene usato per gli *alias*.

In queste istruzioni gli *alias* sono ***%0*** e ***%1*** associati ai primi (e unici) due operandi di input.

Le tre istruzioni assembly sono molto semplici: vengono posti i due operandi di input rispettivamente in ***eax*** e ***ebx*** e poi i valori dei due registri vengono scambiati.

Alle righe 17 e 18 vengono specificati gli operandi di output ***%0*** e ***%1*** associati rispettivamente alle variabili ***val1*** e ***val2***; la presenza del simbolo di «=» davanti al nome indica che si tratta di valori in output e si richiede che per essi il sistema userà dei registri generali (simbolo «r»).

Alle righe 19 e 20 vengono specificati gli operandi di input con lo stesso criterio.

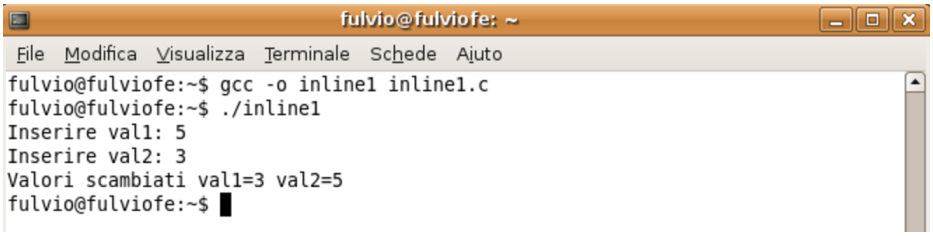
Con la riga 21 si indicano i registri modificati nella *routine* assembly.

Infine, alla riga 22, si chiude la definizione di tale *routine*.

La necessità dell'indicare i registri modificati sta nel fatto che non sappiamo a priori quali registri l'assembly userà per ospitare gli operandi che abbiamo dichiarato, quindi dobbiamo informarlo di quali sono i registri che devono essere riservati per le operazioni svolte dalle istruzioni della *routine*.

Nella figura E.3 vediamo la compilazione e l'esecuzione del programma.

Figura E.3.



Nella fase di assegnazione degli operandi si possono anche specificare con esattezza i registri o le etichette di memoria da associare (invece di usare «r») indicandoli per esteso o con le seguenti abbreviazioni:

- ***"a"***: per ***eax***;

- ***"b"***: per ***ebx***;
- ***"c"***: per ***ecx***;
- ***"d"***: per ***edx***;
- ***"D"***: per ***edi***;
- ***"S"***: per ***esi***;
- ***"m"***: per una etichetta di memoria.

Nel prossimo esempio vediamo un programma che chiede in input un valore intero ***n*** e poi ne calcola il quadrato in assembly come somma dei primi ***n*** dispari; il risultato ovviamente viene visualizzato con istruzioni del 'c'.[2]

```
1   /*
2   Programma:     inline2.c
3   Autore:        FF
4   Data:          gg/mm/aaaa
5   Descrizione:   Secondo esempio assembly inline: quadrato = somma di n dispari
6   */
7   #include <stdio.h>
8   int main()
9   {
10     int n,q;
11     printf("Inserire n: ");
12     scanf("%d",&n);
13     __asm__("xorl %%eax, %%eax;"
14        "addl %%ebx, %%ebx;"
15        "movl $1, %%ecx;"
16        "ciclo: addl %%ecx, %%eax;"
17        "addl $2, %%ecx;"
18        "cmpl %%ecx, %%ebx;"
19        "jg ciclo;"
20        : "=a" (q)
21        : "b"  (n)
22     );
23     printf ("Quadrato = %d\n",q);
24     return 0;
25  }
```

Commentiamo le righe della *routine assembly* iniziando da quelle di assegnazione degli operandi.

In questo caso, alle righe 20 21, vengono associate le variabili ***q*** e ***n*** direttamente ai registri ***eax*** (in output) e ***ebx*** (in input) e quindi non serve specificare la lista dei registri modificati (infatti il relativo parametro qui è assente).

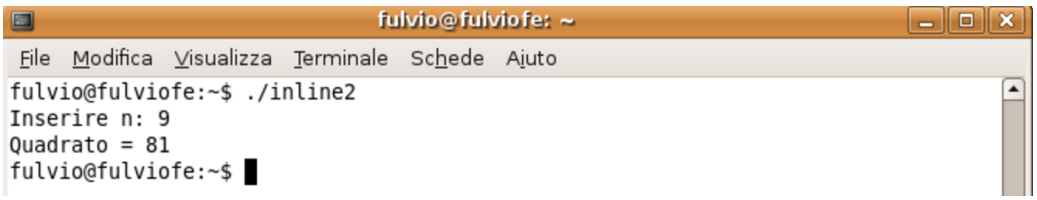
Alla riga 13 viene azzerato il registro ***eax*** che funge da accumulatore, mentre alla riga 14 si raddoppia il valore di ***ebx*** che corrisponde a ***n*** in modo che il ciclo sui numeri dispari continui per valore del contatore minore si ***2*n***.

Alla riga 15 si imposta il contatore ***ecx*** e alla successiva inizia il ciclo con l'accumulo del valore del contatore in ***eax***.

Alla riga 17 si incrementa di due il contatore e poi si confronta con ***ebx*** proseguendo il ciclo se quest'ultimo è maggiore (righe 18 e 19).

Nella figura E.5 vediamo l’esecuzione del programma.

Figura E.5.



E.2 Chiamata di procedure assembly da programmi ‘c’

Un’altra alternativa per far convivere codice ‘**c**’ e assembly in ambiente GNU/Linux consiste nel richiamo da un programma ‘**c**’ di moduli assembly esterni.

Questo è possibile alle seguenti condizioni:

- nel programma ‘**c**’ i moduli assembly devono essere dichiarati come ‘**esterni**’;
- nel modulo assembly la procedura deve essere dichiarata ‘**globale**’ tramite la direttiva ‘**.globl**’ (questo la rende richiamabile in altri moduli);
- si devono compilare separatamente i due moduli per ottenere i rispettivi file ‘**oggetto**’ e poi collegarli in un unico eseguibile.

Occorre poi ricordare che i parametri eventualmente passati alla funzione assembly si trovano nello stack a partire dall’ultimo a destra, mentre l’eventuale valore di ritorno va inserito in ***al***, o ***ax***, o ***eax***, o ***edx:eax***, secondo che sia lungo 8, 16, 32 o 64 bit.

Chiariamo tutto con un esempio che consiste ovviamente in due sorgenti: uno per il programma principale in ‘**c**’ e uno per la funzione assembly.

Nel programma principale predisponiamo l’input di due valori interi, il richiamo della funzione assembly per il calcolo della potenza con base ed esponente i due valori e la stampa del risultato.[3]

```
/*
Programma:      c-asm.c
Autore:         FF
Data:           gg/mm/aaaa
Descrizione:    Richiamo di asm da c - main in c
*/
extern int pot_asm(int b, int e);
#include <stdio.h>
int main()
{
   int base, esp, potenza;
   printf("Inserire base: ");
   scanf("%d",&base);
   printf("Inserire esponente: ");
   scanf("%d",&esp);
   potenza=pot_asm(base,esp);
   printf ("Valore della potenza = %d\n",potenza);
   return 0;
}
```

Nella funzione assembly effettuiamo il calcolo della potenza come successione di prodotti.[4]

```
/*
Programma:      c-asm-funz.s
Autore:         FF
Data:           gg/mm/aaaa
Descrizione:    Richiamo di asm da c - funzione asm per potenza con succ. *
*/
.data
.text
.globl pot_asm              # funzione deve essere globale
pot_asm:
   pushl %ebp               # salvo %ebp precedente (event. chiamate nidificate)
   movl %esp, %ebp          # imposto %ebp
   movl 8(%ebp), %ebx       # leggo primo par. (base) dopo 8 byte dalla cima
                            # della pila: 4 per ebp e 4 per ind. di rientro
   movl 12(%ebp), %ecx      # leggo altro parametro (espon)
   xorl %edx, %edx          # azzero %edx coinvolto nella mul
   movl $1, %eax            # imposto accumulatore
ciclo:
   mull %ebx                # %eax = %eax*%ebx
   loopl ciclo               # il ciclo va fatto %ecx (espon) volte
                            # al termine il ris si trova in %eax o in
                            # edx:%eax come atteso dal chiamante
   popl %ebp                # ripristino %ebp precedente
   ret
```

Notiamo come, trattandosi di una procedura assembly, ci siamo comportati, relativamente alla gestione dello stack e all'istruzione di terminazione, allo stesso modo di quando abbiamo scritto procedure «normali» inglobate nei sorgenti assembly.

Per compilare separatamente i due sorgenti eseguiamo i comandi:

```
$ gcc -c -o c-asm.o c-asm.c
```

```
$ gcc -c -o c-asm-funz.o c-asm-funz.s
```

e poi colleghiamoli con il comando:

```
$ gcc -o c-asm c-asm.o c-asm-funz.o
```

Nella figura E.8 sono mostrati appunto tali comandi seguiti dall'esecuzione dell'eseguibile ottenuto.

Figura E.8.

[1] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨ *allegati/programmi-assembly/inline1.c* ⟩.

[2] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨ *allegati/programmi-assembly/inline2.c* ⟩.

[3] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨ *allegati/programmi-assembly/c-asm.c* ⟩.

[4] una copia di questo file, dovrebbe essere disponibile anche qui: ⟨ *allegati/programmi-assembly/c-asm-funz.s* ⟩.