AXO – Architettura dei Calcolatori e Sistemi Operativi

come tradurre da C a 68000
(un possibile modello per generare codice macchina)

catena di traduzione
(breve riassunto)

struttura generale

- il processo di traduzione da linguaggio sorgente di alto livello a linguaggio macchina è diviso in quattro fasi
  - analisi sintattica e trattamento errori - ling. sorgente **alto livello**
  - generazione del codice assembly - ling. assembly
  - ottimizzazione del codice macchina - facoltativa
  - assemblaggio e collegamento - ling. macchina **numerico**
- di solito si vede il risultato: codice macchina eseguibile
- volendo si possono avere le rappresentazioni intermedie
  - albero sintattico del programma
  - codice macchina non ottimizzato
  - eventuali altre forme intermedie …
- spesso si può scegliere il modello specifico di processore per cui tradurre e così ottenere codice macchina “ad hoc”

analisi sintattica e generazione di codice

- analisi sintattica
  - verifica se il programma è sintatticamente corretto
  - segnala e corregge errori
- generazione di codice
  - il programma viene tradotto in linguaggio assembly
  - il codice assembly prodotto è in forma simbolica
  - non numerica ossia non eseguibile direttamente

ling. sorgente di alto livello

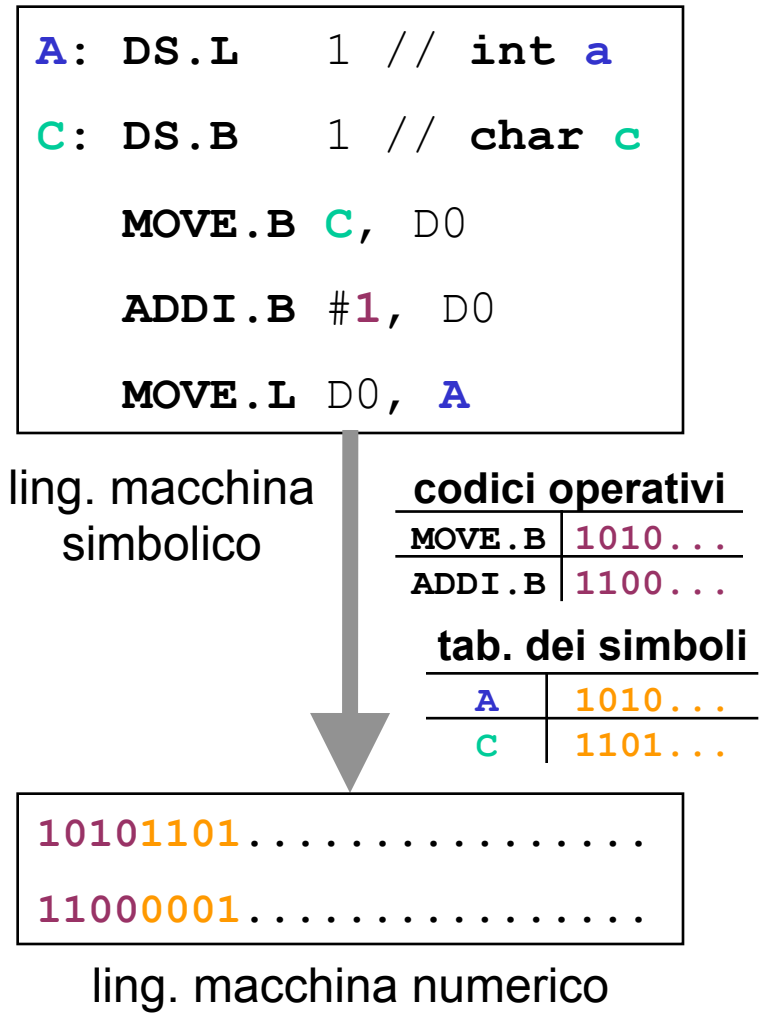
```
int a;

char c;

a = c + 1;
```

```
A: DS.L   1 // int a
C: DS.B   1 // char c
   MOVE.B C, D0
   ADDI.B #1, D0
   MOVE.L D0, A
```

ling. macchina simbolico

assemblaggio e collegamento

- assemblaggio
  - risolve simboli assemblatore
    - indirizzo di memoria
    - etichetta d'istruzione
    - spiazzamento
  - l'istruzione simbolica viene tradotta in codifica numerica
- collegamento
  - unisce più moduli eseguibili
  - per esempio programma e librerie standard (IO ecc)



struttura del programma di alto livello

considerazioni generali

- schema di compilazione, ispirato a GCC, per tradurre da linguaggio sorgente C a linguaggio macchina 68000
- presuppone di conoscere 68000: banco di registri, classi d'istruzioni, modi d'indirizzamento e organizzazione del sottoprogramma (chiamata rientro e passaggio parametri)
- consiste in vari insiemi di convenzioni e regole per
  - segmentare il programma
  - dichiarare le variabili
  - usare (leggere e scrivere) le variabili
  - rendere le strutture di controllo
- non attua necessariamente la traduzione più efficiente
  - È solamente un'esempio didattico
- sono possibili varie ottimizzazioni "ad hoc" del codice

SO e modello di memoria del processo

- per il SO il processo tipico ha tre segmenti essenziali
  - codice main e funzioni utente
  - dati variabili globali e dinamiche
  - pila aree di attivazione con indirizzi parametri e variabili locali
- codice e dati sono segmenti dichiarati nel programma
- il segmento di pila viene creato al lancio del processo
- processi grandi o sofisticati hanno facoltativamente
  - due o più segmenti codice o dati
  - segmenti di dati condivisi
  - segmenti di libreria dinamica
  - e altre peculiarità …
- questo modello di memoria è valido in generale

dichiarare i segmenti tipici

gli indirizzi sono virtuali (non fisici)

```
// var. glob.
...
main (...) {
    // corpo
    ...
}
// funzioni
...
```

```
        TITLE "..." // nome facoltativo
        // segmento dati
        DATA
        // indirizzo iniziale dati
        ORG   num
        ...           // var. glob. e din.
        // segmento codice
        CODE
        // indirizzo iniziale codice
MAIN:   ORG   num
        ...           // istruzioni (main e funz)
        END   MAIN  // indirizzo iniziale
codice
```

non occorre dichiarare il segmento di pila

convenzioni per assegnare memoria e registri

considerazioni generali

- per tradurre da linguaggio sorgente, p. es. C, a linguaggio macchina, p. es. 68000, occorre definire un modello di architettura “run-time” per memoria e processore
- le convenzioni del modello run-time (indicate con la sigla ABI) comprendono
  - collocazione e ingombro delle diverse classi di variabile
  - destinazione d’uso e dimensionamento dei registri
- il modello di architettura run-time consente interoperabilità tra porzioni di codice di provenienza differente, come per esempio codice utente e librerie standard precompilate
- esempio tipico in linguaggio C è la libreria standard di IO

convenzioni per variabili

- le variabili del programma sono collocate così
  - globali in memoria a indirizzo fissato
  - locali nelle aree di attivazione in pila
  - dinamiche in memoria (qui non sono considerate in quanto allocate nello heap e gestite dal Sistema Operativo)
- le istruzioni aritmetico-logiche operano su registri
- dunque per operare sulla variabile (glob – loc – din)
  - prima caricala in un registro libero (load)
  - poi elaborala nel registro (confronto e aritmetica-logica)
  - infine riscrivila in memoria (store)
- talvolta si può ottimizzare senza passare per i registri

ingombro in memoria delle variabili

- l'unità di memoria minima indirizzabile è il singolo byte
- in C sotto Linux a 32 bit i diversi tipi di variabile ingombrano così

```
sizeof (char)       = 1 byte
sizeof (short int)  = 2 byte
sizeof (int)        = 4 byte
sizeof (long int)   = 4 byte (non c'è in 68000)
sizeof (array)      = somma ingombri elementi
sizeof (struct)     = somma ingombri campi
```

- Differenti convenzioni posso essere usate per altri sistemi operativi (es. `sizeof (long int) = 8 byte` a 64 bit) e per diverse architetture hardware.
- per il tipo reale (float) vedi lo standard IEEE 754
  (qui non si considera il tipo reale)

convenzioni per uso dei registri

- usa i registri di 68000 nel modo seguente
  - registri di dato D0-D7 per variabili di tipo carattere o intero
  - registri d'indirizzo A0-A7 per variabili di tipo indirizzo, così
    - registri A0-A5 per puntatori oppure per indici a vettori
    - registro A6 (o FP) come puntatore all'area di attivazione
    - registro A7 (o SP) come puntatore alla pila (uSP o sSP)
- il registro SR contiene i bit di esito
  - è aggiornato dalle istruzioni macchina di trasf. conf. e arit.-log. (MOVE – CMP – ADD SUB NEG ecc – AND OR NOT ecc)
  - è esaminato dalle istruzioni macchina di salto condizionato (Bcc)
  - contiene anche il bit di modo e i bit di priorità d'interruzione
- il registro PC è il contatore di programma
  (e per lo più è usato automaticamente dai salti)

dimensionare i registri di dato

- dimensiona i registri di dato (D0-D7) così
  - a 8 bit (byte) per carattere
  - a 16 o 32 bit per intero
    - in Linux l'intero è a 32 bit
    - in Windows 95/98 (sistema operativo a 16 bit) l'intero è a 16 bit
- dimensionare il dato: usa un suffisso
  - **B** per dato da 8 bit byte
  - **W** per dato da 16 bit parola
  - **L** per dato da 32 bit parola lunga o doppia
- concatena il suffisso al nome dell'istruzione
  - `MOVE.B D1, D2  // D2 ← [D1] 8 bit meno signif.`
  - `MOVE.W D1, D2  // D2 ← [D1] 16 bit meno signif.`
  - `MOVE.L D1, D2  // D2 ← [D1] registro completo`

dimensionare i registri d'indirizzo

- dimensiona i registri d'indirizzo (A0-A5) così
  - a 16 bit per memoria fisica max da 64 K byte
  - a 32 bit per memoria fisica max da 4 G byte
- dimensionare l'indirizzo: usa un suffisso
  - **W** per indirizzo da 16 bit indirizzo corto
  - **L** per indirizzo da 32 bit indirizzo lungo
- concatena il suffisso al nome dell'istruzione
  - `MOVEA.W A1, A2  // A2 ← [A1] 16 bit meno signif.`
  - `MOVEA.L A1, A2  // A2 ← [A1] registro completo`

come dichiarare le diverse classi di variabile

considerazioni generali

- in C la variabile è un oggetto formale e ha
  - nome per identificarla e farne uso
  - tipo per stabilirne gli usi ammissibili
- in 68000 la variabile è un elemento di memoria (byte parola o regione di mem) e ha una "collocazione" con
  - nome per identificarla
  - modo per indirizzarla
- in 68000 la variabile viene manipolata tramite indirizzo simbolico o nome di registro
- occorre però distinguere tra diverse classi di variabile (var globale – parametro – var locale – eventualmente var dinamica)

variabile globale nominale

- la variabile globale è collocata in memoria a indirizzo virtuale fisso stabilito dall'assemblatore
- per comodità l'indirizzo simbolico della variabile globale coincide con il nome della variabile
- gli ordini di dichiarazione e disposizione in memoria delle variabili globali coincidono
- il processore gestisce l'accesso disallineato a memoria
- pertanto le variabili globali sono collocate in memoria consecutivamente, senza lasciare byte inutilizzati

variabile globale - scalare e vettore

intero ordinario e corto a 32 e 16 bit rispettivamente

```
char c;
int a;
int b = 5;
int vet [10];
int * punt;
short int d;
```

tab. dei simboli

C	1000
A	1001
B	1005
VET	1009
PUNT	1039
D	1043

```
        ORG   1000   // decimale
C:      DS.B  1      // spazio per 1 byte
A:      DS.L  1      // oppure DS.B 4
B:      DC.L  5      // inizializzazione
VET:    DS.L  10     // oppure DS.B 40
PUNT:   DS.L  1      // oppure DS.B 4
D:      DS.W  1      // oppure DS.B 2
```

DS riserva solo spazio senza inizializzarlo
DC riserva spazio e lo inizializza
il puntatore (di ogni tipo) è equiparato all'intero

variabile globale - struct (per completezza)

intero ordinario a 32 bit

```
struct s {
    char c;
    int  a;
}
```

```
        ORG  1000   // decimale
S:      DS.B 5      // = somma ingombri di c e a
S.C:    EQU  0      // spiazzamento di c in s
S.A:    EQU  1      // spiazzamento di a in s
```

tab. dei sim.

S	1000
S.C	0
S.A	1

i campi c e a ingombrano un byte e una parola lunga, rispettivamente, pertanto la struct s ingombra cinque byte; la direttiva DS assegna cinque byte a s; la direttiva EQU dichiara gli spiazzamenti S.C e S.A dei campi c e a, rispettivamente, all'interno di s; il compilatore rinomina gli spiazzamenti, incorporando il nome della struct, per non confonderli con quelli di eventuali campi omonimi in altre struct

parametro in ingresso alla funzione

- il parametro è impilato nell'area di attivazione con spiazzamento fisso stabilito dall'assemblatore
- per comodità lo spiazzamento simbolico del parametro coincide con il nome del parametro
- gli ordini di dichiarazione e disposizione in pila dei parametri sono opposti
  - il primo parametro attuale passato è l'ultimo impilato (indirizzo più piccolo)
  - l'ultimo parametro attuale passato è il primo impilato (indirizzo più grande)
- il processore gestisce l'accesso disallineato a memoria
- pertanto i parametri sono impilati consecutivamente, senza lasciare byte inutilizzati

variabile locale nominale

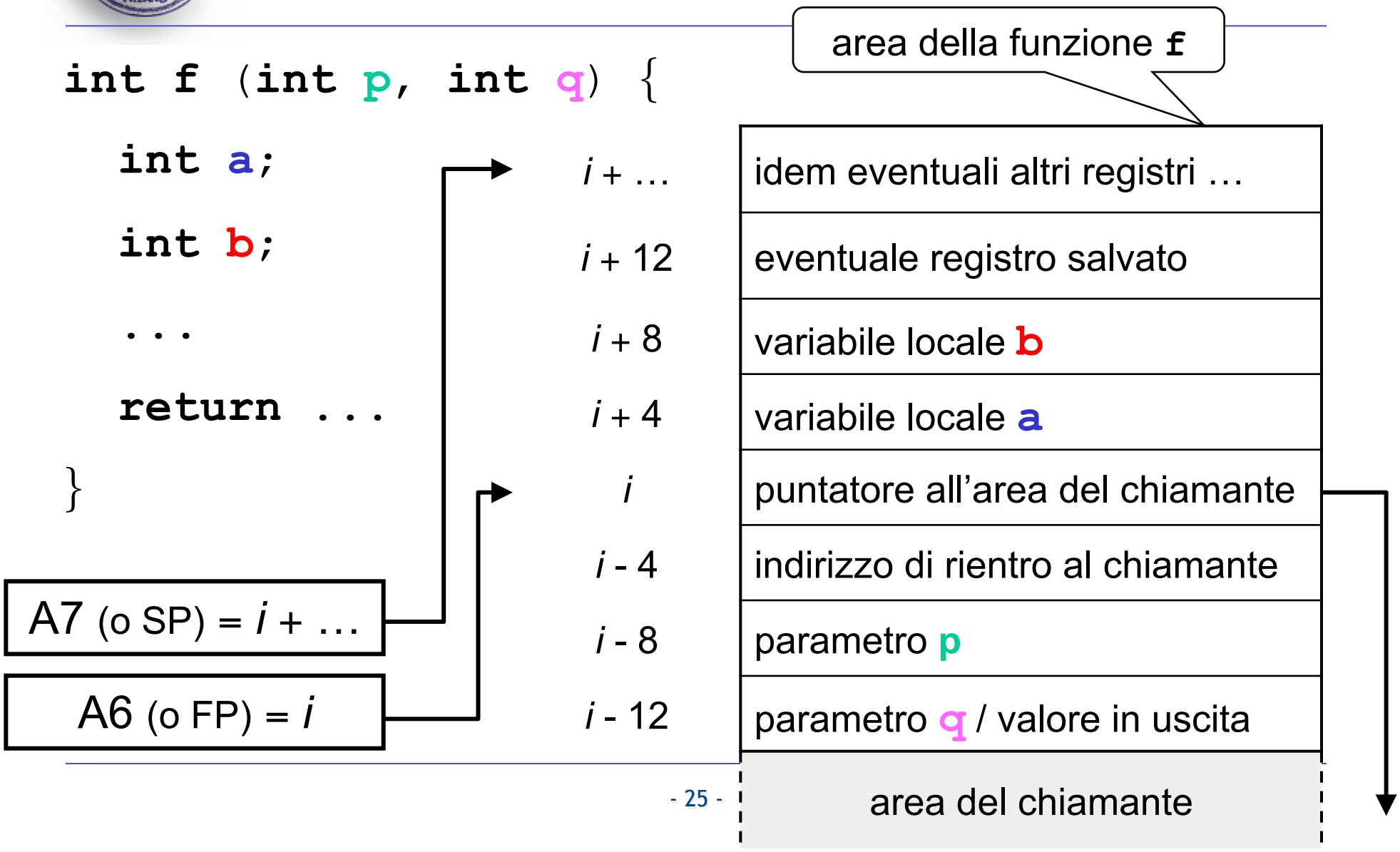
- la variabile locale è impilata nell'area di attivazione con spiazzamento fisso stabilito dall'assemblatore
- per comodità lo spiazzamento simbolico della variabile locale coincide con il nome della variabile
- gli ordini di dichiarazione e disposizione in pila delle variabili locali coincidono
  - la prima variabile locale dichiarata è la prima impilata
  - l'ultima variabile locale dichiarata è l'ultima impilata
- Convenzione opposta a quella del passaggio dei parametri
- il processore gestisce l'accesso disallineato a memoria
- pertanto le variabili locali impilate consecutivamente, senza lasciare byte inutilizzati

valore in uscita dalla funzione - casi speciali

- il valore in uscita alla funzione va sovrascritto al primo parametro impilato (o ai primi se le taglie differiscono), ossia all'ultimo (o agli ultimi) parametro formale dichiarato
- se la funzione ha valore in uscita ma non ha parametri, l'area di attivazione ha comunque spazio per il valore
- se la funzione non ha né parametri né valore in uscita, l'area di attivazione non ha spazio per parametri o valore
- se la funzione non ha variabili locali, l'area di attivazione non ha spazio per variabili locali
- eventualmente ci sono registri salvati in cima alla pila
  (solo alcuni o tutti, tali registri non fanno parte dell'area di attivazione)

area di attivazione

```
int f (int p, int q) {
   int a;
   int b;
   ...
   return ...
}
```



parametri e variabili locali

```
// chiamante
... // impila parametri
// funzione
int f (int p, int q) {
  int a;      // dich.
  int b;      // dich.
  ...         // elabora
  return ... // uscita
}
```

tab. dei sim.

p	-8
Q	-12
A	4
B	8

```
F:    LINK FP, #-8
Q:    EQU  -12 // spi. par. q
P:    EQU  -8  // spi. par. p
A:    EQU  4   // spi. var. a
B:    EQU  8   // spi. var. b
      ...  // elabora
      ...  // sovrascrive Q
      UNLK FP
      RFS
```

EQU dichiara il simbolo senza riservare spazio di memoria; lo spazio viene riservato dal chiamante (impilando i parametri) e da LINK (nella funzione)

come usare le diverse classi di variabile scalare

usare la variabile - regola base

CONSTATAZIONE

- la variabile è sempre collocata in memoria
- gran parte delle istruzioni lavora solo nei registri
  (o ha vincoli di ortogonalità tali da limitarne l'uso in memoria)

REGOLA BASE PER TRADURRE

- se hai uno statement C che usa e magari modifica una variabile (per esempio l'assegnamento `a = a + b + 1`), comportati così con la variabile (nell'esempio `a`)
  - caricala nel registro all'inizio dello statement (o non appena serve)
  - e memorizzala (se è stata modificata) alla fine dello statement
- se la variabile figura nello statement C successivo, non tentare di "tenerla" nel registro – memorizzala e ricaricala !

usare la variabile - ottimizzazione

TUTTAVIA …

- certe istruzioni macchina possono lavorare direttamente in memoria
- scegli tu se avvalerti o no di questa possibile ottimizzazione di codice
- niente perfezionismo: prima chiarezza e correttezza, poi efficienza !
- i “compilatori ottimizzanti” sfruttano in pieno i modi d’indirizzamento

NEL SEGUITO SI RISPETTA SEMPRE LA REGOLA BASE

FALLO ANCHE TU DI PREFERENZA E IN PARTICOLARE ALL’ESAME

(talvolta però nel seguito si mostrano ottimizzazioni semplici …)

variabile globale nominale - carattere

il registro D0 è libero (o è stato liberato salvandolo in pila)

```
char c;
...
c = 'z';
...
```

```
C:      DS.B    1           // spazio per char
        MOVE.B  #'z', D0    // D0 ← 'z'
        MOVE.B  D0, C       //  C ← [D0]
```

oppure (ottimizza senza passare per il registro D0)

```
        MOVE.B  #'z', C     //  C ← 'z'
```

dato che MOVE può lavorare direttamente in memoria

variabile globale nominale - intero

registro D0 libero (o liberato) – intero a 32 bit

```
int a;
...
a = 1;
...
```

```
A:      DS.L   1            // spazio per int
        MOVE.L #1, D0       // D0 ← 1
        MOVE.L D0, A        //  A ← [D0]
```

oppure (ottimizza senza passare per il registro D0)

```
        MOVE.L #1, A        //  A ← 1
```

dato che MOVE può lavorare direttamente in memoria

variabile globale nominale - intero - 1

registro D0 libero (o liberato) – intero a 32 bit

```
int a;
...
a = a + 1;
...
```

```
A:      DS.L   1          // spazio per int
        MOVE.L A, D0      // D0 ← [A]
        ADDI.L #1, D0     // D0 ← [D0] + 1
        MOVE.L D0, A      //  A ← [D0]
```

oppure (ottimizza senza passare per il registro D0)

```
        ADDI.L #1, A      //  A ← [A] + 1
```

dato che ADDI può lavorare direttamente in memoria

variabile globale nominale - intero - 2

registri D0-D1 liberi (o liberati) – intero a 32 bit

```
int a;
int b;
...
a = a + b;
...
```

```
A:      DS.L   1          // spazio per int a
B:      DS.L   1          // spazio per int b
        MOVE.L A, D0      // D0 ← [A]
        MOVE.L B, D1      // D1 ← [B]
        ADD.L  D0, D1     // D1 ← [D0] + [D1]
        MOVE.L D1, A      //  A ← [D1]
```

oppure (ottimizza senza passare per il registro D1)

```
        MOVE.L A, D0      // D0 ← [A]
        ADD.L  B, D0      // D0 ← [B] + [D0]
        MOVE.L D0, A      //  A ← [D0]
```

dato che ADD può lavorare direttamente in memoria

variabile globale per puntatore - intero

registri D0 e A0 liberi (o liberati) – dato a 32 bit e indir. a 16 bit

```
int a;
int * punt;
...
punt = &a;
*punt = *punt + 1;
...
```

```
A:     DS.L    1          // spazio per int
PUNT:  DS.W    1          // spazio per punt
       // punt = &a
       MOVEA.W #A, A0     //   A0 ← A
       MOVE.W  A0, PUNT   // PUNT ← [A0]
       // *punt = *punt + 1
       MOVEA.W PUNT, A0   //   A0 ← [PUNT]
       MOVE.L  (A0), D0   //   D0 ← [[A0]]
       ADDI.L  #1, D0     //   D0 ← [D0] + 1
       MOVE.L  D0, (A0)   // [A0] ← [D0]
```

non confondere tra assegnamento a puntatore (che ne modifica la cella di memoria) e a oggetto puntato (che non modifica la cella di memoria del puntatore)

variabile locale nominale - intero

registro D0 libero (o liberato) – intero a 32 bit

```
int f (...) {
    int a;
    ...
    a = a + 1;
    ...
}
```

```
    ...                 // LINK e altre EQU
A:  EQU     8           // vedi area di attivazione
    MOVE.L  A(FP), D0   //        D0 ← [A + [FP]]
    ADDI.L  #1, D0      //        D0 ← [D0] + 1
    MOVE.L  D0, A(FP)   // A + [FP] ← [D0]
```

oppure (ottimizza senza passare per il registro D0)

```
    ADDI.L  #1, A(FP)   // A + [FP] ← [A + [FP]] + 1
```

dato che ADDI può lavorare direttamente in memoria

parametro di funzione - intero

registro D0 libero (o liberato) – intero a 32 bit

```
int f (..., int q) {
   int a;
   ...
   a = q + 1;
   ...
}
```

```
   ...                  // LINK e altre EQU
Q: EQU     -12          // vedi area di attiv.
A: EQU     4            // vedi area di attiv.
   MOVE.L Q(FP), D0     //        D0 ← [Q + [FP]]
   ADDI.L #1, D0        //        D0 ← [D0] + 1
   MOVE.L D0, A(FP)     // A + [FP] ← [D0]
```

oppure (ottimizza senza passare per il registro D0)

```
   ADDI.L #1, A(FP)  // A + [FP] ← [A + [FP]] + 1
```

dato che ADDI può lavorare direttamente in memoria

valore in uscita da funzione - intero

registri D0 e D1 liberi (o liberati) – intero a 32 bit

```
int f (int p, int q) {
   int a;
   ...
   return (p + a);
}
```

ricorda che il valore in uscita va sovrascritto all'ultimo parametro formale, qui `q`

```
...                          // LINK e altre EQU
Q:  EQU     -12              // vedi area di attiv.
p:  EQU     -8               // vedi area di attiv.
A:  EQU     4                // vedi area di attiv.
    MOVE.L  P(FP), D0        //          D0 ← [P + [FP]]
    MOVE.L  A(FP), D1        //          D1 ← [A + [FP]]
    ADD.L   D0, D1           //          D1 ← [D0] + [D1]
    MOVE.L  D1, Q(FP)        // Q + [FP] ← [D1]
```

e magari si può ottimizzare sfruttando la semiortogonalità di ADD e la piena ortogonalità di MOVE … (esercitati se vuoi)

come rendere le strutture di controllo

considerazioni generali

- le strutture di controllo comportano l'uso di salto
- servono etichette univoche per ogni struttura

- qui si danno schemi di salto corretti e uniformi
- ma non necessariamente i più efficienti possibili

- traduci in cascata le strutture di controllo annidate
- parti dalla struttura di controllo più interna e risali

goto - salto incondizionato

in C la struttura “goto” modella il salto incondizionato

```
        ... // parte I
        goto MARCA
        ... // parte II
MARCA:  ... // destinazione
        ... // parte III
```

in C ben strutturato “goto” è usato raramente, ma esiste

```
        ...         // parte I
        BRA MARCA   // va' a MARCA
        ...         // parte II
MARCA:  ...         // destinazione
        ...         // parte III
```

naturalmente si può anche saltare indietro

if - condizionale a una via

registro D0 libero (o liberato) – intero a 32 bit

```
// var. globale
int a;
...
// condizione
if (a == 5) {
   // ramo then
   ...
} /* end if */
... // seguito
```

```
A:      DS.L   1        // riserva mem per a
        ...
        MOVE.L A, D0    //     D0 ← [A]
        CMPI.L #5, D0   // esiti ← [D0] - 5
        BNE    FINE     // se != 0 va' a FINE
        ...              // ramo then
FINE:   ...              // seguito
```

va modificata l'istruzione di branch, com'è ovvio, per ">", "<", "<=", "==" e "!="

if - condizionale a due vie

registro D0 libero (o liberato) – intero a 32 bit

```
// var. globale
int a;
...
// condizione
if (a >= 5) {
    // ramo then
    ...
} else {
    // ramo else
    ...
} /* end if */
... // seguito
```

```
A:      DS.L   1        // riserva mem per a
        ...
        MOVE.L A, D0    //    D0 ← [A]
        CMPI.L #5, D0   // esiti ← [D0] - 5
        BLT    ELSE     // se < 0 va' a ELSE
        ...             // ramo then
        BRA    FINE     // va' a FINE
ELSE:   ...             // ramo else
FINE:   ...             // seguito
```

va modificato com'è ovvio per ">", "<", "<=", "==" e "!="

idem se **a** è variabile locale od oggetto puntato

while - ciclo a condizione iniziale

registro D0 libero (o liberato) – intero a 32 bit

```
// var. globale
int a;
...
// condizione
while (a >= 5) {
    // corpo
    ...
} /* end while */
// seguito
...
```

```
A:      DS.L   1        // riserva mem per a
        ...
CICLO:  MOVE.L A, D0    //    D0 ← [A]
        CMPI.L #5, D0   // esiti ← [D0] - 5
        BLT    FINE     // se < 0 va' a FINE
        ...             // corpo del ciclo
        BRA    CICLO    // torna a CICLO
FINE:   ...             // seguito del ciclo
```

va modificato com'è ovvio per ">", "<", "<=", "==" e "!="

idem se **a** è variabile locale od oggetto puntato

do - ciclo a condizione finale

registro D0 libero (o liberato) – intero a 32 bit

```
// var. globale
int a;
...
do {
   // corpo
   ...
// condizione
} while (a >= 5);
// seguito
...
```

```
A:      DS.L   1        // riserva mem per a
        ...
CICLO:  ...             // corpo del ciclo
        MOVE.L A, D0    //    D0 ← [A]
        CMPI.L #5, D0   // esiti ← [D0] - 5
        BGE    CICLO    // se >= 0 torna a
CICLO
FINE:   ...             // seguito del ciclo
```

va modificato com'è ovvio per ">", "<", "<=", "==" e "!="
idem se **a** è variabile locale od oggetto puntato

for - ciclo a conteggio

registro D0 libero (o liberato) – intero a 32 bit

```
// variabile globale
int a;
...
// testata del ciclo
for (a = 1; a <= 5; a++) {
    // corpo del ciclo
    ...
} /* end for */
// seguito del ciclo
...
```

la variabile di conteggio `a` viene aggiornata in fondo al corpo del ciclo (“`a`++” è post-incremento)

```
A:        DS.L   1        // riserva mem per a
          ...
a = 1     MOVE.L #1, D0   //     D0 ← 1
          MOVE.L D0, A    //      A ← [D0]
CICLO:    MOVE.L A, D0    //     D0 ← [A]
a <= 5    CMPI.L #5, D0   // esiti ← [D0] - 5
          BGT    FINE     // se > 0 va' a FINE
          ...             // corpo del ciclo
          MOVE.L A, D0    //     D0 ← [A]
a++       ADDI.L #1, D0   //     D0 ← [D0] + 1
          MOVE.L D0, A    //      A ← [D0]
          BRA    CICLO    // torna a CICLO
FINE:     ...             // seguito del ciclo
```

va modificato per “>”, “<“, “<=”, e “`a`--”, “++`a`”, “--`a`”

idem se `a` è variabile locale od oggetto puntato

break - troncamento di ciclo

registro D0 libero (o liberato) – intero a 32 bit

```
// var. globale
int a;
...
// condizione
while (...) {
   // corpo I
   ... // corpo I
   if (a == 5) {
      break;
   } /* end if */
   ... // corpo II
} /* end while */
... // seguito
```

```
A:      DS.L   1        // riserva mem per a
        ...
CICLO:  ...             // condizione ciclo
        ...             // corpo del ciclo I
        MOVE.L A, D0    //    D0 ← [A]
        CMPI.L #5, D0   // esiti ← [D0] - 5
        BEQ    FINE     // se = 0 va' a FINE
        ...             // corpo del ciclo II
FINE:   ...             // seguito del ciclo
```

va modificato com'è ovvio per ">", "<", "<=", "==" e "!="

idem se **a** è variabile locale od oggetto puntato

costrutto valido per ogni tipo di ciclo

come usare i diversi tipi di variabile strutturata

considerazioni generali

- la variabile strutturata è un aggregato di elementi
- accesso a singolo elemento con indice costante
  - per indice ed eventualmente spiazzamento
  - per puntatore ed eventualmente spiazzamento
- accesso a singolo elemento con indice casuale
- scansione con indice o puntatore scorrevole
- sono possibili diverse varianti e ottimizzazioni

- vettori e struct hanno peculiarità rispettive
- nel seguito pochi casi semplici ed esplicativi
- solo per la classe di variabile globale

vettore - indice costante

registri D0 e A0 liberi (o liberati) – elem e indice a 32 bit

```
// variabili globali
int v [5]; // vettore
...
// indice costante
v[1] = 4;
v[3] = v[2] + 5;
...
```

attenzione estremi del vettore

con aritmetica dei puntatori

```
*(v + 1) = 4;
Ecc
```

```
V:  DS.L    5           // riserva mem per v
    // v[1] = 4
    MOVEA.L #V, A0      //          A0 ← V
    MOVE.L  #4, D0      //          D0 ← 4
    MOVE.L  D0, 4(A0)   //  4 + [A0] ← [D0]
    // v[3] = v[2] + 5
    MOVEA.L #V, A0      //          A0 ← V
    MOVE.L  8(A0), D0   //          D0 ← [8 + [A0]]
    ADDI.L  #5, D0      //          D0 ← [D0] + 5
    MOVE.L  D0, 12(A0)  // 12 + [A0] ← [D0]
```

poiché il vettore ha elementi con taglia di quattro byte, lo spiazzamento è pari all'indice costante di posizione moltiplicato per quattro

non è applicabile se *indice elemento* × 4 > *spiazzamento max*

vettore - accesso casuale

registri D0 e A0 liberi (o liberati) – elem e indice a 32 bit

```
// variabili globali
char v [5]; // vettore
int a;      // indice
...
// accesso casuale
v[a] = 'z';
...
```

attenzione estremi del vettore

con aritmetica dei puntatori

```
*(v + a) = 'z';
```

```
V:  DS.B    5           // riserva mem per v
A:  DS.L    1           // riserva mem per a
    MOVEA.L #V, A0      //   A0 ← V
    MOVE.L  A, D0       //   D0 ← [A]
    ADDA.L  D0, A0      //   A0 ← [D0] + [A0]
    MOVE.L  #'z', (A0)  // [A0] ← 'z'
```

qui la somma di base del vettore **v** e indice variabile **a** riesce semplice perché gli elementi del vettore sono byte
in generale l'indice variabile va scalato di un fattore 2, 4, ecc secondo la taglia in byte dell'elemento del vettore
pertanto l'accesso tramite variabile può riuscire complesso
… però lo schema mostrato qui non ha restrizioni …

vettore - scansione sequenziale

registri D0-D1 e A0 liberi (o liberati) – elem e indice a 32 bit

```
// variabili globali
int v [5]; // vettore
int a;     // indice
...
// testata del ciclo
for (a = 2; a <= 3; a++) {
   v[a] = v[a + 1] - 6;
} /* end for */
// seguito del ciclo
...
```

attenzione estremi del vettore

con aritmetica dei puntatori

```
// prova a farlo !
```

```
V:  DS.L    5          // riserva mem per v
A:  DS.L    1          // riserva mem per a
    // inizializza a = 2 - usa D0
CI: // verifica a <= 3   - usa D0
    BGT     FI         // se > 0 va' a FI
    MOVEA.L #V + 8, A0 //   A0 ← V + 8
    MOVE.L  4(A0), D1  //   D1 ← [4 + [A0]]
    SUBI.L  #6, D1     //   D1 ← [D1] - 6
    MOVE.L  D1, (A0)+  // [A0] ← [D1] e poi
                       //   A0 ← [A0] + 4
    // aggiorna a++       - usa D0
    BRA     CI         // torna a CI
FI: ...                // seguito ciclo
```

poiché il dato ha taglia di quattro byte, la scansione parte dall'indirizzo `V + 2 × 4 = V + 8`, lo spiazzamento vale `1 × 4 = 4` e l'auto-incremento somma `4` all'indirizzo in A0

struct (per completezza)

registri D0 e A0 liberi (o liberati) – puntatore a 32 bit

```
struct s {
    char c;
    int a;
}
...
s.c = 'z';
...
s.a = s.a + 1;
...
```

```
S:    DS.B    5            // somma ingombri di c e a
S.C:  EQU     0            // spiazzamento di c in s
S.A:  EQU     1            // spiazzamento di a in s
      // s.c = 'z'
      MOVEA.L #S, A0       //          A0 ← S
      MOVE.B  #'z', D0     //          D0 ← 'z'
      MOVE.L  D0, S.C(A0)  // S.C + [A0] ← [D0]
      // s.a = s.a + 1
      MOVEA.L #S, A0       //          A0 ← S
      MOVE.L  S.A(A0), D0  //          D0 ← [S.A + [A0]]
      ADDI.L  #1, D0       //          D0 ← [D0] + 1
      MOVE.L  D0, S.A(A0)  // S.A + [A0] ← [D0]
```

naturalmente si potrebbe ottimizzare …

come ottimizzare codice
(osservazioni a margine)

considerazioni riassuntive

- ormai hai viste svariate ottimizzazioni di codice
- altre ne avrai intuite e magari sperimentate da te
- fondamentalmente sono dei due tipi seguenti
  - sfruttare l'ortogonalità e lavorare direttamente in memoria
  - tenere in registro una variabile pertinente a più statement
- si basano sulla capacità di avvalersi appieno
  - di ortogonalità e repertorio di modi d'indirizzamento
  - di una buona schedulazione dei registri di processore
- i compilatori ottimizzanti hanno algoritmi sofisticati per sfruttare a fondo modi e ortogonalità, e schedulare bene

conclusioni operative

- tu però non sei un compilatore ottimizzante, bensì un programmatore umano inefficiente
- non tentare di produrre direttamente codice ottimizzato !
- piuttosto produci codice “plain” corretto, va benissimo !
- e se proprio vuoi o devi ottimizzare, procedi così
  - scrivi codice “plain” senza alcuna ottimizzazione
  - ripassa il codice plain scritto e se possibile
    - unifica istruzioni sfruttando ortogonalità e modi
    - elimina load e store tenendo variabili in registri
  - può servire ripassare due o più volte
- anche i compilatori spesso procedono in questo modo

esempio di ottimizzazione

```
int a;
int b;
int c;
...
a = a + b;
c = a + 2;
...
```

codice C di alto livello sorgente

```
A:  DS.L   1
B:  DS.L   1
C:  DS.L   1
    // a = a + b
    MOVE.L A, D0
    MOVE.L B, D1
    ADD.L  D0, D1
    MOVE.L D1, A
    // c = a + 2
    MOVE.L A, D0
    ADDI.L #2, D0
    MOVE.L D0, C
```

codice 68000 “plain” ossia non ottimizzato

```
A:  DS.L   1
B:  DS.L   1
C:  DS.L   1
    // a = a + b
    MOVE.L A, D0
    ADD.L  B, D0
    MOVE.L D0, A
    // c = a + 2
    MOVE.L A, D0
    ADDI.L #2, D0
    MOVE.L D0, C
```

unificate

unifica tramite semi-ortogonalità di ADD

```
A:  DS.L   1
B:  DS.L   1
C:  DS.L   1
    // a = a + b
    MOVE.L A, D0
    ADD.L  B, D0
    MOVE.L D0, A
    // c = a + 2
    ADDI.L #2, D0
    MOVE.L D0, C
```

eliminata

elimina tenendo var in registro dato D0

come rendere espressione e condizione

calcolare l'espressione

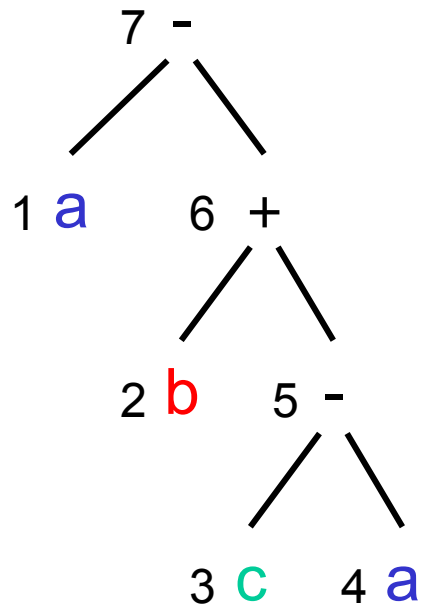
- l'espressione modella un calcolo complesso
- va tradotta da C a 68000 in modo sistematico
- calcola l'espressione nei registri di tipo dato

- esamina l'albero sintattico e scrivi il codice
  - quando l'operando va in uso, caricalo in registro
  - esegui nei registri le operazioni prescritte
- referenzia gli operandi come hai visto prima
  - secondo siano variabili globali o locali nominali
  - oppure siano variabili riferite per puntatore
- prima di scrivere il codice, alloca i registri per le variabili
- se non ci sono sufficienti registri liberi, liberali

esempio di espressione

a − (**b** + (**c** − **a**)) *a*, *b*, *c* var glob – intero 32 bit

albero sintattico




```
// dich.
A: DS.L 1
B: DS.L 1
C: DS.L 1
```

```
// schedula i registri D0-D3
// se occorre libera D0-D3
MOVE.L A, D0   // 1 : carica a
MOVE.L B, D1   // 2 : carica b
MOVE.L C, D2   // 3 : carica c
MOVE.L A, D3   // 4 : (ri)carica a
SUB.L  D3, D2  // 5 : c - a
ADD.L  D1, D2  // 6 : b + ...
SUB.L  D2, D0  // 7 : a - ...
// risultato in D0
```

ottimizza senza ricaricare **a** …

modifica se var loc o par o punt

verificare la condizione

- la condizione modella un confronto complesso
- contiene una o due espressioni da confrontare

- calcola le due espressioni separatamente
- confrontane i risultati e salta condizionatamente
- per calcolare tali espressioni, ricorri al metodo

- spesso contiene operatori logici (AND OR e NOT)
- da trattare come le istruzioni aritmetiche

esempio di condizione - generico

sufficienti registri di dato liberi (o liberati) – intero 32 bit

```
...
// condizione
if (espr1 >= espr2) {
    // ramo then
    ...
} /* end if */
// seguito
...
```

```
        // calcola espr1 - ris. in D1
        // calcola espr2 - ris. in D2
        CMP.L D2, D1   // esiti ← [D1] - [D2]
        BLT   FINE     // se < 0 va' a FINE
        ...            // ramo then
FINE:   ...            // seguito del condiz.
```

va modificato com'è ovvio per ">", "<", "<=", "==" e "!=", nonché per le altre strutture di controllo

e vale anche per "&&", "||" e "!" (AND OR e NOT)

esempio di condizione - specifico - arit

a − b >= c + a *a*, *b*, *c* var glob – intero 32 bit

```
...
// condizione
if (a - b >= c + a) {
    // ramo then
    ...
} /* end if */
// seguito
...
```

```
// dich.
A: DS.L 1
B: DS.L 1
C: DS.L 1
```

```
// schedula i registri D0-D3
// se occorre libera D0-D3
MOVE.L A, D0   // carica a
MOVE.L B, D1   // carica b
SUB.L  D1, D0  // a - b
MOVE.L C, D2   // carica c
MOVE.L A, D3   // carica a
ADD.L  D2, D3  // c + a
CMP.L  D3, D0  // conf. a-b e c+a
BLT    FINE    // se <0 va' a FINE
...            // ramo then
FINE:  ...     // seguito
```

ottimizza e modifica per var loc par o punt …

esempio di condizione - specifico - log

(a || b) && !c *a*, *b*, *c* var glob – intero 32 bit

```
...
// condizione
if ((a || b) && !c) {
    // ramo then
    ...
} /* end if */
// seguito
...
```

riscrivi l'espressione logica

`(a || b) && !c`

così

`((a || b) && !c) != 0`

e procedi come visto prima

```
// dich.
A: DS.L 1
B: DS.L 1
C: DS.L 1
```

```
// schedula i registri D0-D2
// se occorre libera D0-D2
MOVE.L A, D0   // carica a
MOVE.L B, D1   // carica b
OR.L   D0, D1  // a || b
MOVE.L C, D2   // carica c
NOT.L  D2      // !c
AND.L  D1, D2  // ... && ...
CMPI.L #0, D2  // confronta con 0
BEQ    FINE    // se =0 va' a FINE
...            // ramo then
FINE:  ...     // seguito
```

ottimizza e modifica per var loc par o punt …

come chiamare la funzione e rientrare

considerazioni generali

- lo schema di chiamata-rientro si basa su
  - modello di area di attivazione
  - istruzioni LINK, UNLK e RFS

  che dovresti ormai conoscere bene …
- per scrivere codice uniforme, comportati così
  - per impilare-spilare parametri e valore, passa per i registri
  - per elaborare variabili e parametri, passa per i registri
  - prima di scrivere il codice, schedula i registri opportuni
    (stabilisci a quali scopi destinarli – un registro per ciascuno scopo)
  - se serve salva i registri in pila all'inizio e ripristinali alla fine
- sono possibili ottimizzazioni più o meno evidenti
- scegli tu se ottimizzare, ma evita complicazioni

esempio di funzione

```
// funzioni
int f (int p, int q) {
  int a, ...;
  a = 1;
  return (p + a);
} /* end f */
// variabili globali
char c;
int d;
// prog principale
main ( ) {
  d = 2;
  d = f (d, 3);
} /* end main */
```

sono possibili ottimizzazioni

```
C:    DS.B   1
D:    DS.L   1
MAIN: MOVE.L #2, D0
      MOVE.L D0, D
      // impila 3 (= q)
      MOVE.L #3, D0
      MOVE.L D0, -(SP)
      // impila d (= p)
      MOVE.L D, D0
      MOVE.L D0, -(SP)
      BSR    F
      // abbandona p
      ADDA.L #4, SP
      // spila val usc
      MOVE.L (SP)+, D0
      MOVE.L D0, D
```

```
F: LINK     FP, #-8
Q: EQU      -12
p: EQU      -8
A: EQU      4
B: EQU      8
// MOVEM.L  D0-D1, -(SP)
   MOVE.L   #1, D0
   MOVE.L   D0, A(FP)
   MOVE.L   P(FP), D0
   MOVE.L   A(FP), D1
   ADD.L    D0, D1
   MOVE.L   D1, Q(FP)
// MOVEM.L  (SP)+, D0-D1
   UNLK     FP
   RFS
```

idem - con segmentatazione

```
// segmento dati
   DATA
// indir iniziale
   ORG     ...
// var globali
C: DS.B    1
D: DS.L    1
```

ricorda che il segmento di pila non va dichiarato

```
// segmento codice
      CODE
// indir iniziale
MAIN: ORG     ...
      MOVE.L #2, D0
      MOVE.L D0, D
      MOVE.L #3, D0
      MOVE.L D0, -(SP)
      MOVE.L D, D0
      MOVE.L D0, -(SP)
      BSR    F
      ADDA.L #4, SP
      MOVE.L (SP)+, D0
      MOVE.L D0, D
// chiama servizio exit
// seg. continua a dx
```

```
F: LINK    FP, #-8
// par e var locali
Q: EQU     -12
p: EQU     -8
A: EQU     4
B: EQU     8
   // salva in pila
   MOVE.L #1, D0
   MOVE.L D0, A(FP)
   MOVE.L P(FP), D0
   MOVE.L A(FP), D1
   ADD.L  D0, D1
   MOVE.L D1, Q(FP)
   // ripristina da pila
   UNLK   FP
   RFS
   END    MAIN
```

programma completo

vedi temi d’esame