1 Procedure

La procedura è il modo fondamentale per dividere un programma in molti componenti elementari, ciascuno dei quali risolve una parte separata del problema globale.
Con un accorto uso delle procedure anche un programma in Assembly può diventare moderatamente leggibile.
L'elemento distintivo della procedura è il "ritorno". Invece di saltare "a sola andata", come succede con le jump, la chiamata a procedura è un meccanismo automatico per fare un salto con ritorno, cioè tale che, alla conclusione della procedura, il flusso del programma riprenda dal punto dal quale la procedura era stata chiamata.
La possibilità di ritornare apre la strada all'utilizzazione delle procedure da molti punti dello stesso programma chiamante.
Questo è un vantaggio fondamentale, che rende possibile programmare realizzando sezioni di programma specializzate nella soluzione di problemi significativi, che vengono risolti una volta per tutte, senza doverne riscrivere il codice tutte le volte che serve.

Quando si salta ad una procedura si usa l'istruzione CALL (chiama), che indicherà anche l'indirizzo al quale il programma deve saltare. In Assembly quell'indirizzo è quasi sempre indicato tramite una etichetta.
Con la chiamata il flusso del programma viene trasferito all'inizio della procedura, però prima di fare questo salto l'istruzione **CALL** farà in modo di "ricordare" il punto da cui era partita, per far sì che il programma possa tornare indietro dopo la fine della procedura.
Il codice della procedura eseguirà tutte le istruzioni cui è preposto, fino a che la sua funzione sarà esaurita. Allora il flusso del programma verrà fatto ritornare all'istruzione successiva alla CALL. Ciò verrà indicato dalla presenza dell'istruzione **RET** (Return).

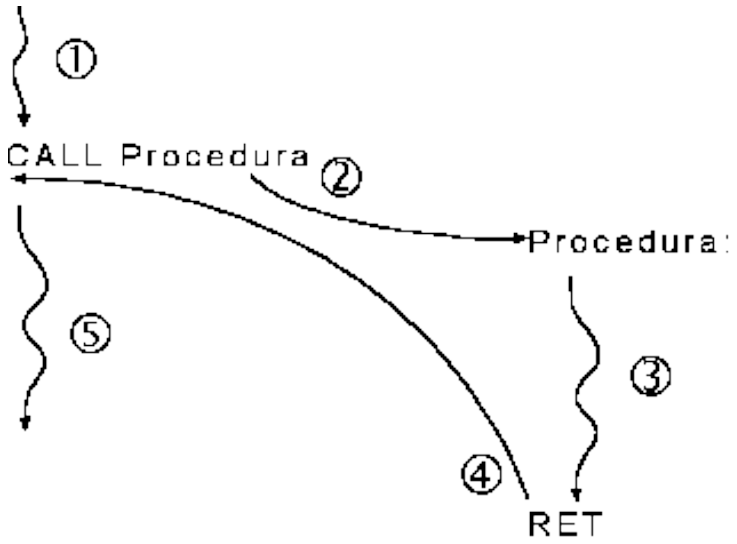

Figura 1: chiamata di una procedura

Nella figura "chiamata di una procedura" l'indirizzo è rappresentato dalla label "Procedura" (naturalmente si può dare un nome qualsiasi) e le frecce ondulate rappresentano l'ordine di esecuzione del programma. Nel flusso del programma ci sono dunque due salti: un primo (punto 2) dovuto alla chiamata alla procedura, ed un secondo (punto 4) dovuto al ritorno.
La presenza di una chiamata individua dunque due parti di codice, una detta "programma principale" (**main** program) nel quale il flusso del programma è attualmente in corso, ed un'altra parte detta "**procedura**", "**routine**", "**subroutine**" od anche "**function**", alla quale si salta, mantenendo però la possibilità di ritornare all'istruzione successiva a quella del salto.
Nella figura il flusso indicato dai numeri 1 e 5 fa parte del programma principale, mentre il flusso 3 è la procedura.

1.1 Il diabolico trucco

Vediamo ora qual è il meccanismo a basso livello che fa funzionare le procedure nel modo descritto.
Il funzionamento è diabolicamente semplice, ma di straordinaria efficacia.
Per capirlo occorre dare risposta a due domande:
1 - Cosa serve sapere per poter ritornare all'istruzione seguente a quella che si sta eseguendo?
2 – Dove si può memorizzare questa informazione?
Le risposte che si trovano sono le più semplici:

1 - Alla fine della fase di fetch di un'istruzione il Program Counter della CPU punta alla prossima istruzione da eseguire. Quindi è il P.C. l'informazione che ci serve per poter ritornare.
2- Lo stack è la struttura dati ideale per memorizzare l'indirizzo di ritorno, perchè esso ritorno è un'informazione di tipo temporaneo e che si adatta perfettamente alla natura LIFO dello stack. Infatti le procedure si chiamano una "dentro all'altra" e l'indirizzo di ritorno che serve è sempre e solo quello dell'ultima procedura chiamata.
Perciò l'istruzione CALL memorizzerà nello stack il valore corrente del Program Counter, poi salterà all'indirizzo della procedura, mentre l'istruzione RET prenderà dallo stack il numero che attualmente ne "emerge" e lo metterà nel Program Counter, effettuando così il salto a ritroso.
Commentiamo ora una figura che illustra in dettaglio questo semplice meccanismo:

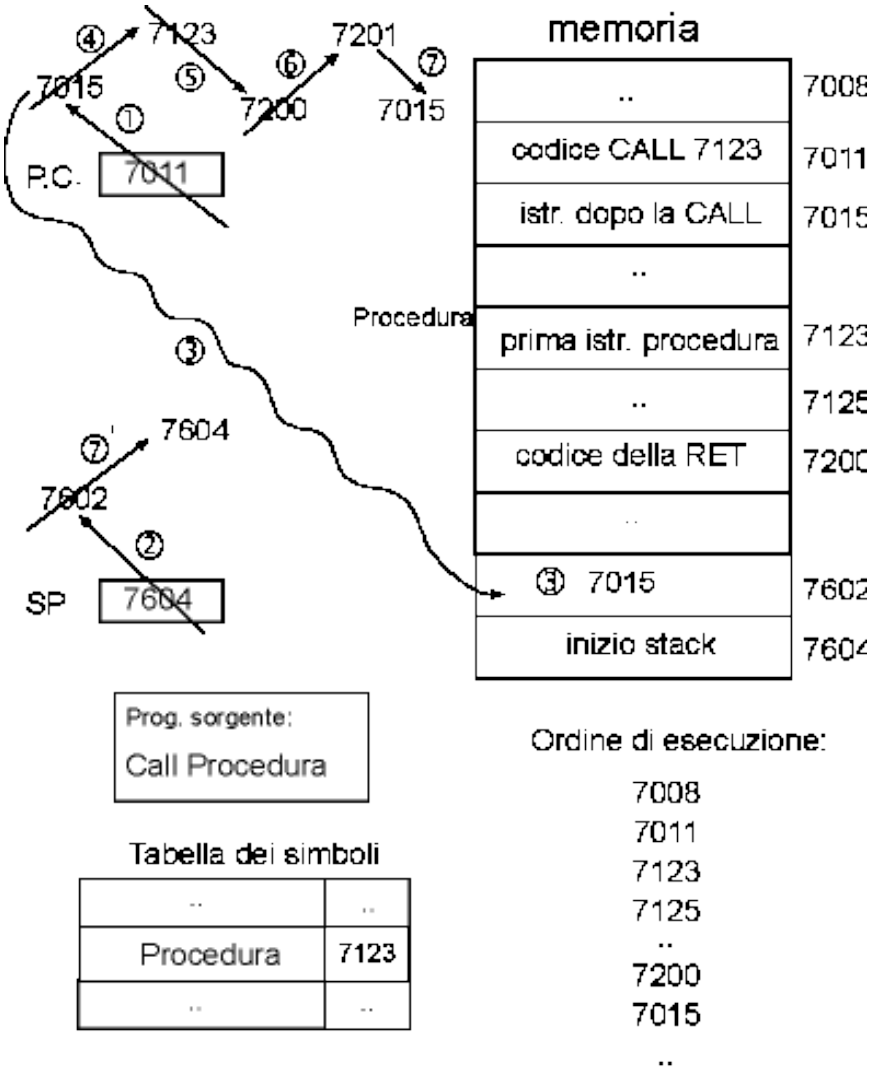

Figura 2: realizzazione di una chiamata a procedura[1]

Nella figura è illustrato un esempio di come potrebbe essere configurata la memoria al momento della chiamata.

La sequenza di ciò che succede durante l'accesso ad una procedura è dunque la seguente:
1 - Fine fetch dell'istruzione CALL, P.C. aggiornato, diventa 7015
2 – Inizio salvataggio dell'indirizzo di ritorno nello stack, SP decrementato
3 - Scrittura P.C. nello stack, come indirizzo di ritorno
4 – Salto a procedura (indirizzo della procedura in P.C. (7123 in Figura 2))
5 – Esecuzione di tutta la procedura, fino all'istruzione RET (in Figura 2: indirizzo 7200 nel P.C.)
6 – Fetch istruzione RET, che nell'esempio occupa un Byte (in Figura 2: P.C. = 7201)
7 – Lettura indirizzo di ritorno dallo stack (in Figura 2: P.C. = 7015)
7' - Aggiornamento di SP (in Figura 2: SP = SP + 2 = 7604)
8 – Continuazione del programma principale (in Figura 2: da indirizzo 7015)

1 La figura è generale, non fa riferimento all'8086, per il quale le nomenclature sono leggermente diverse (in particolare il P.C. è CS:IP).

Nota: i punti 1, 2, 3 e 4 fanno parte dell'istruzione CALL, i punti 6, 7 e 7' fanno parte dell'istruzione RET, pertanto 1, 2, 3, 4 e 6, 7, 7' sono "indivisibili", vengono eseguite tutte "insieme", in parte anche contemporaneamente.

Quando un programma sta eseguendo una procedura deve sempre concluderla. In taluni casi si può essere tentati all'uscita da una procedura attraverso una jump. Questo non si deve mai fare, pena la "morte" certa del programma!
Se ci si ricorda del fatto che la chiamata a procedura lascia nello stack l'indirizzo di ritorno si può capire che uscendo da una procedura con una semplice jump si lascia lo stack sbilanciato, con esiti incontrollabili e di solito fatali!

1.1.1 Procedure annidate

Durante l'esecuzione di una procedura è lecito e naturale chiamarne un'altra.
Analogamente al caso dei cicli, due procedure una dentro all'altra vengono dette **nidificate**, **annidate**; o anche "**innestate**".
Se si chiama una procedura da un'altra, quando la procedura chiamata per ultima ritorna la prima potrà continuare ad eseguire a sua volta, fino alla sua conclusione con una RET. Ciò viene illustrato nella seguente figura:

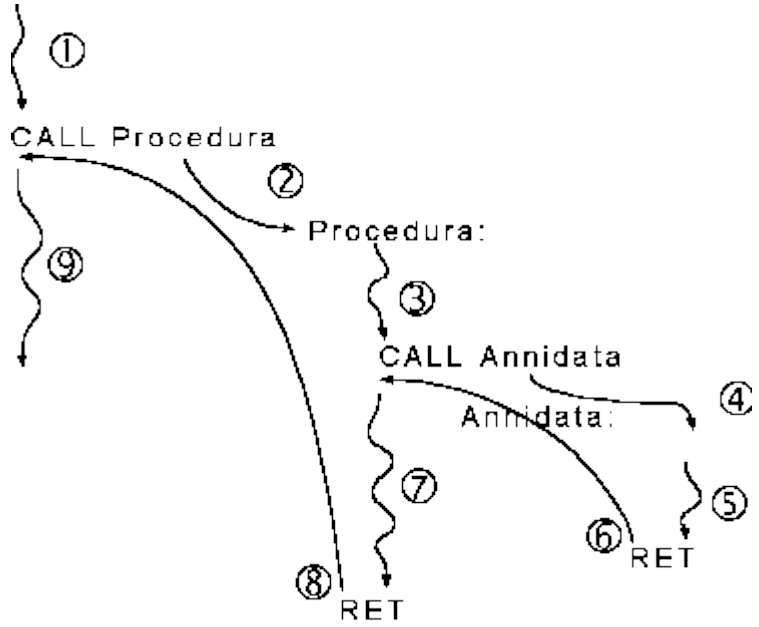

Figura 3: procedure annidate

All'interno della procedura "Procedura" viene chiamata la procedura"Annidata". Il flusso del programma principale (1 e 9) viene interrotto dalla chiamata a "Procedura" e riprende solo dopo che essa è tornata (8). Lo stesso succede per "Procedura" il cui flusso (3 e 7) viene interrotto dalla chiamata ad "Annidata" fino a che essa non torna (6).
A basso livello non ci sono problemi nel nidificare le procedure, dato che ci viene incontro la natura LIFO dello stack. Infatti se una procedura chiama un'altra procedura l'ordine con cui si fa il ritorno è inverso rispetto a quello delle chiamate.
Ad ogni chiamata annidata lo stack accoglie un nuovo indirizzo di ritorno. Fino a che lo stack non si esaurisce, a forza di accogliere indirizzi di procedure "in sospeso", non c'è nessun problema ad accettare molte chiamate nidificate una dentro l'altra.
La procedura nidificata esegue e, se è fatta bene, ritorna. Mentre ritorna riprende il suo indirizzo di ritorno dallo stack. In questo modo restituisce alla procedura chiamante uno stack "pulito".
Ora in cima allo stack emerge di nuovo l'indirizzo di ritorno della prima procedura annidata, che verrà usato dalla prossima RET per tornare al programma principale.
La seguente figura mostra ciò che è contenuto nello stack durante le chiamate di esempio indicate nella figura precedente:

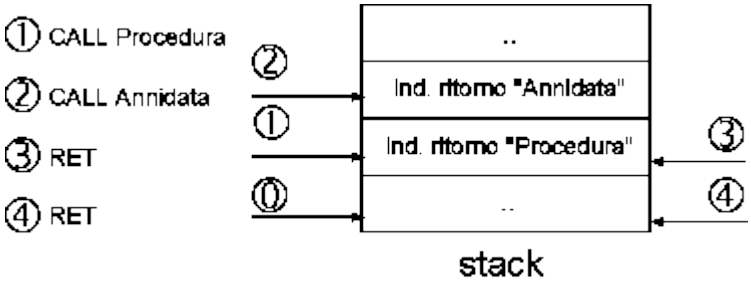

Figura 4: stato dello stack durante procedure annidate

Le frecce mostrano dove punta lo stack pointer dopo ogni istruzione indicata (per i numeri delle istruzioni fare riferimento alla figura precedente).
Dato il funzionamento dello stack non ci sono specifici limiti al numero di procedure che si possono nidificare una dentro all'altra. Naturalmente ciò è condizionato dalla capienza dello stack, che si può esaurire, e dal buon comportamento di tutte le procedure chiamate, che devono tornare correttamente, lasciando lo stack bilanciato.

1.2 Usare le procedure

Ci sono almeno tre ragioni importanti per le quali usare le procedure è decisivo:
- risparmiare la duplicazione di codice
- rendere il programma più facile da correggere
- documentare il programma

Risparmio di memoria
La possibilità di ritornare al punto da cui si chiama una procedura permette di isolare il codice che esegue una funzione complessa e chiamarlo da qualsiasi punto del programma principale senza bisogno di riscriverlo tutte le volte. Ciò può comportare un considerevole risparmio nell'occupazione della memoria, anche se implica anche un leggerissimo aggravio dei tempi di esecuzione, dato che la presenza di CALL e RET richiede alcuni cicli di macchina per il salvataggio ed il ripristino dell'indirizzo di ritorno.
Correzione
Un codice ben strutturato in procedure è più facile da correggere per due ragioni:
- il codice della procedura è scritto solo una volta: è necessario "aggiustarlo" solo una volta quando non funziona, senza riportare le modifiche in molte parti del programma
- durante il processo di debugging l'individuazione degli errori è facilitata, perché sono chiari i valori di ciò che "entra" ed "esce" dalle procedure, per cui è più facile individuare la parte del programma responsabile dell'errore.

Documentazione
Dando alle procedure nomi intelligenti è più facile capire il flusso del programma. Se nel definire i nomi si usa sempre il criterio si spiegare chiaramente a cosa serve quello che si sta facendo, il codice stesso si "autodocumenterà", e si possono rendere inutili molti commenti.
Per chi legge il programma non c'è bisogno di capire nel dettaglio come funziona internamente una procedura, interessa solo sapere cosa fa e come si deve usare.

Nella stesura del codice le procedure possono essere scritte prima o dopo il programma principale. Di solito è meglio scriverle prima, perché in questo modo il compilatore può conoscere il loro indirizzo prima che la sua etichetta sia effettivamente usata nella corrispondente CALL. Infatti alcuni compilatori non riescono a fare riferimento a procedure che sono scritte dopo la chiamata (forward reference).

1.2.1 Parametri di una procedura

Abbiamo appena visto come uno dei maggiori vantaggi nello scrivere procedure è il fatto che esse possono essere chiamate da molti punti del programma principale. Per sfruttare pienamente questo vantaggio è utile poter cambiare ogni volta si chiama una procedura i numeri sui quali essa lavora.

Ciò che può cambiare ogni volta che una procedura è chiamata si dice "**parametro**" di quella procedura.

Usando i parametri in modo accorto è possibile generalizzare l'uso della procedura e rendere possibile il suo uso anche in contesti diversi da quelli per i quali è stata scritta inizialmente. Tutto ciò senza cambiare nulla del codice della procedura, ma modificando solo il valore dei parametri prima della chiamata.
L'utilizzatore di una procedura deve solo sapere qual è la funzione che la procedura svolge, quali parametri ha e come le si devono passare quei parametri.
In definitiva egli può disinteressarsi del codice della procedura e conoscere solo la sua "interfaccia", cioè le sue caratteristiche esterne. Questo permette anche di suddividere il lavoro, perchè ad occuparsi di scrivere il programma principale e la procedura potrebbero essere persone diverse.

Parametri in ingresso ed in uscita
Ciò che passiamo ad una procedura come dati da elaborare sono i suoi **parametri d'ingresso**.

Ciò che viene elaborato da una procedura e che viene restituito al programma principale sono i **parametri d'uscita** della procedura.

Un blocco per le procedure
Se si vuole dare un'evidenza particolare alle procedure nei diagrammi di flusso si può usare un blocco come questo:

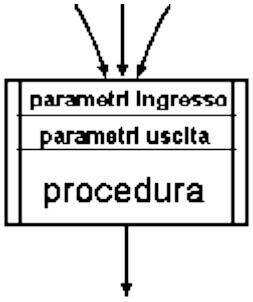

Figura 5: un blocco per le procedure nei diagrammi di flusso

Per chiarezza indichiamo nel blocco i parametri della procedura, distinguendo fra quelli in ingresso e quelli in uscita.

Un esempio di analisi: ordinamento di un vettore
Spesso è il problema stesso che "obbliga" a trovare i parametri di una procedura. Analizziamo in dettaglio un problema concreto per vedere come scegliere procedure e parametri.
Supponiamo di dover ordinare in ordine decrescente un vettore di 100 numeri.
Esistono centinaia di algoritmi per l'ordinamento; quello che utilizzeremo è molto semplice e piuttosto efficiente.
Per prima cosa risolviamo il problema del primo elemento del vettore ordinato.
Il primo elemento sarà quello che ha il valore massimo; dunque si può individuare un sottoproblema interessante nella ricerca del massimo del vettore: la ricerca del massimo.
Questo è un sottoproblema ben definito e che molto probabilmente può essere usato in occasioni diverse dall'ordinamento del vettore. E' dunque un buon candidato per divenire una procedura. Vedremo poi, nel corso dell'analisi, che la ricerca del massimo verrà utile anche in seguito, in questo stesso algoritmo, dopo aver sistemato il primo valore.
Confiniamo il problema della ricerca del massimo in una procedura e supponiamo di averlo risolto.
Chiamiamo v() il vettore e TrovaMassimo la procedura.
TrovaMassimo deve comunicare al programma chiamante qual è il valore del massimo. Questa è una quantità non nota a priori, che cambia ogni volta che la procedura viene eseguita. Perciò deve essere un parametro di uscita della procedura. Lo chiamiamo ValMassimo.

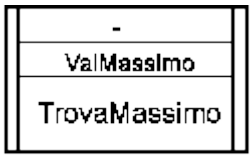

Figura 6: esigenza di una prima procedura, un parametro di uscita

Altri parametri verranno introdotti quando ve ne sarà bisogno.

Primo elemento del vettore ordinato
Per sistemare il primo elemento del vettore bisogna mettere il massimo al primo posto; scambieremo il primo elemento del vettore con quello che contiene il valore massimo. A questo punto il primo elemento del vettore sarà "a posto" e non dovrà più essere toccato (Figura 8).
Per fare lo scambio bisogna sapere dov'è l'elemento del vettore che contiene il massimo.
La procedura TrovaMassimo può sapere dove si trova l'elemento massimo, mentre il programma principale no.
Dunque i parametri che abbiamo non bastano; siamo costretti ad inventare un nuovo parametro di uscita dei TrovaMassimo. Lo chiamiamo IndMassimo e sarà l'indirizzo dell'elemento che contiene il massimo.

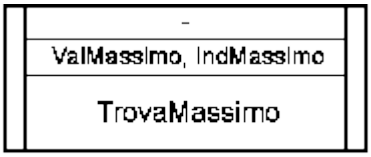

Figura 7: esigenza di un altro parametro: due parametri di uscita

Per ottenere il primo elemento ordinato bisogna scambiare l'elemento iniziale con IndMassimo.
Possiamo affidare il compito dello scambio ad una procedura. La procedura "Scambio" ha tutte le caratteristiche per essere una buona procedura, perché risolve un problema ben definito e piuttosto generale, dunque può essere riutilizzata facilmente.

!!!! figura scambio
Figura 8: sistemazione del primo elemento

Ordinare tutti gli altri elementi
Una volta messo a posto il primo elemento del vettore ordinato, bisogna sistemare il secondo, che è il massimo fra gli elementi "rimasti" (punto (2) di Figura 10).
Si può utilizzare la stessa procedura TrovaMassimo, a patto di inventare un nuovo parametro. Infatti TrovaMassimo cerca fra tutti gli elementi del vettore, mentre ora non si deve cercare fra gli elementi "già a posto", che sono stati già ordinati.
Introduciamo perciò il parametro PrimoElemento, che è indirizzo del primo elemento del vettore dal quale cercare il massimo.
Esso è un parametro che viene passato dal programma chiamante alla procedura, per cui è un parametro di ingresso.

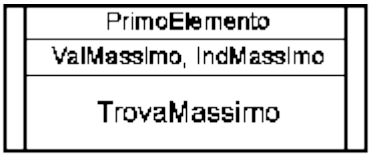

Figura 9: esigenza di un altro parametro: un parametro in ingresso

Al ritorno da TrovaMassimo si dovranno scambiare i contenuti degli elementi di indirizzo PrimoElemento ed IndMassimo. Così il secondo valore sarà a posto.
Per gli altri valori si procederà in modo simile, cercando ogni volta fra un numero minore di elementi, fino a che non si sistema il novantanovesimo.

!!!! figura fine ordinamento
Figura 10: completamento dell'ordinamento

Per completare l'analisi si può ora disegnare un diagramma di flusso, che aiuterà nella stesura del codice.

!!!! figura diagramma di flusso ordinamento
Figura 11: diagramma di flusso dell'ordinamento

La realizzazione in linguaggio Assembly 8086 di questo programma è presentata in fondo a questo capitolo, al paragrafo "".
Volendo generalizzare ulteriormente questo programma lo si potrebbe trasformare a sua volta in una procedura, che può essere utilizzata da altri programmi.
Si può infine individuare un nuovo parametro, che rende la procedura ancora più flessibile.
L'analisi del programma è stata fatta considerando un vettore di 100 elementi. La procedura TrovaMassimo diventa ancora più flessibile se le si aggiunge un parametro che indichi in qualche modo la lunghezza del vettore, come il numero degli elementi o l'indirizzo dell'ultimo elemento.

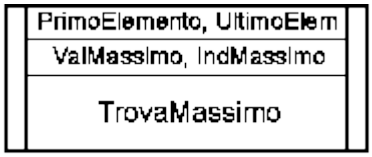

Figura 12: un nuovo parametro in ingresso per il riuso di TrovaMassimo con ogni vettore

L'individuazione di parametri che rendano più flessibili le procedure e, più in generale, i programmi viene detta "**parametrizzazione**".
Tutte le volte che si scrive una procedura ha senso fermarsi un attimo a riflettere su cosa si possa parametrizzare; una procedura ben parametrizzata ha più probabilità di essere riutilizzata.

Passaggio dei parametri

Per passare parametri ad una procedura il programma chiamante li deve scrivere in memoria o in un registro, prima della CALL. Con la CALL la procedura viene eseguita e può leggere i parametri ed utilizzarli.
Bisogna fare una distinzione, importante, fra il "modo" con cui vengono passati i parametri ed il "mezzo" attraverso il quale essi passano alle procedure.

Modi per il passaggio dei parametri

I due modi principali per passare parametri ad una procedura sono:

- passaggio per valore
- passaggio per riferimento

I parametri di una procedura possono essere dati od indirizzi. Se un parametro è un dato si dice che viene "**passato per valore**" (by value). Se un parametro è un indirizzo si dice che viene "**passato per riferimento**" (by reference) o "**per indirizzo**". Esistono altri modi di passaggio dei parametri, ma sono poco comuni.

Passaggio per valore

Il passaggio per valore è il più sicuro ed affidabile. Quando si passa un parametro per valore esso viene copiato prima di essere trasferito alla procedura. La procedura non ha un riferimento ad una locazione di memoria ma solo un numero da elaborare. Per questo non potrà in alcun modo fare danni alle variabili del programma principale.
Esempio:

```
MOV AX, [A]  ; passa il valore del parametro di ingresso A
             ; attraverso il registro AX alla procedura P
CALL P       ; chiamata della procedura
MOV [B], AX  ; al ritorno memorizza in B il valore del parametro
             ; di uscita, che la procedura ha passato in AX
```

Passaggio per valore

Il passaggio per indirizzo è più "pericoloso" di quello per valore. Utilizzando il passaggio per riferimento l'accesso ai valori dei parametri è "indiretto": la procedura dispone degli indirizzi dei parametri e con quegli indirizzi si procura i valori da elaborare con un ulteriore accesso alla memoria.
La procedura sa dov'è in memoria la variabile che è usata come parametro, perché le viene passato il suo indirizzo, con il quale può modificarla.
La modifica del parametro spesso è un effetto voluto, ed è un buon modo per passare strutture dati complesse, come per esempio gli array, che sarebbe "costoso" copiare prima del passaggio.
Esempio:

```
MOV SI, OFFSET A ; passa l'indirizzo (parte di offset) del parametro
             ; di ingresso A, attraverso il registro SI alla procedura P1
CALL P1      ; chiamata della procedura
; al ritorno non deve memorizzare nulla. Ci ha già pensato
; la procedura P1, che ha scritto ad OFFSET A
```

Può accadere però che la modifica dei parametri passati sia un **effetto collaterale**, non desiderato e potenzialmente pericoloso!
Per questo è sempre consigliabile usare, quando se ne ha l'opzione, il passaggio per valore e di usare il passaggio per indirizzo solo se non se ne può fare a meno e sapendo bene quello che si fa.
Per un esempio di un programma con effetti collaterali che provocano errori, vedere il programma in fondo a questo stesso capitolo, a pagina 20, paragrafo "Salvataggio del contesto" (si noti peraltro che l'effetto collaterale descritto in quell'esempio riguarda la modifica di registri, non di variabili in memoria passate per indirizzo!).

Mezzi per il passaggio dei parametri
A basso livello i parametri di una procedura si possono passare attraverso tre mezzi: i registri, la memoria e lo stack.
- passaggio attraverso i registri
- passaggio attraverso la memoria
- passaggio attraverso lo stack

Passaggio attraverso i registri
Il passaggio attraverso i registri è il più efficiente ed il più usato nei casi in cui sia il programma principale che la procedura sono scritti in Assembly.
Il parametro viene copiato dal programma chiamante in un registro, prima di fare la chiamata. La procedura "sa" in quale registro è il parametro e lo usa.
Ma se i registri sono pochi ed i parametri sono molti il passaggio con i registri non può essere più usato, perché i registri "finiscono". Per esempio se una procedura per 8086 ha parametri tanto lunghi e numerosi da occupare più di 14 Byte è impossibile passarli attraverso i registri, dato che AX, BX, CX, DX, SI, DI e BP contengono appunto 14 Byte.
L'uso di questo mezzo per il passaggio dei parametri non è quindi generalizzabile a tutti i casi. Per questo non viene usato dai linguaggi ad alto livello, che non possono avere limitazioni significative sul numero e/o sulla dimensione dei parametri da usare.
Entrambi gli esempi del paragrafo precedente passano i parametri nei registri; un altro esempio è il seguente:

```
MOV AH, 01   ; passaggio attraverso il registro AH
             ; della richiesta al DOS del servizio n.1
             ; (input da tastiera)
INT 21h      ; questo è un modo un po' più sofisticato di
             ; chiamare una procedura, che sarà illustrato
             ; nel seguito
; al ritorno di INT 21h la procedura restituisce in AL come parametro
; di uscita il codice ASCII del carattere che è stato battuto sulla
; tastiera
CMP AL, 27   ; guardo se è stato premuto il tasto Esc (ASCCI = 27)
JE Fine      ; se sì termino il programma
..
```

Passaggio attraverso la memoria
Quando si passano i parametri attraverso la memoria la procedura li cerca in specifiche locazioni, dove sono stati scritti dal programma principale.
Il procedimento è un po' più macchinoso e meno efficiente del passaggio attraverso i registri, dato che l'accesso alla memoria è più lento dell'accesso ai registri. Peraltro in alcuni casi è l'unico che si può usare (per esempio nelle procedure d'interruzione, che vedremo nel seguito).
L'esempio che segue è la versione che passa per la memoria del primo esempio del precedente paragrafo:

```
; copiatura del valore di A in memoria, in ParametroIngresso:
MOV AX, [A]  ; prima per AX perché no si possono fare accessi
             ; a due locazioni diverse
MOV [ParametroIngresso], AX
CALL P       ; chiamata della procedura
; al ritorno memorizza in B il valore del parametro di uscita
; che la procedura ha passato in memoria in ParametroUscita:
MOV AX, [ParametroUscita]
MOV [B], AX
```

Passaggio attraverso lo stack
Il passaggio attraverso lo stack è il più generale e flessibile, perciò, anche se è meno efficiente del passaggio nei registri, è il metodo usato dai compilatori di linguaggi ad alto livello. Questi linguaggi non hanno limiti nel numero di parametri che una procedura può ammettere, per cui l'uso dello stack è quasi obbligato.
Lo spazio nello stack può essere usato "a piacimento", a patto di non esaurirlo, e liberato facilmente quando i parametri non servono più.
Il procedimento per l'uso dello stack nel passaggio dei parametri è il seguente:
- il programma chiamante memorizza nello stack i parametri
- il programma chiamante esegue la chiamata a procedura
- la procedura recupera i parametri dallo stack, li utilizza e li elabora
- al momento del ritorno viene effettuata la "pulizia" dello stack, togliendone i parametri, ora non più utili.

Il problema della pulizia dello stack si risolve in due modi alternativi:
- il programma chiamante pulisce lo stack appena dopo il ritorno della procedura
- la procedura pulisce lo stack appena prima di tornare con la RET.

Quale che sia la strategia adottata, la procedura ed il programma chiamante dovranno comportarsi in modo coerente, pena gravi malfunzionamenti del programma; se è la procedura che deve pulire lo stack il programma chiamante non dovrà far nulla al riguardo, e viceversa.
L'esempio che segue è la versione che passa per lo stack dell'esempio precedente:

```
PUSH [A]     ; salvataggio del valore nello stack
CALL P       ; chiamata della procedura
POP [B]      ; ripristino del valore di ritorno dallo stack
; N.B. i valori di ritorno sono passati attraverso lo stack
; solo raramente, ed in casi particolari!
```

Per esempi più completi che riguardano il passaggio attraverso lo stack vedere "Esempi di passaggio attraverso lo stack con l'8086".
Come avremo modo di vedere nel seguito, i linguaggi di alto livello (C, Pascal, BASIC ..) passano i parametri in ingresso alle loro procedure utilizzando lo stack, mentre usano i registri per passare i parametri di ritorno.

1.2.2 Esempi completi

Vediamo ora alcuni esempi completamente sviluppati. Questi programmi funzionano con un 8086, ma hanno una valenza più generale, dato che ogni CPU si comporta nello stesso modo per quel che riguarda i salti a procedure.
In questi esempi risolveremo sempre lo stesso problema, con una procedura, passando i parametri in modi e con mezzi diversi. Vedremo anche come deve cambiare la procedura per poter accettare i parametri in quei mezzi e con quei modi.

Problema
Supponiamo di dover realizzare il calcolo della seguente espressione:

$$B = ((A + B)^2 + C)^2$$

Nella quale A, B e C sono numeri interi di 16 bit con segno. Supponiamo anche che i loro valori siano piccoli, in modo che il calcolo dell'espressione non vada mai in overflow.
Per il calcolo della precedente espressione può essere interessante realizzare una procedura. Vediamo quali passi compiere per progettarla e realizzarla.

Definizione della funzione della procedura
Nell'espressione si possono individuare due funzioni comuni, che realizzano l'espressione $(x + y)^2$.
Il calcolo di questa espressione può essere isolato in una procedura, che viene eseguita due volte per ottenere il risultato finale.

Definizione dei parametri
La procedura che andiamo a progettare avrà come parametri in ingresso i valori x e y dell'espressione precedente, mentre il risultato dell'espressione sarà il parametro d'uscita.
Il programma principale chiamerà la procedura due volte; la prima volta fornirà come parametri A e B, ottenendo $(A + B)^2$, la seconda volta passerà $(A + B)^2$ e C, ottenendo $((A + B)^2 + C)^2$.
Veniamo ora alla realizzazione pratica, presentando, per ogni modo e del mezzo con cui sono passati i parametri, una versione diversa.

Passaggio per valore
In questo primo esempio passeremo i parametri attraverso registri.

Codice del programma principale

```
..
;prima di chiamare la procedura copio il valore dei parametri in due registri:
MOV AX, [A]  ; primo parametro (x)
MOV BX, [B]  ; secondo parametro (y)
; chiamo la procedura per la prima volta
CALL SommaQuadratica
; supponiamo ora che quando la procedura ritorna lasci il suo risultato in AX
; (poi realizzeremo la procedura in modo che faccia proprio così)
; perciò ora in AX c'è (A + B)^2, che è già pronto come primo parametro
; per la seconda chiamata; l'altro parametro deve essere C:
MOV BX, [C]
; ora si può chiamare le procedura per la seconda volta:
CALL SommaQuadratica
; al ritorno da questa seconda chiamata in AX c'è ((A + B)^2 + C)^2,
; per cui lo devo solo salvare in B:
MOV [B], AX      ; B <- ((A + B)^2 + C)^2
..
```

Codice della procedura

SommaQuadratica:

```
    ; la procedura usa i numeri contenuti in AX e BX (passaggio per valore),
    ; li somma e poi moltiplica per due, sommando AX con AX.
```

```
; si possono subito sommare i due parametri:
ADD AX, BX       ; AX <- x + y
; (non controllo l'overflow perché so che i numeri sono sempre abbastanza piccoli
;  da non farlo mai scattare)
; moltiplico per se stesso:
IMUL AX          ; AX <- (x + y)²
; (il risultato sta tutto in AX (perciò DX = 0) per la stessa ragione di prima)
; ora in AX c'è (x + y)²  => posso ritornare:
RET
```

Passaggio per indirizzo

Riscriviamo il programma precedente, usando ancora i registri come mezzo per il trasferimento dei parametri, ma passando gli indirizzi delle variabili invece che il loro valore corrente[2].

Codice del programma principale

Scrivendo il programma principale dovremo fare alcune assunzioni sul comportamento della procedura, che andranno rispettate nel momento in cui la realizzeremo.

```
..
; prima di chiamare la procedura passo l'indirizzo (offset 8086) dei parametri
; nei due registri SI e DI:
MOV SI, offset A ; uso un registro puntatore, ottimo per contenere un indirizzo;
                 ; questo semplificherà il codice della procedura
MOV DI, offset B ; secondo parametro (y)
CALL SommaQuadratica2
; supponiamo che la procedura scriva il suo risultato in memoria
; all'indirizzo puntato da DI (Destination Index .. :-))
; ciò significa che ora B <- (A + B)²
; preparo la seconda chiamata:
MOV SI, offset C
; ora si può chiamare le procedura per la seconda volta:
CALL SommaQuadratica2
; la procedura salva il suo risultato in memoria all'indirizzo del secondo operando,
; per cui ora in B c'è già ((A + B)² + C)²
..
```

Codice della procedura

La procedura prende gli indirizzi degli operandi, ne recupera in memoria i valori, calcola l'espressione e scrive il risultato in memoria all'indirizzo del secondo parametro.

SommaQuadratica2:

```
; si possono subito sommare i due parametri:
MOV AX,  [DI]        ; AX <- y
MOV BX,  [SI]        ; BX <- x
ADD AX, BX           ; AX <- x + y
IMUL AX              ; AX <- (x + y)²
MOV [DI], AX             ; y  <- (x + y)²
RET
```

Passaggio attraverso la memoria

Facciamo un'altra variazione del solito programma, passando i parametri attraverso la memoria. Il modo di passaggio è per valore, per cui questo esempio va raffrontato con l'esempio precedente mostrato in "Passaggio per valore".

Codice del programma principale

Nella parte del programma in cui si definiscono i dati allochiamo le locazioni di memoria attraverso le quali passare i parametri in ingresso ed in uscita, poi nella parte di codice facciamo il passaggio e le chiamate alle procedure:

```
..

X DW ?           ; per un parametro d'ingresso
Y DW ?           ; per un parametro d'ingresso
Risultato DW ?   ; per il parametro d'uscita
..

; prima di chiamare la procedura passo l'indirizzo (offset 8086) dei parametri
; nei due registri SI e DI:
; passo il valore di A nella locazione X:
MOV AX, [A]
MOV [X], AX
```

2 Supponiamo anche che le variabili A, B e C stiano nello stesso segmento di memoria e che DS punti all'inizio di quel segmento. Se è così non è necessario passare alla procedura il valore del registro di segmento, ma solo la parte di offset dell'indirizzo dei parametri.

```
    ; passo il valore di B nella locazione Y:
    MOV AX, [B]
    MOV [Y], AX
    CALL SommaQuadratica3
    ; la procedura scriva il suo risultato in memoria
    ; all'indirizzo di Risultato, per cui quel valore ora
    ; deve andare in [X]:
    MOV AX, [Risultato]
    MOV [X], AX      ; Y <- (A + B)^2
    ; passo C come altro parametro d'ingresso
    MOV AX, [C]
    MOV [Y], AX
    CALL SommaQuadratica3

    ; ora in Risultato c'è ((A + B)^2 + C)^2; lo devo copiare in B:
    MOV AX, [Risultato]
    MOV [B], AX      ; B <- ((A + B)^2 + C)^2
```

Codice della procedura
La procedura prende gli indirizzi degli operandi, ne recupera in memoria i valori, calcola l'espressione e scrive il risultato in memoria all'indirizzo del secondo parametro.

SommaQuadratica2:

```
    ; copio gli operandi in registri, per le elaborazioni: possono subito sommare i due
parametri:
    MOV AX, [X]          ; AX <- x
    MOV BX, [Y]          ; BX <- y
    ADD AX, BX           ; AX <- x + y
    IMUL AX              ; AX <- (x + y)^2
    ; il calcolo dell'espressione viene copiato in memoria, all'indirizzo di Risultato:
    MOV [Risultato], AX  ; Risultato <- (x + y)^2
    RET
```

Passaggio attraverso lo stack
Si evita di fornire esempi di passaggi attraverso lo stack a questo punto. Ciò perché un esempio fatto con un 8086 richiede l'introduzione preventiva di alcuni argomenti specifici di quella CPU.
Perciò gli esempi relativi al passaggio nello stack sono riportati più avanti, al paragrafo "Esempi di passaggio attraverso lo stack con l'8086".

1.2.3 Scrivere buone procedure

Si può senz'altro dire che una buona procedura conferisce al programma un più alto livello d'astrazione, dato che nasconde i dettagli non necessari e rende disponibili comandi "più potenti" (le procedure stesse sono "comandi più potenti"!).
Per progettare buone procedure occorre andare alla ricerca delle parti dell'algoritmo che hanno delle similarità, poi mettere in evidenza quali sono le differenze che si trovano in quelle parti simili.
Poi bisognerà concentrare nel codice delle procedure le somiglianze, mentre le differenze costituiranno i parametri per quelle procedure.
Uno dei maggiori vantaggi nell'uso delle procedure è il fatto che possono essere utilizzate in contesti diversi da quello nel quale sono state ideate. Per aumentare la probabilità che una procedura possa essere utile in futuro bisogna progettarla in modo che essa esegua qualcosa di "compiuto", che si distingue chiaramente dal resto del programma.
In questo modo si ha anche il vantaggio non trascurabile che il programma diventa più comprensibile, autodocumentato e facile da realizzare. Il problema generale viene infatti suddiviso in piccoli problemi particolari, "finiti", che si risolvono "uno alla volta" e sono più comprensibili e controllabili.

1.3 Procedure nell'8086

Le istruzioni dell'Assembly 8086 che si usano per le procedure sono le stesse illustrate fino ad ora (CALL e RET).
Le particolarità tipiche dell'8086 riguardano, naturalmente, la segmentazione. Dato che esistono le jump FAR e quelle NEAR, e dato che una CALL è una jump con ritorno, nell'8086 ci saranno anche due tipi di CALL: quelle NEAR e quelle FAR:
Come abbiamo già detto il Program Counter dell'8086 è CS:IP. Sarà perciò CS:IP che deve essere memorizzato nello stack, quando si esegue la chiamata ad una procedura.
Se l'indirizzo della procedura è nello stesso segmento del punto dal quale essa viene chiamata, si può raggiungere la procedura cambiando solo IP, senza toccare CS. Il salto che si deve fare per raggiungere la procedura è quindi di tipo NEAR. In questo caso non è necessario memorizzare nello stack CS, dato che non viene modificato, la procedura è di tipo NEAR.

Quando si vuole saltare ad ogni possibile indirizzo della memoria bisogna memorizzare sia CS che IP, poi fare un salto che li modifichi entrambi. In questo caso la procedura è di tipo FAR.

La sintassi TASM per le procedure NEAR e FAR è la seguente:

```
CALL [NEAR PTR | FAR PTR] <LabelProcedura>
```

Spesso si omette la dicitura NEAR o FAR PTR, lasciando fare al compilatore, che sceglie secondo altri criteri che illustreremo fra poco. L'uso tipico della CALL è quindi il seguente:

```
CALL <LabelProcedura>
```

La chiamata a procedura NEAR fa le seguenti "istruzioni equivalenti":

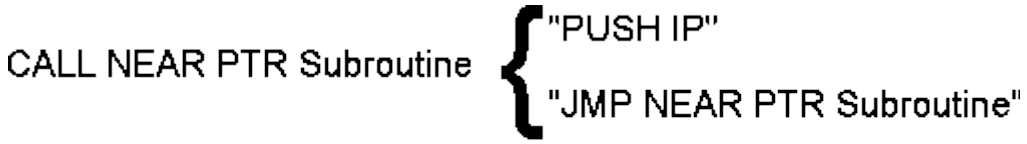

Figura 13: istruzioni equivalenti di una CALL NEAR

Analogamente, una CALL FAR:

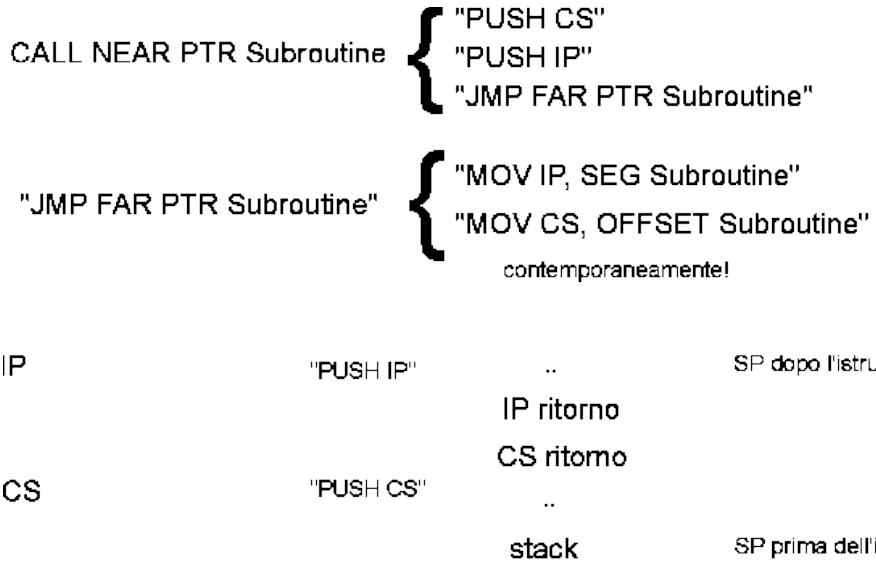

Figura 14: funzionamento di una CALL FAR

Si noti che l'ordine di scrittura nello stack è tale che all'indirizzo più basso finisce IP, che è la "parte bassa" dell'indirizzo segmentato. All'indirizzo più alto finisce la parte di segmento dell'indirizzo, cioè CS.

Le istruzioni di ritorno sono:

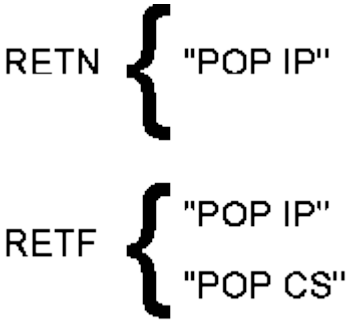

Figura 15: istruzioni equivalenti di RET, near e far

Che costituiscono una jump, near o far, all'indirizzo che attualmente è in cima allo stack.

Come per la CALL anche la RET viene usata abitualmente senza l'indicazione di NEAR o FAR:

```
RET
```

Non è detto che una procedura debba avere solo una RET, per quanto debba avere almeno una. E' cioè possibile fare tutte le RET che si ritiene opportuno. Ovviamente solo la prima RET incontrata dal programma sarà eseguita, dato che si tornerà indietro.

Per chiarire come la differenza fra CALL NEAR o FAR facciamo il seguente esempio:

```
    ..
Proced:
    .. qui c'è il codice della procedura ..

    ; in questo esempio compare prima il codice della procedura, poi
    ; quello del main. La cosa è del tutto lecita.

    RETF    ; ritorno FAR

    .. qui c'è il codice del main ..

    CALL FAR PTR Proced  ; OK (la procedura è FAR, come si vede dalla RET)
Ritorno:
    ..
    CALL Proced          ; COMPILA ed esegue, ma è SBAGLIATO (default = NEAR)
AltroRitorno:
    ..
```

In questo esempio la prima chiamata funziona regolarmente, tornando all'indirizzo della label Ritorno; la seconda salta mettendo in IP l'offset di AltroRitorno, ma mettendo in CS un numero sconosciuto, che ha avuto la ventura di finire nello stack prima dell'offset di AltroRitorno. Il programma salta perciò ad un indirizzo sconosciuto, con le conseguenze che ci possiamo aspettare!
ra:

!!!! Da mettere !

Figura 16: ritorno avventuroso

Vediamo un altro esempio di un errore, in cui la procedura è NEAR:

```
    .. qui c'è il codice del main ..

    ; in questo esempio compare prima il codice del main, poi quello
    ; della procedura. La cosa è altrettanto lecita della precedente.

    ..
    CALL ProcedUno       ; 1 Giusto
RitornoUno:

    CALL FAR PTR Proced  ; 2 COMPILA ed esegue, ma è SBAGLIATO
AltroRitornoUno:
    ..

ProcedUno:
    .. qui c'è il codice della procedura ..

    RETF    ; ritorno FAR
```

L'istruzione indicata con 1 è giusta: sia chiamata che ritorno sono NEAR.
L'istruzione indicata con 2 è sbagliata, anche se il programma non si blocca subito e riesce a tornare al chiamante. Dato che la chiamata è FAR il programma chiamante mette due word nello stack, mentre la RET, che è NEAR, ne toglie solo una. All'esecuzione della RETF lo stack risulta dunque sbilanciato, perché sono stati inseriti due valori e ne è stato tolto solo uno. E' solo un questione di tempo perché la cosa si manifesti con risultati nefasti.
In Assembly 8086 esistono direttive che permettono di diminuire il rischio di scrivere procedure con istruzioni di chiamata e di ritorno "incompatibili". Esse permettono di dichiarare qual è la parte del sorgente che deve essere riservata alla procedura. Nel prossimo paragrafo illustriamo tale direttiva.

1.3.1 Procedure nelle CPU a 32 bit

Nelle CPU a 32 bit è stato esteso il funzionamento delle procedure in modo del tutto analogo al caso dei 16 bit. E' logico che, nel caso a 32 bit, venga salvato nello stack il valore di EIP, dunque un offset di 32 bit dell'indirizzo di ritorno. Il segistro di segmento salvato nello stack nelle procedure FAR rimane invece di 16 bit.
Infatti anche in questo caso esistono le procedure NEAR e quelle FAR, che salvano anche il valore del registro di segmento. Nel caso a 32 bit però, le procedure FAR non sono mai usate. Infatti tutti i Sistemi Operativi significativi per X86 utilizzano i segmenti nel modo "flat" (piatto) facendoli puntare tutti allo stesso indirizzo. Dunque non è interessante salvare il registro di segmento e le chiamate, in tutti i S.O. di interesse, sono tutte NEAR.

1.3.2 Direttive Assembly 8086 per le procedure

Per quanto non sia strettamente indispensabile, l'Assembly 8086 dispone di direttive che possono leggermente semplificare la vita al programmatore che debba scrivere procedure.

Direttive PROC e ENDP
Usando queste direttive si può "confinare" il codice della procedura fra due "parentesi logiche" che lo racchiudono.
La sintassi è la seguente:

```
<LabelProcedura> PROC [NEAR | FAR]

    .. qui "dentro" ci va il codice della procedura ..

<LabelProcedura> ENDP
```

Usando queste direttive il compilatore può sapere qual è la parte del sorgente che contiene il codice della procedura.
Indicando nella direttiva PROC se la procedura è NEAR o FAR il compilatore potrà controllare che tutte le chiamate siano dello stesso tipo, evitando errori come quelli illustrati precedentemente.
Inoltre se le chiamate non hanno indicazione se NEAR o FAR, il compilatore, sapendo che la procedura è NEAR o FAR farà automaticamente chiamate e ritorni dello stesso tipo; se la procedura è definita FAR compilerà CALL FAR PTR e RETF, se è NEAR saranno CALL NEAR PTR e RET.
Se NEAR o FAR viene omesso la procedura viene compilata come NEAR.

Esempi:

```
pFAR PROC FAR
    ..
    RET ; = RETF, perché PROC dice che pFAR è FAR
pFAR ENDP

pNEAR PROC NEAR
    ..
    RET ; = RETN
pNEAR ENDP

    .. main ..

    CALL FAR PTR pFAR    ; OK
    CALL NEAR PTR pNEAR  ; NON PASSA! Il compilatore genera errore!
    CALL pFAR            ; Viene compilata come FAR
    CALL pNEAR           ; Viene compilata come NEAR
    CALL NEAR PTR pFAR   ; non viene compilata ed il compilatore dà errore!
```

Nella maggior parte dei casi non si indicheranno NEAR o FAR, né nelle chiamate, né nelle RET. Ci penserà la direttiva PROC a dire se la procedura deve essere NEAR o FAR. Il compilatore agirà di conseguenza.
Un'eccezione importante a questo modo di procedere si avrà quando si passeranno parametri alla procedura attraverso lo stack. Dato che in quel caso è decisivo sapere cosa c'è nello stack è altrettanto importante sapere se la procedura è NEAR o FAR. In questo caso lo scriveremo anche nella chiamata, così il compilatore ed il programmatore potranno controllare se tutto va bene.

Direttiva PARAMETER
Esiste una direttiva, che segue la direttiva PROC nella stessa riga, che permette di semplificarsi leggermente la vita quando alle procedure vengono passati molti parametri.
Usando la direttiva PARAMETER il compilatore genererà automaticamente il codice che fa passare il parametri attraverso lo stack, in base al tipo che viene indicato per ciascuna "variabile".
Dato che questa direttiva non è molto utile nei programmi più semplici ed oscura un po' il funzionamento del passaggio dei parametri, non la spiegheremo ulteriormente, rimandando i lettori che sono interessati all'argomento al manuale dell'Assembler che essi utilizzano.

1.3.3 Esempi di passaggio attraverso lo stack con l'8086

Consideriamo ancora il calcolo dell'espressione $B = ((A + B)^2 + C)^2$; in questo esempio passiamo i parametri per valore attraverso lo stack. Le soluzioni presentate in questo paragrafo saranno confrontabili con l'esempio del paragrafo "Passaggio per valore".

```
    ..
    ; (0) prima della chiamata copio nello stack il valore dei parametri:
    PUSH [A]         ; (1) primo parametro
    PUSH [B]         ; (2) secondo parametro
    CALL FAR PTR SommaQuadratica4 ; (3) chiamata
    ; è INDISPENSABILE sapere se la procedura è NEAR o FAR, perché nello
    ; stack, "sopra" ai parametri, finisce l'indirizzo di ritorno, che
    ; può consistere in una o due word!

    ; Suppongo che la procedura metta il risultato nello stack
    ; allora (A + B)^2 è già nello stack, devo aggiungere l'altro
    ; parametro, cioè C:
    PUSH [C]
    CALL FAR PTR SommaQuadratica4 ;  seconda chiamata
    ; ora nello stack c'è il risultato finale, lo recupero:
    POP AX       ; AX <- ((A + B)^2 + C)^2
    ..
```

A questo punto è necessario analizzare ciò che è contenuto nello stack.
Le due PUSH prima della CALL hanno messo nello stack i valori correnti di A e B, poi la chiamata di procedura ha aggiunto CS e IP dell'indirizzo di ritorno, dato che la procedura è FAR:. La Figura 17 fa riferimento ai commenti del precedente brano di programma.
Ora possiamo pensare a come scrivere il codice della procedura.
Il primo modo che viene in mente per recuperare dallo stack durante la procedura è leggere lo stack con delle POP. Notiamo subito che questo metodo, che illustriamo solo perché è semplice da capire, non è molto elegante né efficiente, non si dovrebbe utilizzare in programmi "veri".
Dopo aver letto i parametri dallo stack con una serie di POP c'è il problema di "rimetterlo a posto" prima della RET, dato che questa istruzione si aspetta di trovare alla posizione corrente dello stack l'indirizzo di ritorno.
La seguente procedura realizza il metodo appena descritto, anche il parametro d'uscita (la somma quadratica) viene passata nello stack:

```
SommaQuadratica4 PROC FAR
    ; Somma Quadratica con parametri passati
    ; per valore nello stack, sia in ingresso che in uscita
    POP CX   ; (4) CX <- IP di ritorno
    POP DX   ; (5) DX <- CS di ritorno
    POP AX   ; (6) AX <- valore del SECONDO operando
    POP BX   ; (7) BX <- valore del PRIMO operando

    ADD AX, BX   ; AX <- x + y
    IMUL AX          ; AX <- (x + y)^2
    ; risistemazione dello stack per il ritorno:
    PUSH AX  ; (8) parametro di uscita
             ; (il risultato della procedura)
    PUSH DX  ; (9) CS di ritorno, per la RET
    PUSH CX  ; (10) IP di ritorno, per la RET

    RET      ; (11)
SommaQuadratica4 ENDP
```

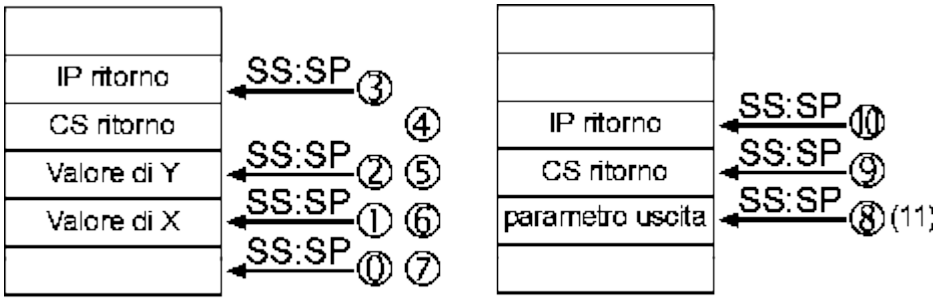

Figura 17: contenuto dello stack durante la chiamata a SommaQuadratica4

Il procedimento testè illustrato è macchinoso e non efficiente, soprattutto se pensiamo che le procedure potrebbero avere molti parametri! Se si hanno molti parametri, non si possono togliere tutti insieme dallo stack, perché i registri non bastano.
Per cui si inventa un'altra tecnica, molto migliore, che è quella che viene usata anche dai compilatori dei linguaggi ad alto livello.
Lasciamo i parametri nello stack, senza toglierli con delle POP, ed andiamo a "rovistare" nello stack con un indirizzamento indiretto.
Se ci ricordiamo che il registro BP (Base Pointer) usa come segmento di default SS, invece di DS, usiamo BP come registro per l'indirizzamento e possiamo anche capire perché i progettisti dell'8086 abbiano fatto in modo che BP usasse SS invece di DS.
Teniamo presente che all'ingresso nella procedura la situazione dello stack è quella illustrata al punto (1) della Figura 18, per cui i parametri sono agli indirizzi SP + 4 e SP + 6. Ricordiamo infine che nel'8086 il registro SP non si può usare in indirizzamenti indiretti (è vietato fra le parentesi quadre).
Fatte queste premesse la seguente procedura ed i suoi commenti dovrebbero spiegarsi da soli:

```
SommaQuadratica4_1 PROC FAR ; !! versione 4_1, migliorata da 4!!
    ; Somma Quadratica con parametri passati per valore nello stack,
    ; sia in ingresso che in uscita
    MOV BP, SP   ; copio SP in BP per poterlo poi usare nel seguente
                 ; indirizzamento indiretto:
; (1)
    MOV BX, [BP + 4] ; secondo parametro (Y) in BX
    MOV AX, [BP + 6] ; primo parametro (X) in AX
    ADD AX, BX   ; AX <- x + y
    IMUL AX          ; AX <- (x + y)²
    MOV [BP + 6], AX ; scrittura del risultato nello stack

    RET
SommaQuadratica4_1 ENDP
```

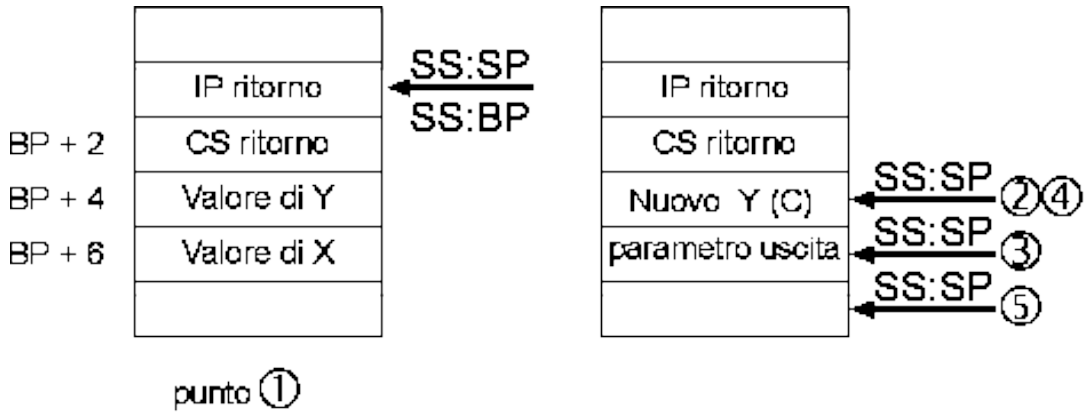

Figura 18: contenuto dello stack durante le chiamate a SommaQuadratica4_1

Come si può ben vedere questo procedimento è molto più snello del precedente, visto che lo stack non si modifica se non per scriverci dentro il risultato finale.
C'è però una cosa importante da notare. Dato che all'interno della procedura non ci sono PUSH o POP alla sua uscita l'occupazione dello stack rimane identica a prima (punto (2) di Figura 18), per cui il programma principale deve essere cambiato. Infatti la versione precedente di SommaQuadratica4 aveva lo stack già pronto con il risultato, mentre questa versione c'è ancora il valore del secondo parametro, che la procedura non ha tolto.
Per questo il programma principale deve essere modificato così:

```
    ..
    ; come prima:
    PUSH [A]
    PUSH [B]
    CALL FAR PTR SommaQuadratica4
;(2)
    ; la procedura mette il risultato nello stack, nel posto dove
    ; era stato passato A, quando torna il valore di B è
    ; ancora nello stack, allora lo deve togliere:
    ADD SP, 2    ; questa "toglie" dallo stack il vecchio valore di BP,
                 ; che non serve più
;(3)
```

```
    ; ora nello stack c'è rimasto il risultato, aggiornato dalla procedura
    ; per cui devo metterci solo C:
    PUSH [C]
;(4)
    CALL FAR PTR SommaQuadratica4 ;  seconda chiamata
    ; ora nello stack c'è prima il vecchio C, poi il risultato finale
    ADD SP, 2    ; "toglie" il vecchio C dallo stack
    ' recupera il risultato finale:
    POP AX       ; B <- ((A + B)² + C)²
;(5)
    ..
```

1.4 Tecniche di programmazione con le procedure

Una buona procedura è corta; se possibile sta in una singola videata, al massimo in poche.
In questo modo si tiene facilmente tutta sott'occhio in fase di sviluppo ed è più facile affiancarla ad altre parti del programma in un ambiente di sviluppo multifinestre.
Le procedure devono aiutare a strutturare il programma, cioè a dividerlo in blocchi "autonomi". Se un blocco contiene tutto il programma non è di molta utilità :-(

Usare il lavoro degli altri

Utilizzando le procedure è possibile integrare porzioni di programma scritti in tempi, luoghi e da persone diverse.
Questo permette di dividere fra diverse persone il lavoro su un programma complesso, oppure di acquistare librerie di procedure già provate e compilate, in modo da non dover scrivere anche le parti di codice più banali o troppo specializzate.

Lavorare per gli altri

Scrivere buone procedure è un esercizio di disciplina, che non deve servire solo per chi le sviluppa, ma soprattutto per chi le usa.
Per l'utente di una procedura l'unica cosa che conta è la fiducia. In particolare deve avere:
- fiducia che la procedura funzioni
- fiducia che la potrà usare correttamente

Per il primo di questi fattori vale solo l'abilità tecnica del programmatore.
Per il secondo vale la documentazione della procedura.
Procedure anche ottime non verranno mai usate se non è assolutamente chiaro cosa si deve fare per usarle ed a cosa si va incontro quando le si usano.
Per una buona documentazione è necessario darsi uno "standard" e rispettarlo, come illustrato nel successivo capitolo "Sviluppo".

Documentare i parametri

In questo paragrafo illustriamo una tecnica per documentare bene i parametri delle procedure, quando si fa uso di passaggi attraverso i registri.
Consideriamo ancora l'esempio, ormai abusato, del calcolo dell'espressione $(x + y)^2$, nella sua versione migliorata con passaggi nello stack (paragrafo 1.3.3, pagina 1):

```
..
; definiamo i displacement dei parametri nello stack:
x EQU BP + 6
y EQU BP + 4
..
; nel codice della procedura accediamo allo stack utilizzando il "nome"
; del parametro che abbiamo appena definito:

SommaQuadratica4 PROC FAR ; !! versione migliorata con migliore documentazione!!
    ; Somma Quadratica con parametri passati per valore nello stack,
    ; sia in ingresso che in uscita
    MOV BP, SP   ; copio SP in BP per poterlo poi usare nel seguente
                 ; indirizzamento indiretto:
    MOV AX, [x]  ; primo parametro in AX
    ADD AX, [y]  ; AX <- x + y
    IMUL AX          ; AX <- (x + y)²
    MOV [x], AX  ; scrittura del risultato nello stack

    RET
SommaQuadratica4 ENDP
```

In questo programma funzionalmente non cambia nulla, il programma compilato è identico alla versione precedente, ma ne aumenta considerevolmente la leggibilità.

Passaggio di un indirizzo segmentato
Non solo gli indirizzi dei salti possono essere NEAR o FAR, ma anche i dati, che possono essere in segmenti diversi.
Se i dati sono in segmenti diversi ed è necessario un passaggio per indirizzo, bisogna passare sia l'offset che il segmento.
All'interno della procedura si accederà ai dati utilizzando non solo l'offset, ma l'intero indirizzo segmentato:

```
OffsetX EQU BP + 10
SegX EQU BP + 12
OffsetY EQU BP + 6
SegY EQU BP + 8
    ..
    ; nel main, passaggio per indirizzo FAR nello stack:
    ; parametro x
    PUSH SEG A
    ; ^ solo 286>, perché è una PUSH con operando in immediato
    PUSH OFFSET A    ; solo 286>
    ; parametro y
    PUSH SEG B       ; solo 286>
    PUSH OFFSET B    ; solo 286>
    CALL SommaQuadratica5

    ..  ; !! non si dettaglia il resto del main !!
        ; !! ormai si saprà come fare :-)       !!
    ..
SommaQuadratica5 PROC FAR
    ; Somma Quadratica con parametri in ingresso passati per indirizzo FAR
    ; nello stack.
    ; Il parametro di uscita (somma quadratica) è scritto
    ; direttamente in memoria all'indirizzo parametro x
    ; **** salvataggio di contesto *****
    PUSH BP      ; salvo il valore di BP del programma chiamante
    MOV BP, SP   ; copio SP in BP per poterlo poi usare nell'accesso allo stack
; (1)
    PUSH DS      ; salvo DS ed ES perché qui li modifico
    PUSH ES
    ; **** FINE salvataggio di contesto *****
    ; caricamento indirizzo parametro X, in DS:SI:
    MOV SI, [OffsetX]
    MOV AX, [SegX]
    MOV DS, AX
    ; caricamento indirizzo parametro Y, in ES:DI:
    MOV DI, [OffsetY]
    MOV AX, [SegY]
    MOV ES, AX
    ; elaborazione:
    MOV AX, [SI]     ; AX <- x
    ADD AX, ES:[DI]  ; AX <- x + y
    IMUL AX              ; AX <- (x + y)2
    MOV [SI], AX ; scrittura del risultato direttamente
                 ; nel parametro X
    ; **** ripristino di contesto *****
    POP ES       ; ripristino BP, DS ed ES del programma chiamante
    POP DS
    POP BP
    ; **** FINE ripristino di contesto *****
    RET
SommaQuadratica5 ENDP
```

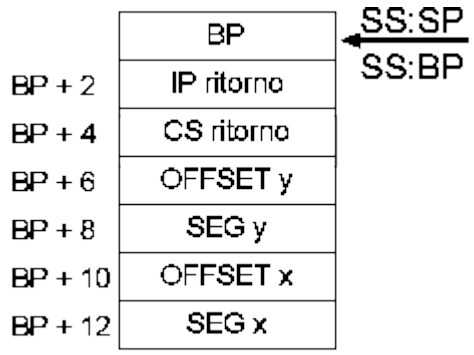

Figura 19: situazione dello stack nel punto (1) della procedura SommaQuadratica5

I parametri sono stati passati nello stack in modo che agli indirizzi bassi dello stack stessero le parti di offset, a quelli alti i segmenti. In questo modo si possono caricare gli indirizzi con LDS e LES, vediamo come:
Le istruzioni:

```
    MOV SI, [OffsetX]
    MOV AX, [SegX]
    MOV DS, AX
```

Possono essere sostituite con:

```
    LDS SI, [OffsetX]
```

e:

```
    MOV DI, [OffsetY]
    MOV AX, [SegY]
    MOV ES, AX
```

con:

```
    LES DI, [OffsetX]
```

Passaggio di parametri booleani

Spesso è necessario che una procedura comunichi al programma principale un valore vero o falso, come una variabile di tipo boolean in Pascal o Visual BASIC.
Se le chiamate sono da Assembly a Assembly questo passaggio viene di solito realizzato attraverso uno dei flag.
Spesso per sistemare il valore del flag che interessa non si deve fare niente, perché ci pensano le istruzioni aritmetiche utilizzate (caso (1) dell'esempio successivo).
In alcuni casi invece bisogna assegnare un preciso valore ad uno dei flag, perché non c'è un'istruzione aritmetica che lo fa "automaticamente" (caso (2) dell'esempio successivo). Per questo scopo si può usare ST o CL, per i flag che ammettono tale istruzione; per esempio STC mette a uno il flag di carry, mentre CLC lo mette OFF.
Nelle CPU X86 per alcuni flag non esiste l'istruzione ST, né la CL, per cui bisogna ricorrere a dei trucchi, come nel punto (2) del seguente esempio:

```
Trovato PROC
    ..
; Parametri in uscita: flag di zero (ZF) = ON se il valore è stato trovato
    ..
    CMP [LaStringaCheStoScansionando], "A"
    ; (1)
    JE lHoTrovato    ; se il carattere corrente di LaStringaCheStoScansionando è "A"
                ; il flag di zero è ON
    ..
    ; si giunge a questo punto solo se NON si è trovato ciò che si cercava.
    ; (2) : ora voglio che il flag di zero sia OFF:
    OR AL, 1     ; questa istruzione dà senz'altro risultato diverso da zero,
              ; per cui ZF = OFF
; ATTENZIONE, l'istruzione precedente MODIFICA il bit meno significativo di AL,
; per cui AL non deve servire in seguito!
    RET

lHoTrovato:
    ; il flag di zero è ON (si salta qui solo da JE lHoTrovato)
    RET
Trovato ENDP
```

I linguaggi ad alto livello non possono manipolare il flag della CPU, per cui se si deve realizzare il passaggio di parametri booleani con programmi scritti in linguaggi ad alto livello bisogna conoscere in dettaglio come quel linguaggio rappresenta internamente le variabili booleane.
Tutti i linguaggi ad alto livello usano almeno un Byte per memorizzare le variabili booleane. In quel Byte rappresentano il valore falso sempre con un zero, mentre il valore vero può essere espresso da 1 oppure da –1, dipende da come funziona il programma traduttore.
Il valore –1 è più comune, perché è il risultato dell'istruzione NOT eseguita su 0, che, come detto, è il valore falso.
Esempio:

```
    ..
    Bool DB (?)
    ..
    MOV [Bool], 0    ; Bool <- false
```

```
    NOT [Bool]       ; Bool <- NOT false ora Bool = -1 = 0FFh
    ..
```

Riguardo a questo problema si farà ulteriore riferimento nel prossimo volume di questo testo.

1.4.1 Uso dello stack all'interno delle procedure

All'interno delle procedure si può usare lo stack, naturalmente bisognerà farlo con grande accortezza visto che nello stack sono presenti gli indirizzi di ritorno delle procedure.
Molti tipi di errore software possono essere in qualche modo "recuperabili", se il programma lo prevede. Al verificarsi di un errore, come per esempio una divisione per zero, il software può provare a mettervi rimedio ed a proseguire, o quantomeno può avvertire l'utente dell'errore mantenendo il sistema sotto controllo.

Gli errori nella programmazione dello stack sono quasi sempre fatali perché viene completamente perso il controllo del programma.

Nel caso delle procedure il risultato tipico di un errore di bilanciamento dello stack è che il programma salta ad una locazione "a caso" per cui il programma, e spesso tutto il sistema, va necessariamente in crash.
Vediamo con un semplice esempio quanto possa essere pericoloso sbagliare nell'utilizzazione dello stack all'interno di una procedura:

```
..
  CALL NonFunziona
  ..

NonFunziona PROC
  PUSH AX
  PUSH BX

  .. qui non è scritto il codice della procedura ..

  ; alla fine della baldoria, facciamo pulizia:
  POP CX         ; OOPS! mi sono "dimenticato" di togliere una word dallo stack!
  RET        ; DISASTRO! La RET salta nel vuoto all'indietro: l'offset della
             ; locazione ove verrà fatta la prossima fase di fetch non
             ; è il valore di IP di ritorno, ma il valore che aveva AX
             ; al momento dell'ingresso nella procedura!
NonFunziona ENDP
```

Salvataggio del contesto
Quando si scrive una procedura bisogna sempre usare almeno un registro, per i trasferimenti ed i calcoli. Quel registro sarà inevitabilmente sovrascritto, per cui il suo valore vecchio sarà cancellato.
Se la procedura non prende "precauzioni", quando essa ritorna, il valore di quel registro è cambiato.
In questo caso si dice che la procedura ha l'"**effetto collaterale**" (side effect) di modificare un registro. La procedura, oltre a realizzare la funzione per cui è stata scritta, esegue anche una modifica indesiderata ad un registro.
Per lavorare con procedure che non conosciamo è necessario che il programma chiamante e/o la procedura provvedano a memorizzare il valore dei registri che sono modificati ed il cui valore non si può perdere.
Questa memorizzazione viene detta "**salvataggio di contesto**" (context saving).
Il salvataggio di contesto è la memorizzazione di tutti i valori che potrebbero essere modificati da una procedura e che non si vuole perdere.

Vediamo un esempio di codice SBAGLIATO, perché non tiene conto di effetti collaterali. Il codice è ben documentato, usando una convenzione "standard", ed è scritto da due programmatori diversi. Il codice ha diversi errori, ma tutti ben nascosti, difficili da trovare leggendo il codice. Molti lettori per scoprirli dovrebbero usare il debugger (prima di proseguire nel testo provare il file SBAGLIAT.ASM del CD ROM1).

```
;*****************************************************************************
; Nome del programma: Conta le lettere            Nome del file: CONTlett.ASM
; Versione: 0.9.0                                 Data rev.: XX.XX.XXXX
; ------------------------------------------
; Funzionalità del programma:
; Scrive nel vettore di word "Risultati" il numero di occorrenze
; di ogni carattere dell'alfabeto in una stringa. Trova anche il massimo
; dei conteggi sulle lettere e lo memorizza in Massimo
; Al primo posto (Risultati[0]) finisce il numero di "a" nella stringa,
; all'ultimo posto Risultati[25] il numero di "z" contenute
; nella stringa)
; La stringa è lunga sempre e solo 1 kByte
; ------------------------------------------
; Storia delle revisioni (in ordine cronologico inverso):
;  Revisione: 0.1.0                               Data rev.: XX.XX.XXXX
;    Scritto da: MONTI Data: yy.yy.yyyy
;*****************************************************************************

.. dichiarazioni ed istruzioni di inizio programma, qui sono omesse ..
    ; una stringa di un k:
    Stringa DB "24 car.poi 500 volte LH ", 500 DUP ("LH")
    ; un vettore di risultati:
    Risultati DW 25 DUP (?)
    ; il massimo numero di conteggi:
    Massimo DW (?)

    MOV SI, OFFSET Risultati     ; indirizzo da dove cominciare a scrivere
    MOV AL, "a"  ; prima lettera da cercare
    MOV CX, 25   ; inizializzo il conteggio dei caratteri da cercare
    MOV DX, 0    ; registro temporaneo in cui tengo il massimo
CicloAlfabeto:   ; ciclo da 25 giri, come le lettere dell'alfabeto
    MOV BX, OFFSET Stringa   ; indirizzo da dove cominciare a leggere
    ; conta nella stringa il cui indirizzo è passato in BX
    ; il numero di occorrenze del carattere passato in AL
    Call ContaCaratteri
    ; restituisce in AX il conteggio cercato
    ; memorizza il conteggio al posto giusto di Risultati
    MOV [SI], AX
    ; se il nuovo conteggio è maggiore del massimo corrente, diventa il massimo:
    CMP AX, DX
    JNA NonMaggiore  ; numero senza segno
    ; maggiore: nuovo massimo:
    MOV DX, AX
NonMaggiore:
    INC AL       ; Prossimo carattere per il confronto (es. da "a" a "b")
    ADD SI,2         ; Punta al prossimo elemento di Risultati
    LOOP CicloAlfabeto
    ; fine del programma principale
    MOV [Massimo], DX

    .. altre istruzioni che concludono il programma, qui omesse ..

    ; terminazione "DOS" del programma:
    MOV AH, 4Ch
    INT 21h
```

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ContaCaratteri PROC
; Scritta da: MARI                              Data ultima modifica: XX.XX.XXXX
;   ---------------------------------------------------------------------------
;   Descrizione:
;   Conta il numero di occorrenze del carattere passato in AL
;   nella stringa di 1 kByte il cui offset è passato in BX
;   ---------------------------------------------------------------------------
;   Parametri passati attraverso:
;     Ingressi: registri
;     Uscite  : registri
;
;     Ingressi: ------------------
;       AL: carattere da cercare e contare
;       BX: offset della stringa in cui cercare e contare
;     Uscite: --------------------
;       AX: conteggio dei caratteri trovati
;   ---------------------------------------------------------------------------
    MOV CX, 1024 ; la stringa è lunga 1 kByte
    XOR DX, DX   ; inizializza un contatore temporaneo
unkVolte:
    CMP [BX], AL
    JNE NonUguale
    INC DX       ; conta un altro carattere trovato
NonUguale:
    INC BX       ; prosegue nella stringa
    LOOP unkVolte
    MOV AX, DX   ; scrive il conteggio ottenuto nel parametro di uscita
    RET
ContaCaratteri ENDP
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Il programma precedente non funziona per diverse ragioni, molte da imputarsi a mancanza di comunicazione fra i due programmatori. Inoltre MONTI, che ha scritto il programma principale, ha commesso un errore da principiante.

Vediamo per primo l'errore più clamoroso. La documentazione della procedura specificava chiaramente che il registro AX era usato sia in ingresso che in uscita (AL in ingresso e AX in uscita). Dunque non era il caso di "fidarsi" del valore di AL, che il programma principale usa nel suo loop aggiornandolo con una INC AL. All'uscita della procedura in AL c'è la parte bassa del numero di conteggi! Probabilmente MONTI ha dimenticato che AL e AX sono parti diverse dello stesso registro.
Per far funzionare il programma MONTI deve correggere l'uso di AX, per esempio così:

```
    PUSH AX  ; salva AL, carattere corrente (di AH non ci interessa)
    Call ContaCaratteri
    .. la parte dove si usa il conteggio rimane come prima ..
NonMaggiore:
    ; ora serve il carattere corrente: lo recupero dallo stack:
    POP AX   ; riprende AL (anche AH, ma non serve a niente)
    INC AL   ; prossimo carattere ..
    .. il resto del programma rimane uguale ..
```

Una volta corretto quest'errore, il main e la procedura da soli sono ineccepibili. Però insieme non funzionano.
Ci sono altri errori che sono da ascrivere a mancanza di comunicazione; con una documentazione più attenta non sarebbero successi.
MARI, che ha scritto la procedura, non ha avvertito che i registri CX e DX vengono da essa modificati. Per cui la LOOP del programma principale, che si "fida" di CX, non potrà mai funzionare; CX infatti è modificata dalla procedura, che l'usa per fare la sua LOOP, da 1024 a zero. Perciò all'uscita della procedura CX varrà zero.
Un discorso analogo vale per DX, che viene usato dalla procedura per memorizzare temporaneamente il conteggio, dato che AL deve contenere il carattere da cercare. DX viene usato anche dal programma principale, per tenere il massimo in un registro e velocizzare l'esecuzione, ma il suo valore viene regolarmente rovinato dalla procedura ed in uscita da essa sarà sempre uguale ad AX.
Cosa si può fare per rimediare a questi errori? C'è da dire che MONTI non è tenuto a sapere se la procedura modifica alcuni dei registri e non deve essere costretto ad analizzare tutto il codice della procedura, che potrebbe essere molto più complessa.
Peraltro è compito di MARI fare in modo che chi usa la sua procedura sappia esattamente cosa succede.
Per cui il modo migliore per evitare quest'errore sarebbe stato modificare la documentazione; nella parte di intestazione delle procedura, in questo modo:

```
;     Uscite: --------------------
;       AX: conteggio dei caratteri trovati
;   ---------------------------------------------------------------------------
;     Registri modificati: -------
;       AX, CX, DX
;     Effetti collaterali sulla memoria:
;       Nessuno
;   ---------------------------------------------------------------------------
```

Per cui ora MONTI sa che i registri CX e DX vengono modificati dalla procedura e provvederà a salvarli prima della chiamata ed a ripristinarli dopo il ritorno. Qualora non avesse bisogno dei registri modificati dalla procedura potrebbe anche fare a meno di salvarli, risparmiando in tempo di esecuzione.
Si fa notare che è stata inserita nella documentazione della procedura anche una sezione "Effetti collaterali sulla memoria" che recita: "nessuno". Con questa frase si intende che la procedura non modifica nessun Byte della memoria.
Si sottolinea il fatto che non è la stessa cosa scrivere "effetti collaterali: nessuno" e non scrivere nulla. Infatti la presenza della indicazione "effetti collaterali nessuno" ci rassicura che quell'aspetto è stato tenuto in considerazione da chi ha scritto la procedura, e ci dà fiducia sul fatto che essa non modificherà la nostra memoria. Se al contrario non ci fosse stato scritto nulla si sarebbe potuto pensare ad una dimenticanza del programmatore ed aspettarsi effetti strani in memoria.

Un'altra soluzione a questo problema di effetti collaterali è demandare alla procedura tutta la sicurezza dei registri, effettuando nella procedura un salvataggio di contesto completo. In questo caso il programma chiamante si può fidare dei registri che usa e non farà nessun salvataggio.
Inutile dire che anche in questo caso non documentare equivale a non fare. Nel nostro esempio si può fare così:

```
;     Registri modificati: -------
;       AX (gli altri sono salvati e ripristinati)

.. poi, all'inizio del codice della procedura: ..
    PUSH DX
    PUSH CX
    MOV CX, 1024
..
.. ed alla fine, prima della RET: ..
    POP CX
    POP DX
    RET
```

Riflettiamo ora un attimo su cosa fare se non c'è la documentazione sufficiente, per esempio se non sappiamo nulla sui registri eventualmente modificati? In questo caso il programma chiamante non può far altro che salvare e ripristinare tutti i registri che gli servono.
Si badi bene che non è sufficiente provare il programma senza salvare i registri che servono e vedere se funziona lo stesso. Infatti molto spesso durante la fase di test di un programma non si riesce a provare tutte le sue parti, per cui una procedura che non modifica mai AX durante tutta la fase di test non è detto che non lo faccia mai durante tutta la vita operativa del programma!
In questi casi ci si può fidare solo di una dichiarazione esplicita di chi ha fatto la procedura ("la procedura non modifica il registro AX") o di un'analisi ben approfondita del codice (nell'esempio si dovrebbe controllare che in tutto il codice della procedura, comprese le procedure nidificate al suo interno, non c'è MAI una scrittura in AX).
Un caso di "procedure" delle quali non c'è da fidarsi riguardo al salvataggio di contesto è quello dei servizi di MS DOS che, per quanto realizzati con un meccanismo leggermente diverso (interrupt software) sono funzionalmente identici a procedure. I servizi del DOS non garantiscono mai i valori dei registri, ad eccezione naturalmente di quelli che vengono usati come parametri. Dunque usando i servizi DOS dobbiamo salvare e ripristinare tutti i valori dei registri che ci interessano.

Una particolare forma di RET

La RET 80X86 può essere seguita da un operando in immediato da 16 bit, che serve a "fare pulizia" nello stack alla fine della procedura, gettando i parametri passati in ingresso dal programma chiamante.

RET <numero>

Oltre a fare una normale RET, NEAR o FAR, aumenta il valore dello stack pointer del numero indicato in immediato.
La RET <numero> è utile nelle procedure Assembly che si devono collegare a programmi scritti in Turbo Pascal (vedi in seguito).
Esempio:

```
RET 4 ; oltre a ritornare "toglie" due ulteriori word dallo stack
```

Esempio finale
Ordinamento
!!!! da fare