



Tecniche di traduzione da C ad *assembly* 68000

Note generali

Schema di compilazione da C ad assembly 68K

- Ispirato a GCC
- Fa uso di:
 - banco di registri
 - classi d'istruzioni
 - modi d'indirizzamento e organizzazione del sottoprogramma
- Consiste in
 - Traduzione statement **C**
 - Traduzione invocazioni funzioni **C**
 - Traduzione delle strutture di controllo

Application Binary Interface

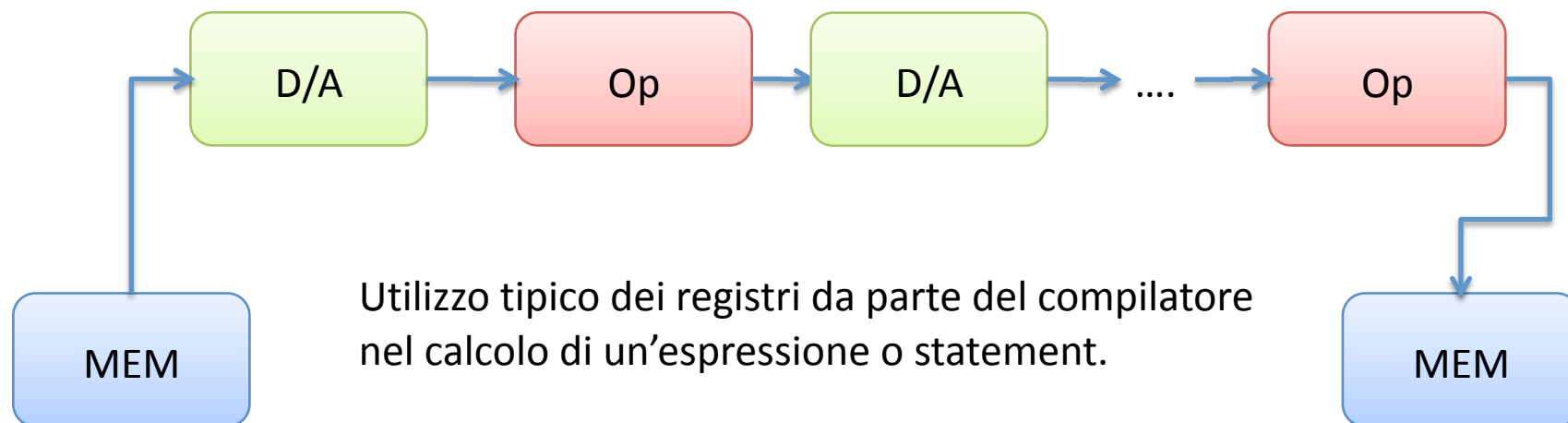
- Specifica di:
 - Dimensione dei tipi di dati
 - Allineamento dati e istruzioni
 - Convenzioni di chiamata delle funzioni (passaggio parametri)
 - Chiamate al sistema operativo 
 - Struttura dell'eseguibile 

Ingombro in memoria delle variabili

- Unita' minima indirizzabile: singolo byte
- In C (Linux):
 - sizeof (char) = 1 byte
 - sizeof (short int) = 2 byte
 - sizeof (int) = 4 byte
 - sizeof (long int) = 8 byte (non utilizzabile su M68k)
 - sizeof (array) = somma ingombri elementi
 - sizeof (struct) = somma ingombri campi
- Non considereremo floating point

Posizione in memoria delle variabili C

- **Globali**: allocate a indirizzo prefissato
- **Locali**: allocate sulla pila
- **Dinamiche**: allocate sullo heap (gestito a sua volta all'interno dell'area **dati** del processo)



Convenzioni per uso dei registri

- **D0-D7** per variabili di tipo carattere o intero
- **A0-A7** per variabili di tipo indirizzo
 - registri **A0-A5**: 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
 - Aggiornato da (MOVE – CMP – ADD SUB NEG ecc
– AND OR NOT ecc)
 - Usato da istruzioni **Bcc** e **DBcc**

Dimensionamento dati

- 8 bit per carattere
- 32 bit per intero
- Le istruzioni assembly devono avere 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
- Es:
 - `MOVE.B D1, D2` // $D2 \leftarrow [D1]$ 8 bit meno signif.
 - `MOVE.W D1, D2` // $D2 \leftarrow [D1]$ 16 bit meno signif.
 - `MOVE.L D1, D2` // $D2 \leftarrow [D1]$ registro completo

Dimensionamento indirizzi

- Registri d'indirizzo (A0-A5):
 - a 16 bit per memoria fisica max da 64 K byte
 - a 32 bit per memoria fisica max da 4 G byte
- Suffissi:
 - **W** per indirizzo da 16 bit indirizzo corto
 - **L** per indirizzo da 32 bit indirizzo lungo
- Es.:
 - MOVEA.**W** A1, A2 // $A2 \leftarrow [A1]$ 16 bit meno signif.
 - MOVEA.**L** A1, A2 // $A2 \leftarrow [A1]$ registro completo

Variabili globali - conversione

```
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  
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 iniziarlo
DC riserva spazio e lo inizializza
il puntatore (di ogni tipo) è equiparato all'intero

Variabile globale - struct

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

tab. dei sim.	
s	1000
s.c	0
s.A	1

```
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
```

- Allocazione spazio per l'intera struct
- Definizione dei simboli di spiazzamento per l'accesso agli elementi della struct
- Utilizzo del '.' per differenziare gli spiazzamenti da altre struct

Tecniche di traduzione da C ad *assembly* 68000

Traduzione di:

Semplici statement

Strutture di controllo

Invocazione delle funzioni

Espressioni

Traduzione semplici statement

Regola base per la traduzione dei semplici statement C

- Assumiamo che lo statement sia una semplice manipolazione della variabile
- Assumere che la variabile sia sempre collocata in memoria
- Come tradurre gli statement C
 - Caricando le variabili nei registri all'inizio dello statement (o non appena serve)
 - Memorizzandole alla fine dello statement
- Ogni variabile cambiata deve essere memorizzata il prima possibile.

Esempio, variabile globale intera

```
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 per puntatore

```
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)

Variabili locali

```
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

Tecniche 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
```

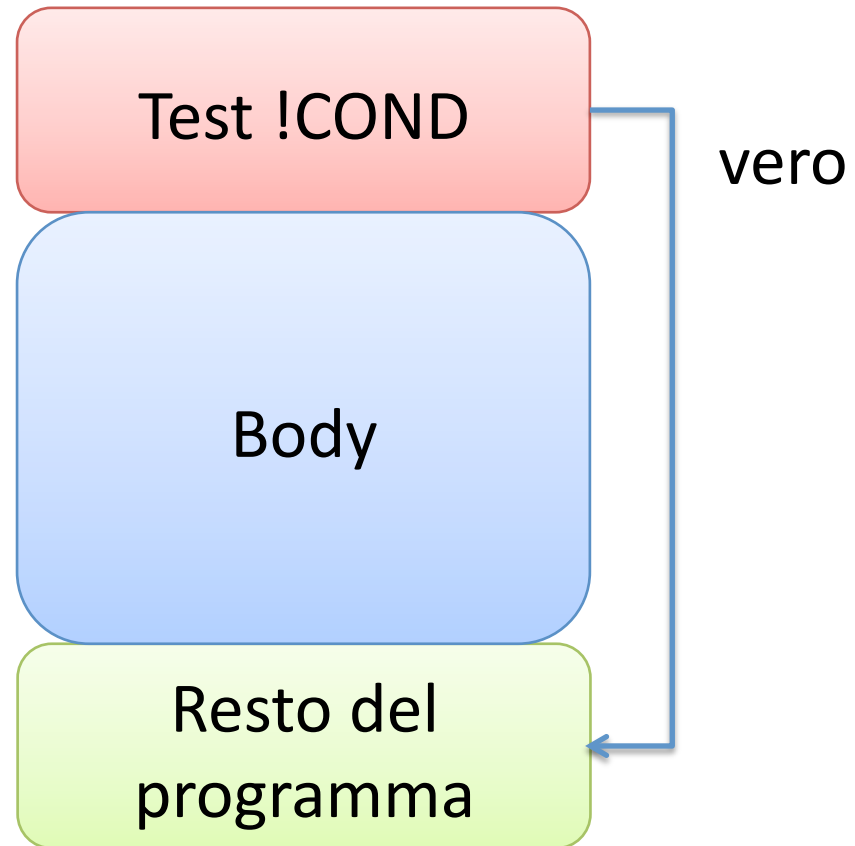
eliminata

elimina tenendo var
in registro dato D0

Traduzione strutture di controllo

If-then

```
if (COND)
{
    ...Body...
}
```



If-then

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

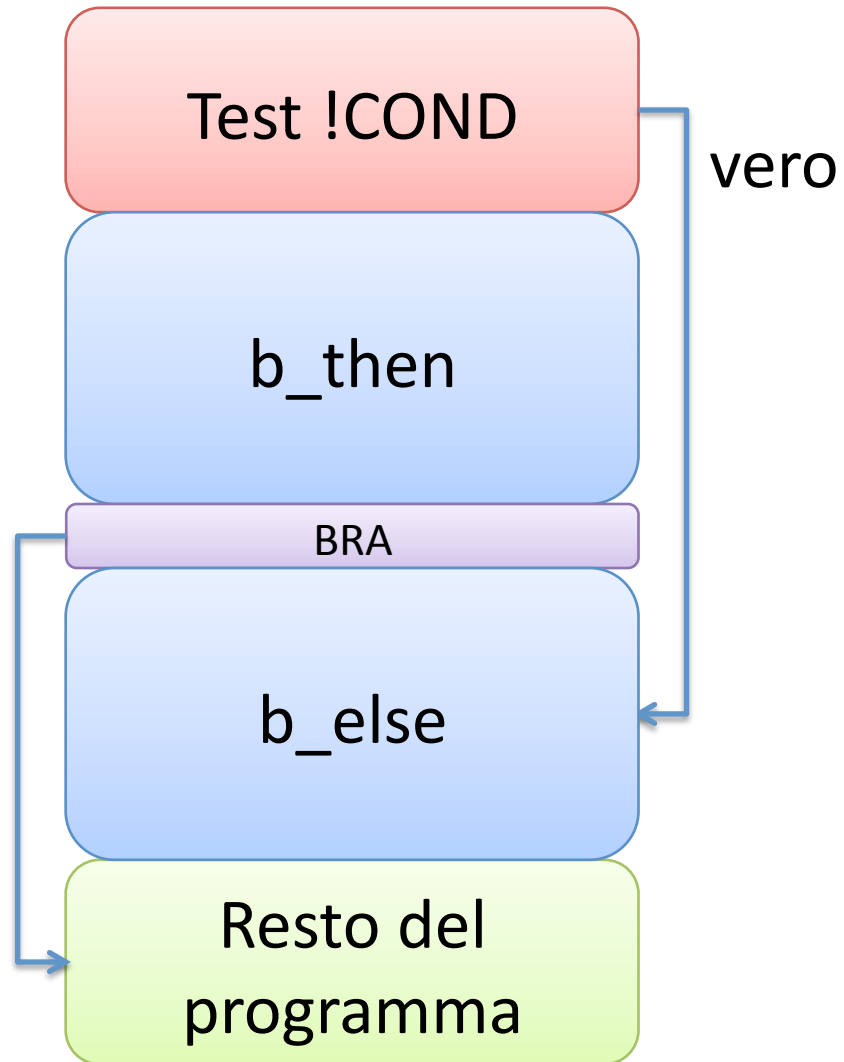
A:	DS.L	1	// riserva mem per a
		...	
	MOVE.L	A , D0	// D0 ← [A]
	CMPI.L	# 5 , D0	// esiti ← [D0] - 5
	BNE	FINE	// se != 0 va' a FINE
		...	// ramo then
FINE:		...	// seguito

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

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

If-then-else

```
if (COND)
{
    ...b_then...
}
else
{
    ...b_else...
}
```



If-then-else

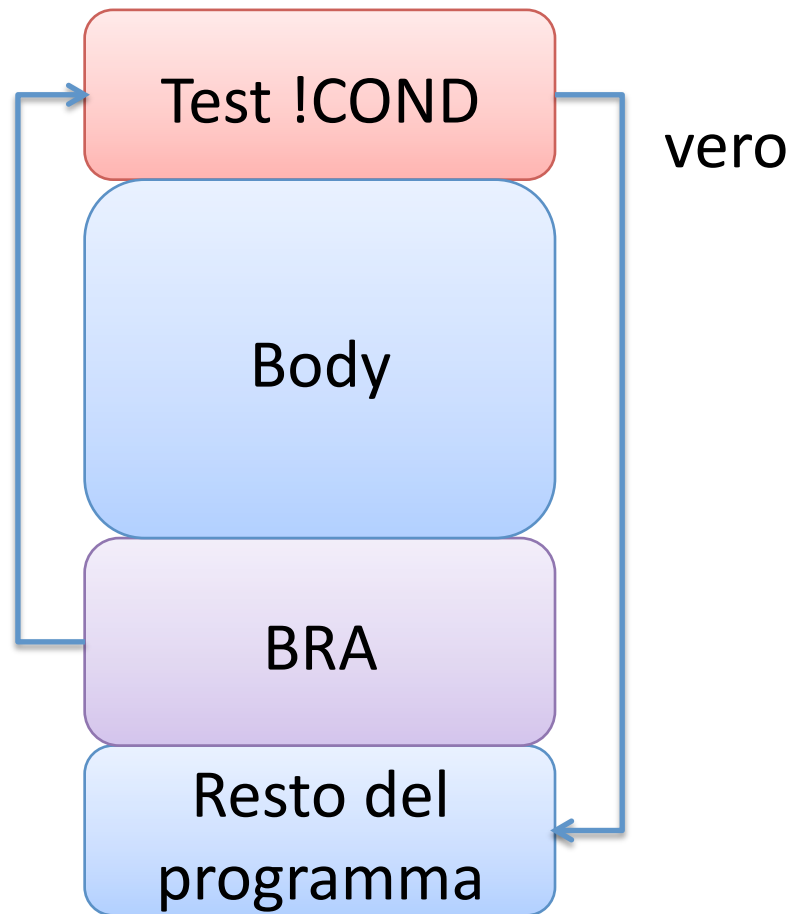
```
// var. globale
int a;
...
// condizione
if (a >= 5) {
    // ramo then
    ...
} else {
    // ramo else
    ...
} /* end if */
... // seguito
```

```
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

```
while(COND)  
{  
    ...Body...  
}
```



While

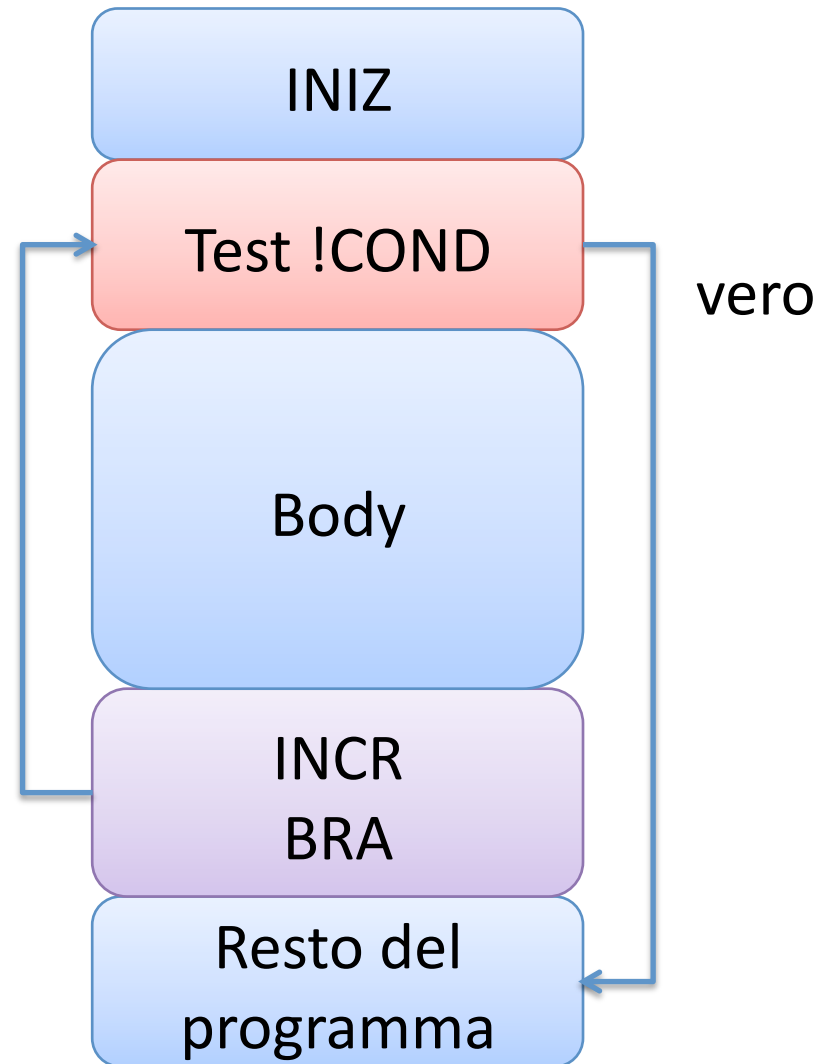
```
// var. globale
int a;
...
// condizione
while (a >= 5)
{
    // corpo
    ...
} /* end while
*/
// seguito
...
```

```
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

For

```
for (INIZ; COND; INCR)
{
    ...Body...
}
```



For

```
// 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
...
MOVE.L  #1, D0 // D0 ← 1
MOVE.L  D0, A  // A ← [D0]
a = 1 | 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]
      | ADDI.L #1, D0 // D0 ← [D0] + 1
a++   | 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