
4 Come viene tradotto in linguaggio macchina un programma C?

I sistemi di calcolo si basano su un certo numero di **astrazioni** che forniscono una visione più semplice del funzionamento della macchina, nascondendo dettagli dell'implementazione che possono essere, almeno in prima battuta, ignorati.

Due delle più importanti astrazioni sono:

- La **memoria**, vista come un grosso array di byte.
- L'**instruction set architecture** (ISA), che definisce:
 - a. lo stato della CPU;
 - b. il formato delle sue istruzioni;
 - c. l'effetto che le istruzioni hanno sullo stato.

Per tradurre codice di alto livello (ad esempio in linguaggio C) in codice macchina, i compilatori si basano sulla descrizione astratta della macchina data dalla sua ISA.

Due delle ISA più diffuse sono:

- IA32, che descrive le architetture della famiglia di processori x86 a 32 bit;
- x86-64, che descrive le architetture della famiglia di processori x86 a 64 bit.

L'x86-64 è ottenuto come estensione dell'IA32, con cui è **retrocompatibile**. Le istruzioni IA32 sono infatti presenti anche nell'x86-64, ma l'x86-64 introduce **nuove istruzioni** non supportate dall'IA32. Programmi scritti in linguaggio macchina per piattaforme IA32 possono essere eseguiti anche su piattaforme x86-64, ma in generale non vale il viceversa. In questa dispensa tratteremo l'IA32.

4.1 Instruction set architecture (ISA) IA32

4.1.1 Tipi di dato macchina

L'IA32 ha sei tipi di dato numerici primitivi (tipi di dato macchina):

Tipo di dato macchina	Rappresen- tazione	Suffisso assembly	Dimensione in byte
Byte	intero	b	1
Word	intero	w	2
Double word	intero	l	4
Single precision	virgola mobile	s	4
Double precision	virgola mobile	l	8

Extended precision	virgola mobile	t	12 (10 usati)
--------------------	----------------	---	---------------

I tipi macchina permettono di rappresentare sia **numeri interi** che **numeri in virgola mobile**. Si noti che il tipo Extended precision richiede 12 byte in IA32. Tuttavia, di questi solo 10 byte (80 bit) sono effettivamente usati.

Ogni tipo ha un corrispondente **suffisso assembly** che, come vedremo, viene usato per denotare il **tipo degli operandi di una istruzione**.

4.1.2 Corrispondenza tra tipi di dato C e tipi di dato macchina

La seguente tabella mostra la corrispondenza tra i tipi di dato primitivi C (interi, numeri in virgola mobile e puntatori) e i tipi di dato primitivi macchina:

Tipo di dato C	Tipo di dato macchina	Suffisso	Dimensione in byte
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Double word	l	4
long long	<i>non supportato</i>	–	8
puntatore	Double word	l	4
float	Single precision	s	4
double	Double precision	l	8
long double	Extended precision	t	12 (10 usati)

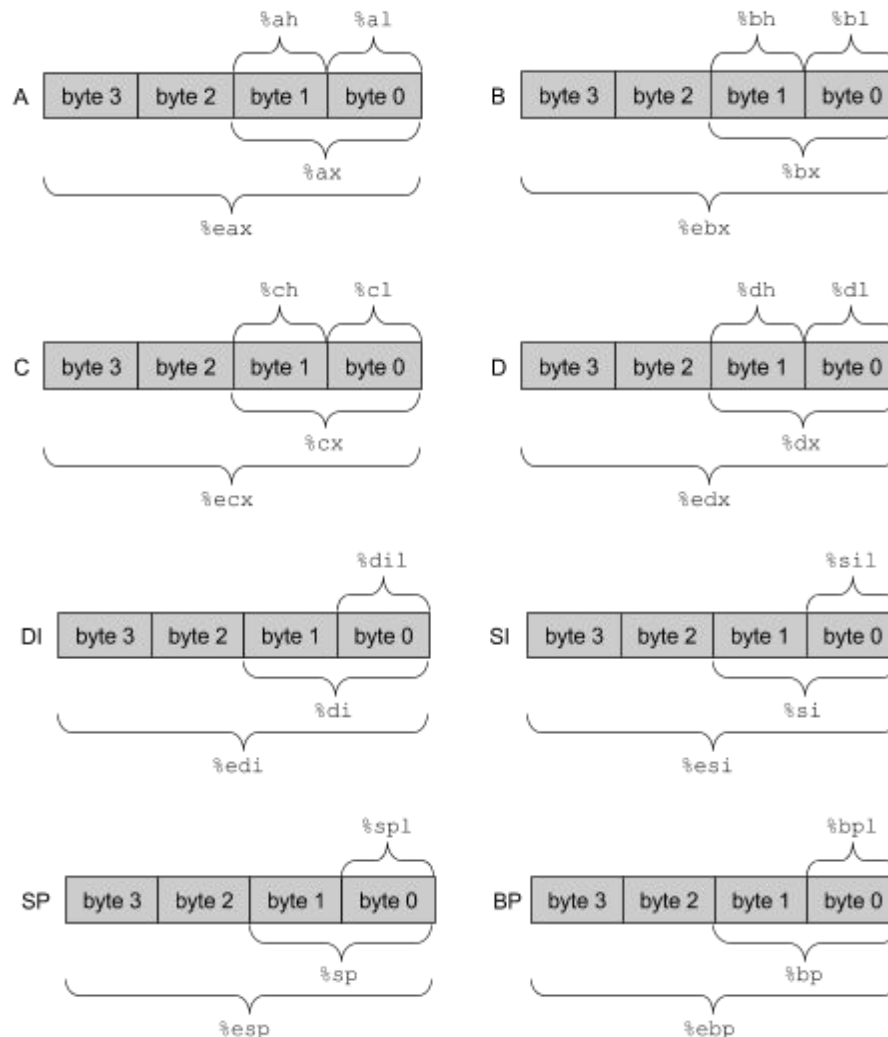
Si noti che il tipo di dato `long long` non è supportato in modo nativo dall'hardware IA32.

Interi con e senza segno hanno il **medesimo tipo macchina** corrispondente: ad esempio, sia `char` che `unsigned char` sono rappresentati come Byte.

4.1.3 Registri

I **registri** sono delle memorie ad altissima velocità a bordo della CPU. In linguaggio assembly, sono identificati mediante dei **nomi simbolici** e possono essere usati in un programma come se fossero variabili.

L'IA32 ha 8 registri interi (A, B, C, D, DI, SI, SP, BP) di dimensione 32 bit (4 byte), di cui i primi 6 possono essere usati come se fossero variabili per memorizzare interi e puntatori:



I registri SP e BP hanno invece un uso particolare che vedremo in seguito. Nella descrizione, byte₀ denota il byte **meno significativo** del registro e byte₃ quello **più significativo**.

Si noti che è possibile accedere a singole parti di un registro utilizzando dei nomi simbolici. Ad esempio, per il registro A:

- %eax denota i 4 byte di A (byte₃, byte₂, byte₁, byte₀)
- %ax denota i due byte meno significativi di A (byte₁ e byte₀)
- %al denota il byte meno significativo di A (byte₀)
- %ah denota il secondo byte meno significativo di A (byte₁)

L'IA32 ha anche altri registri:

- EIP: registro a 32 bit che contiene l'indirizzo della prossima istruzione da eseguire (program counter)
- EFLAGS: registro a 32 bit che contiene informazioni sullo stato del processore

- altri registri per calcoli in virgola mobile e vettoriali che non trattiamo in questa dispensa

4.1.4 Operandi e modi di indirizzamento della memoria

Le istruzioni macchina hanno in genere uno o più **operandi** che definiscono i dati su cui operano. In generale, si ha un **operando sorgente** che specifica un valore di ingresso per l'operazione e un **operando destinazione** che identifica dove deve essere immagazzinato il risultato dell'operazione.

Gli operandi sorgente possono essere di tre tipi:

- *Immediato*: operando immagazzinato insieme all'istruzione stessa;
- *Registro*: operando memorizzato in uno degli 8 registri interi;
- *Memoria*: operando memorizzato in memoria.

Gli operandi destinazione possono essere invece di soli due tipi:

- *Registro*: il risultato dell'operazione viene memorizzato in uno degli 8 registri interi;
- *Memoria*: il risultato dell'operazione viene memorizzato in memoria.

Useremo la seguente notazione:

- Se E è il nome di un registro, $R[E]$ denota il contenuto del registro E ;
- Se x è un indirizzo di memoria, $M_b[x]$ denota dell'oggetto di b byte all'indirizzo x (omettiamo il pedice b quando la dimensione è irrilevante ai fini della descrizione).

Si hanno le seguenti 11 possibili forme di operandi. Per gli operandi di tipo memoria, vi sono vari **modi di indirizzamento** che consentono di accedere alla memoria dopo averne calcolato un indirizzo.

Tipo	Sintassi	Valore denotato	Nome convenzionale
Immediato	$\$imm$	imm	Immediato
Registro	E	$R[E]$	Registro
Memoria	imm	$M[imm]$	Assoluto
Memoria	(E_{base})	$M[R[E_{base}]]$	Indiretto
Memoria	$imm(E_{base})$	$M[imm + R[E_{base}]]$	Base e spiazzamento
Memoria	(E_{base}, E_{indice})	$M[R[E_{base}] + R[E_{indice}]]$	Base e indice
Memoria	$imm(E_{base}, E_{indice})$	$M[imm + R[E_{base}] + R[E_{indice}]]$	Base, indice e spiazzamento
Memoria	(E_{indice}, s)	$M[R[E_{indice}] \cdot s]$	Indice e scala
Memoria	$imm(E_{indice}, s)$	$M[imm + R[E_{indice}] \cdot s]$	Indice, scala e spiazzamento

Memoria	$(E_{base}, E_{indice}, s)$	$M[R[E_{base}] + R[E_{indice}] \cdot s]$	Base, indice e scala
Memoria	$imm(E_{base}, E_{indice}, s)$	$M[imm + R[E_{base}] + R[E_{indice}] \cdot s]$	Base, indice, scala e spiazzamento

Negli indirizzamenti a memoria con indice scalato, il parametro s può assumere solo uno dei valori: 1, 2, 4, 8. Il parametro immediato imm è un valore intero costante a 32 bit, ad esempio -24 (decimale) oppure 0xAF25CB7E (esadecimale).

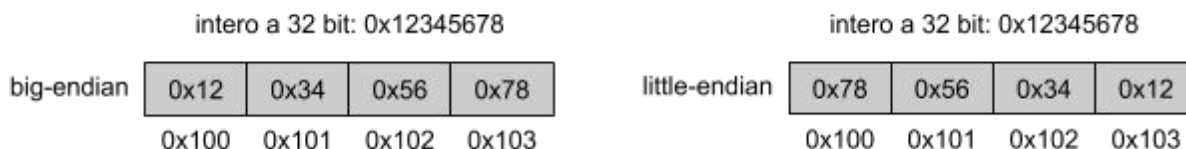
Nel seguito, usiamo la notazione S_n per denotare un operando **sorgente** di n byte, e D_n per denotare un operando **destinazione** di n byte. Omettiamo il pedice quando la dimensione è irrilevante ai fini della descrizione.

4.1.5 Rappresentazione dei numeri in memoria: big-endian vs. little-endian

L'**endianess** di un processore definisce l'**ordine** con cui vengono disposti in **memoria** i **byte** della rappresentazione di un valore numerico:

- **big-endian**: il byte **più** significativo del numero viene posto all'indirizzo più basso;
- **little-endian**: il byte **meno** significativo del numero viene posto all'indirizzo più basso.

Ad esempio, l'intero a 32 bit 0x12345678 viene disposto all'indirizzo 0x100 di memoria con le seguenti sequenze di byte (in esadecimale):



Si noti come nel formato big-endian l'ordine dei byte è lo stesso in cui appare nel letterale numerico che denota il numero, in cui la cifra più significativa appare per prima. Nel little-endian è il contrario.

Esempi di processori big endian sono PowerPC e SPARC. Processori little-endian sono ad esempio quelli della famiglia x86.

4.1.6 Istruzioni di movimento dati

Le istruzioni di movimento dati servono per **copiare byte** da memoria a registro, da registro a registro, e da registro a memoria. Con la notazione $X:Y$ denotiamo la concatenazione delle cifre di X con quelle di Y . Esempio: A3F:C07 denota A3FC07.

4.1.6.1 Stessa dimensione sorgente e destinazione: MOV

Una delle istruzioni più comuni è la `MOV`, dove sorgente e destinazione hanno la stessa dimensione.

Istruzione	Effetto	Descrizione
<code>MOV S, D</code>	$D \leftarrow S$	<i>copia byte da sorgente S a destinazione D</i>
<code>movb S₁, D₁</code>	$D_1 \leftarrow S_1$	copia 1 byte
<code>movw S₂, D₂</code>	$D_2 \leftarrow S_2$	copia 2 byte
<code>movl S₄, D₄</code>	$D_4 \leftarrow S_4$	copia 4 byte

4.1.6.2 Dimensione destinazione maggiore di quella sorgente: `MOVZ`, `MOVS`

Le istruzioni `MOVZ`, e `MOVS` servono per spostare dati da un operando sorgente a un operando destinazione di dimensione maggiore. Servono per effettuare le conversioni di tipi interi senza segno (`MOVZ`) e con segno (`MOVS`).

Istruzione	Effetto	Descrizione
<code>MOVZ S, D</code>	$D \leftarrow \text{ZeroExtend}(S)$	<i>copia byte da sorgente S a destinazione D, riempiendo con zero i byte che D ha in più rispetto a S</i>
<code>movzbw S₁, D₂</code>	$D_2 \leftarrow 0x00:S_1$	copia 1 byte in 2 byte, estendi con zero
<code>movzbl S₁, D₄</code>	$D_4 \leftarrow 0x000000:S_1$	copia 1 byte in 4 byte, estendi con zero
<code>movzwl S₂, D₄</code>	$D_4 \leftarrow 0x0000:S_2$	copia 2 byte in 4 byte, estendi con zero

Esempi:

Si assuma `%eax=0x12341234` e `%ecx=0xABCD0034`:

Istruzione	Risultato (estensione sottolineata)
<code>movzbw %al, %cx</code>	<code>%eax=0x12341234</code> <code>%ecx=0xABCD<u>00</u>34</code>
<code>movzbl %al, %ecx</code>	<code>%eax=0x12341234</code> <code>%ecx=0x<u>000000</u>34</code>
<code>movzwl %ax, %ecx</code>	<code>%eax=0x12341234</code> <code>%ecx=0x<u>0000</u>1234</code>

Vediamo ora l'istruzione `MOVS`:

Istruzione	Effetto	Descrizione
------------	---------	-------------

<i>MOVS S, D</i>	$D \leftarrow \text{SignExtend}(S)$	<i>copia byte da sorgente S a destinazione D, riempiendo con il bit del segno (bit più significativo) di S i byte che D ha in più rispetto a S</i>
<i>movsbw S₁, D₂</i>	$D_2 \leftarrow 0xMM:S_1$	copia 1 byte in 2 byte, estendi con <u>M</u> ='F' se il bit più significativo di S ₁ (bit del segno) è 1 e con <u>M</u> ='0' altrimenti
<i>movsbl S₁, D₄</i>	$D_4 \leftarrow 0xMMMMMM:S_1$	copia 1 byte in 4 byte, estendi con <u>M</u> ='F' se il bit più significativo di S ₁ (bit del segno) è 1 e con <u>M</u> ='0' altrimenti
<i>movswl S₂, D₄</i>	$D_4 \leftarrow 0xMMMM:S_2$	copia 2 byte in 4 byte, estendi con <u>M</u> ='F' se il bit più significativo di S ₂ (bit del segno) è 1 e con <u>M</u> ='0' altrimenti

Esempi:

Si assuma `%eax=0x12341234` e `%ecx=0xABCDE1E2`:

Istruzione	Risultato (estensione sottolineata)
<code>movsbw %al, %cx</code>	<code>%eax=0x123412<u>34</u></code> <code>%ecx=0xABCD<u>0034</u></code>
<code>movsbl %al, %ecx</code>	<code>%eax=0x123412<u>34</u></code> <code>%ecx=0x<u>00000034</u></code>
<code>movswl %ax, %ecx</code>	<code>%eax=0x1234<u>1234</u></code> <code>%ecx=0x<u>00001234</u></code>
<code>movsbw %cl, %ax</code>	<code>%ecx=0xABCD<u>E1E2</u></code> <code>%eax=0x1234<u>FFE2</u></code>
<code>movsbl %cl, %eax</code>	<code>%ecx=0xABCD<u>E1E2</u></code> <code>%eax=0x<u>FFFFFFE2</u></code>
<code>movswl %cx, %eax</code>	<code>%ecx=0xABCD<u>E1E2</u></code> <code>%eax=0x<u>FFFE1E2</u></code>

4.1.6.3 Movimento dati da/verso la stack: PUSH, POP

Le istruzioni `PUSH`, e `POP` servono per spostare dati da un operando sorgente verso la cima della stack (`PUSH`) e dalla cima della stack verso un operando destinazione (`POP`):

Istruzione	Effetto	Descrizione
<code>pushl S₄</code>	$R[\%esp] \leftarrow R[\%esp] - 4$ $M[R[\%esp]] \leftarrow S_4$	copia l'operando di 4 byte S sulla cima della stack
<code>popl D₄</code>	$D_4 \leftarrow M[R[\%esp]]$ $R[\%esp] \leftarrow R[\%esp] + 4$	togli i 4 byte dalla cima della stack e copiali nell'operando D

4.1.7 Istruzioni aritmetico-logiche

Le seguenti istruzioni IA32 servono per effettuare operazioni su interi a 1, 2 e 4 byte:

Istruzione	Effetto	Descrizione
INC D	$D \leftarrow D+1$	incrementa destinazione
DEC D	$D \leftarrow D-1$	decrementa destinazione
NEG D	$D \leftarrow -D$	inverte segno destinazione
NOT D	$D \leftarrow \sim D$	complementa a 1 destinazione
ADD S, D	$D \leftarrow D+S$	aggiungi sorgente a destinazione e risultato in destinazione
SUB S, D	$D \leftarrow D-S$	sottrai sorgente da destinazione e risultato in destinazione
IMUL S, D	$D \leftarrow D*S$	moltiplica sorgente con destinazione e risultato in destinazione, la destinazione deve essere un registro
XOR S, D	$D \leftarrow D \wedge S$	or esclusivo sorgente con destinazione e risultato in destinazione
OR S, D	$D \leftarrow D \vee S$	or sorgente con destinazione e risultato in destinazione
AND S, D	$D \leftarrow D \& S$	and sorgente con destinazione e risultato in destinazione

Omettiamo per il momento istruzioni più complesse come quelle che effettuano divisioni.

4.1.7.1 L'istruzione LEA (load effective address)

L'istruzione LEA consente di sfruttare la flessibilità data dai modi di indirizzamento a memoria per calcolare espressioni aritmetiche che coinvolgono somme e prodotti su indirizzi o interi.

Istruzione	Effetto	Descrizione
leal S, D ₄	$D_4 \leftarrow \&S$	Calcola l'indirizzo effettivo specificato dall'operando di tipo memoria S e lo scrive in D

Si noti che `leal`, diversamente da `movl`, non effettua un accesso a memoria sull'operando sorgente. L'istruzione `leal` calcola infatti l'indirizzo effettivo dell'operando sorgente, senza però accedere in memoria a quell'indirizzo.

Esempi.

Si assuma `%eax=0x100`, `%ecx=0x7` e `M4[0x100]=0xCAFE`

Istruzione	Effetto	Risultato
<code>movl (%eax), %edx</code>	$R[\%edx] \leftarrow M_4[R[\%eax]]$	<code>%edx=0xCAFE</code>
<code>leal (%eax), %edx</code>	$R[\%edx] \leftarrow R[\%eax]$	<code>%edx=0x100</code>

Si noti la differenza fra `leal` e `movl` che abbiamo discusso sopra. Si considerino inoltre i seguenti altri esempi:

Istruzione	Effetto	Risultato
<code>leal (%eax, %ecx), %edx</code>	$R[\%edx] \leftarrow R[\%eax] + R[\%ecx]$	<code>%edx=0x107</code>
<code>leal -3(%eax, %ecx), %edx</code>	$R[\%edx] \leftarrow R[\%eax] + R[\%ecx] - 3$	<code>%edx=0x104</code>
<code>leal -3(%eax, %ecx, 2), %edx</code>	$R[\%edx] \leftarrow R[\%eax] + R[\%ecx] \cdot 2 - 3$	<code>%edx=0x10B</code>
<code>leal (%eax, %ecx, 2), %edx</code>	$R[\%edx] \leftarrow R[\%eax] + R[\%ecx] \cdot 2$	<code>%edx=0x10E</code>
<code>leal (%ecx, 4), %edx</code>	$R[\%edx] \leftarrow R[\%ecx] \cdot 4$	<code>%edx=0x1C</code>

L'istruzione `leal` viene usata per scrivere **programmi più veloci** e viene sfruttata tipicamente per due scopi:

1. calcolare l'indirizzo effettivo di un oggetto in memoria una sola volta, per poi usarlo più volte;
2. calcolare **espressioni aritmetiche su interi o puntatori** usando una sola istruzione.

Si noti infatti che, sebbene sia stata pensata per calcolare indirizzi di memoria, la `leal` può essere usata per calcolare espressioni intere che non rappresentano indirizzi.

Esempio.

Si consideri il seguente frammento di programma C:

```
int x=10;
int y=20;
int z=x+y*4-7;
```

Riformuliamo il frammento in modo che ogni operazione aritmetica abbia la forma: $a = a \text{ op } b$, ottenendo il seguente codice equivalente, la corrispondente traduzione in codice IA32 e una versione ottimizzata del codice IA32 basata sull'istruzione `leal`:

Codice C	Codice IA32	Codice IA32 ottimizzato
<pre>int x=10; // x è in %eax int y=20; // y è in %ecx</pre>	<pre>movl \$10,%eax movl \$20,%ecx</pre>	<pre>movl \$10,%eax movl \$20,%ecx</pre>

<pre>int z=y; // z è in %edx z=z*4; z=z-7; z=z+x;</pre>	<pre>movl %ecx,%edx imull \$4,%edx addl \$-7,%edx addl %eax,%edx</pre>	<pre>leal -7(%eax,%ecx,4),%edx</pre>
--	--	--------------------------------------

Si noti che, se l'espressione da calcolare fosse stata $x+y*5-7$, non sarebbe stato possibile usare la `leal`: infatti il fattore moltiplicativo nei vari modi di indirizzamento a memoria (scala) può essere solo 1, 2, 4, 8. Non tutte le espressioni aritmetiche possono quindi essere calcolate con la `leal`.

4.1.8 Istruzioni di salto

Normalmente, il flusso del controllo di un programma procede in modo sequenziale, eseguendo le istruzioni nell'ordine in cui appaiono in memoria. Ogni volta che un'istruzione *I* viene eseguita, il registro EIP (instruction pointer), che punta alla prossima istruzione da eseguire, viene incrementato automaticamente del numero di byte occupati dall'istruzione *I*.

Vi sono tuttavia istruzioni, chiamate **istruzioni di salto**, che permettono di alterare il flusso del controllo, modificando il contenuto del registro EIP in modo che l'esecuzione non prosegua con istruzione successiva, ma con un'altra che inizia ad un indirizzo diverso.

Vi sono tre tipi di istruzioni di salto:

1. salti **incondizionati**: il registro EIP viene sovrascritto con l'indirizzo di memoria dell'istruzione a cui si vuole saltare;
2. salti **condizionati**: il registro EIP viene sovrascritto con l'indirizzo di memoria dell'istruzione a cui si vuole saltare, ma solo se è verificata una determinata condizione sui dati;
3. **chiamata e ritorno** da funzione (che vedremo in seguito).

4.1.8.1 Salti incondizionati: JMP

Le istruzioni di salto incondizionato possono essere di tipo diretto o indiretto:

Istruzione	Effetto	Nota
<code>jmp etichetta</code>	$R[\%eip] \leftarrow \text{indirizzo associato all'etichetta}$	salto diretto
<code>jmp *S</code>	$R[\%eip] \leftarrow S$	salto indiretto

Esempio.

Si consideri il seguente frammento di programma x86:

<pre>movl \$0, %eax L: incl %eax jmp L</pre>
--

Il programma esegue dapprima l'istruzione `movl`, poi `incl`. Quando incontra la `jmp` ritorna ad eseguire la `incl`. Infatti l'etichetta `L` (introdotta con la sintassi `L:`) denota l'indirizzo dell'istruzione `incl`. Si ha quindi un ciclo infinito.

Esempio.

Si consideri il seguente frammento di programma x86:

```
jmp *(%eax)
```

Il programma salta all'indirizzo effettivo denotato dall'operando `(%eax)`. L'operazione effettuata è quindi: $\%eip \leftarrow M[R[\%eax]]$.

4.1.8.2 Salti condizionati e condition code: Jcc, CMP

Le istruzioni di salto condizionato consentono di modificare il registro EIP, e quindi alterare il normale flusso sequenziale del controllo dell'esecuzione, solo se una determinata condizione è soddisfatta. Il test viene effettuato in base al contenuto di un registro particolare chiamato registro dei FLAG, che viene modificato come effetto collaterale dell'esecuzione della maggior parte delle istruzioni aritmetico-logiche.

Un salto condizionato avviene in due passi:

1. un'operazione aritmetico-logica effettua un'operazione sui dati
2. in base all'esito dell'operazione, l'istruzione di salto condizionato salta o meno a un'etichetta

Il registro dei FLAG contiene in particolare quattro codici di condizione (condition code) booleani:

1. **ZF** (zero flag): viene posto a 1 se l'ultima operazione aritmetico-logica ha prodotto un valore zero e 0 se ha prodotto un valore diverso da zero;
2. **SF** (sign flag): viene posto a 1 se l'ultima operazione aritmetico-logica ha prodotto un valore negativo e 0 se ha prodotto un valore non negativo;
3. **CF** (carry flag): viene posto a 1 se l'ultima operazione aritmetico-logica ha generato un riporto e 0 altrimenti;
4. **OF** (overflow flag): viene posto a 1 se l'ultima operazione aritmetico-logica ha generato un overflow e 0 altrimenti.

La forma generale di una istruzione di salto condizionato è la seguente:

Istruzione	Effetto	Nota
Jcc etichetta	if (condizione) $R[\%eip] \leftarrow$ indirizzo associato all'etichetta	salto condizionato se la condizione associata al suffisso cc è verificata

La seguente tabella riporta le possibili condizioni su cui è possibile saltare e i possibili codici suffissi di istruzione:

Suffisso cc	Sinonimo	Condizione ⁵	Significato
e	z	ZF	Uguale (o zero)
ne	nz	$\sim ZF$	Diverso (o non zero)
s		SF	Negativo
ns		$\sim SF$	Non negativo
g	nle	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Maggiore (g=greater) con segno
ge	nl	$\sim (SF \wedge OF)$	Maggiore o uguale (ge=greater or equal) con segno
l	nge	$SF \wedge OF$	Minore (l=less) con segno
le	ng	$(SF \wedge OF) \ \ ZF$	Minore o uguale con segno
a	nbe	$\sim CF \ \& \ \sim ZF$	Maggiore (a=above) senza segno
ae	nb	$\sim CF$	Maggiore o uguale (ae=above or equal) senza segno
b	nae	CF	Minore (b=below) senza segno
be	na	$CF \ \ ZF$	Minore o uguale (be=below or equal) senza segno

Si noti che i confronti maggiore/minore sono diversi a seconda che si intenda considerare o meno il segno dei valori confrontati.

Esempio 1.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui il registro `eax` è trattato come se fosse una variabile:

<pre>decl %eax jz L</pre>	<pre>eax--; if (eax == 0) goto L;</pre>
---------------------------	---

La prima operazione decrementa il contenuto del registro `eax`. Se `eax` diventa zero, allora l'istruzione `jz` salterà all'etichetta `L`.

Esempio 2.

⁵ Si ricordi che: \sim denota la negazione logica, $\&$ l'and, $|$ l'or, e \wedge l'or esclusivo (xor). La condizione in funzione dei condition code ZF, SF, CF, OF viene riportata per completezza e non è in generale utile per il programmatore.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui i registri sono trattati come se fossero variabili:

<pre>subl %ebx, %eax je L</pre>	<pre>temp = eax eax = eax - ebx if (temp == ebx) goto L</pre>
---------------------------------	---

La prima operazione calcola $R[\%eax] - R[\%ebx]$ e scrive il risultato in $R[\%eax]$. Si noti che il risultato della sottrazione è zero se e solo se i due registri sono uguali. Pertanto, l'istruzione `je` salterà all'etichetta `L` se e solo se i due registri sono uguali prima della `SUB`.

Osserviamo che per effettuare un salto condizionato rispetto al contenuto di due registri abbiamo dovuto modificarne uno: infatti la `SUB` modifica l'operando destinazione. Per ovviare a questo problema il set IA32 prevede una istruzione di sottrazione che non modifica l'operando destinazione, pensata specificamente per essere usata nei confronti:

Istruzione	Effetto	Nota
<code>CMP S, D</code>	calcola $D - S$	la differenza calcolata viene usata per modificare i condition code e poi va persa

La seguente tabella riporta la condizione testata per ciascun prefisso assumendo di aver appena effettuato una operazione `CMP S, D`:

Suffisso cc	Sinonimo	Condizione testata dopo istruzione <code>CMP S, D</code>	Ovvero
e	z	$D - S == 0$	$D == S$
ne	nz	$D - S != 0$	$D != S$
g	nle	$D - S > 0$	$D > S$
ge	nl	$D - S >= 0$	$D >= S$
l	nge	$D - S < 0$	$D < S$
le	ng	$D - S <= 0$	$D <= S$
a	nbe	$(\text{unsigned}) D - (\text{unsigned}) S > 0$	$(\text{unsigned}) D > (\text{unsigned}) S$
ae	nb	$(\text{unsigned}) D - (\text{unsigned}) S >= 0$	$(\text{unsigned}) D >= (\text{unsigned}) S$
b	nae	$(\text{unsigned}) D - (\text{unsigned}) S < 0$	$(\text{unsigned}) D < (\text{unsigned}) S$
be	na	$(\text{unsigned}) D - (\text{unsigned}) S <= 0$	$(\text{unsigned}) D <= (\text{unsigned}) S$

Esempio 3.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui i registri sono trattati come se fossero variabili:

<pre>cmpl %ebx, %eax jle L</pre>	<pre>if (eax <= ebx) goto L;</pre>
----------------------------------	---------------------------------------

La prima operazione calcola la differenza $R[\%eax] - R[\%ebx]$. La seconda salta se $R[\%eax] - R[\%ebx] \leq 0$.

4.1.8.3 Chiamata e ritorno da funzione: CALL e RET

Un ulteriore tipo di istruzione di salto è quello relativo alle chiamate e ritorno da funzione:

Istruzione	Effetto	Nota
CALL S	$R[\%esp] \leftarrow R[\%esp] - 4$ $M[R[\%esp]] \leftarrow R[\%eip]$ $R[\%eip] \leftarrow S$	Chiamata a funzione: mette in stack l'indirizzo dell'istruzione successiva alla CALL (indirizzo di ritorno) e salta all'indirizzo specificato dall'operando S
RET	$R[\%eip] \leftarrow M[R[\%esp]]$ $R[\%esp] \leftarrow R[\%esp] + 4$	Ritorno da funzione: toglie dalla stack l'indirizzo di ritorno e lo scrive in EIP

Esempio.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui i registri sono trattati come se fossero variabili:

<pre>call f imull \$3, %eax ... f: movl \$2, %eax ret</pre>	<pre>f(); eax = eax*3; ... void f() { eax = 2; }</pre>
---	--

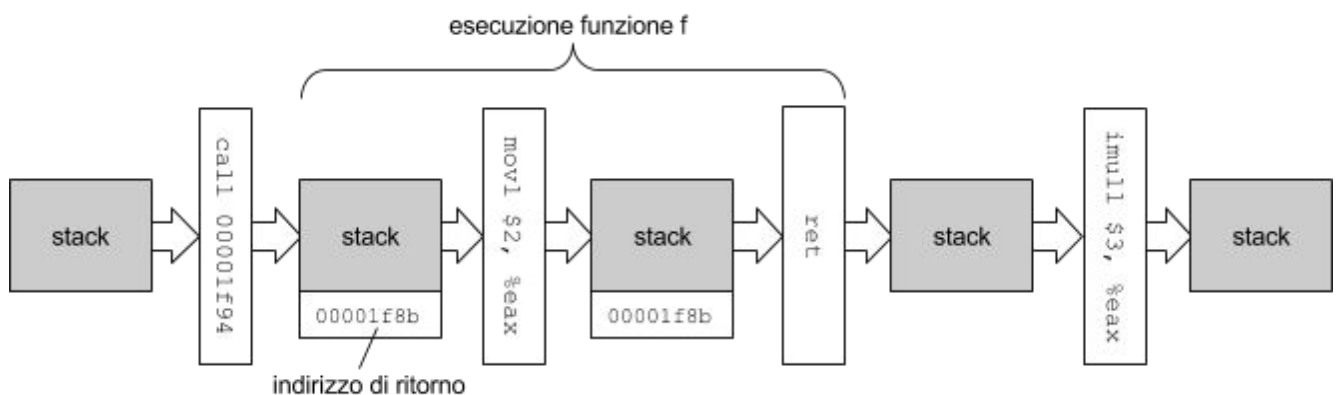
Immaginiamo che il programma sia disposto in memoria ai seguenti indirizzi:

00001f86	call	00001f94	; chiama f
00001f8b	imull	\$3, %eax	
00001f8e	...		
00001f94	movl	\$2, %eax	
00001f99	ret		

Eseguendo le istruzioni a partire dall'indirizzo 00001f86, il flusso delle istruzioni e il loro effetto sui principali registri usati è il seguente:

	%eip (prima)	%eax (prima)	istruzione eseguita	%eip (dopo)	%eax dopo
1	00001f86	-	call 00001f94	00001f94	-
2	00001f94	-	movl \$2, %eax	00001f99	00000002
3	00001f99	00000002	ret	00001f8b	00000002
4	00001f8b	00000002	imull \$3, %eax	00001f8e	00000006

Analizziamo ora il contenuto della stack prima e dopo ogni istruzione:



Si noti che la *CALL* deposita in stack l'indirizzo dell'istruzione successiva, in modo che la *RET* possa proseguire da quella istruzione una volta terminata la chiamata della funzione.

4.1.9 Altre istruzioni e vincoli sulle istruzioni

4.1.9.1 Istruzioni di assegnamento condizionato: *CMOVcc*

L'istruzione *CMOVcc* consente di effettuare degli assegnamenti solo se una determinata condizione è verificata. L'istruzione si basa sulle medesime condizioni della *Jcc*, salvo che invece di saltare, copia l'operando sorgente in quello destinazione.

Istruzione	Effetto	Nota
<i>CMOVcc S,D</i>	if (condizione) $D \leftarrow S$	se la condizione associata al suffisso <i>cc</i> è verificata, copia la sorgente nella destinazione

L'istruzione semplifica alcune operazioni condizionali riducendo il numero di istruzioni richieste. Diversamente dalla *MOV*, l'operando **sorgente** di una *CMOVcc* **non può essere un operando immediato**, la **destinazione deve essere un registro** e solo operandi a 16 e 32 bit sono supportati.

Esempio.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui i registri sono trattati come se fossero variabili:

<pre>cmpl %ecx, %eax cmovgl %eax, %ecx</pre>	<pre>if (eax > ecx) ecx = eax;</pre>
--	---

La prima istruzione calcola $R[\%eax] - R[\%ecx]$. La seconda sovrascrive $R[\%ecx]$ con $R[\%eax]$ se $R[\%eax] > R[\%ecx]$.

4.1.9.2 Altre istruzioni di confronto: TEST

Nello stesso spirito della `CMP`, che corrisponde a una `SUB` in cui non viene modificato l'operando destinazione, l'istruzione `TEST` è identica a una `AND`, tranne che non modifica l'operando destinazione:

Istruzione	Effetto	Nota
<code>TEST S, D</code>	calcola $S \& D$	l'and bit a bit fra gli operandi calcolato viene usato per modificare i condition code e poi va perso

Esempio.

L'istruzione `TEST` può essere usata al posto della `CMP` ad esempio per verificare se un registro è zero o meno:

<pre>testl %eax, %eax jz L</pre>	<pre>cmpl \$0, %eax jz L</pre>	<pre>if (eax==0) goto L</pre>
----------------------------------	--------------------------------	-------------------------------

Si noti che l'AND di un valore con se stesso è zero se e solo se il valore è zero.

4.1.9.3 Altre istruzioni di assegnamento: SETcc

L'istruzione `SETcc` permette di assegnare i valori 0 o 1 a un registro a 8 bit o a un byte di memoria, a seconda che una data condizione sul registro `EFLAGS` sia verificata:

Istruzione	Effetto	Nota
<code>SETcc D₁</code>	$D_1 \leftarrow \text{condizione}$	se la condizione associata al suffisso <code>cc</code> è verificata, scrive 1 in D_1 , altrimenti scrive 0

Esempio.

L'istruzione `SETcc` può essere usata al posto della `Jcc` ad esempio per calcolare il valore di un'espressione booleana:

<code>cmpl \$0, %eax</code> <code>setl %dl</code>	<code>dl = (eax < 0)</code>
--	--------------------------------

4.1.9.4 Vincoli sulle istruzioni

Alcune istruzioni hanno dei vincoli sugli operandi che possono prendere. Elenchiamo i vincoli più comuni:

Istruzione	Vincolo	Esempio
IMUL	la destinazione deve essere un registro	<code>imull \$2, %eax</code> # ok <code>imull \$2, (%edi)</code> # errore
CMOVcc	la sorgente non può essere un immediato	<code>cmovgel %ecx, %eax</code> # ok <code>cmovgel \$2, %eax</code> # errore
	la destinazione deve essere un registro	<code>cmovgel %eax, %eax</code> # ok <code>cmovgel %eax, (%edi)</code> # errore
	solo operandi 16 o 32 bit	<code>cmovgeb %cl, %al</code> # errore
TEST e CMP	secondo operando ("destinazione") non può essere immediato	<code>test %eax, %ecx</code> # ok <code>test (%eax), \$2</code> # errore

4.2 Traduzione dei costrutti C in assembly IA32

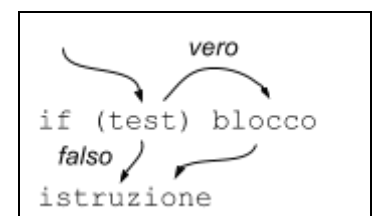
In questo paragrafo vediamo come i compilatori moderni come `gcc` traducono i costrutti del linguaggio C in codice IA32. Si noti come lo stesso programma C potrebbe essere tradotto in assembly in tanti modi diversi. Versioni diverse del compilatore oppure livelli di ottimizzazione diversi portano a codice assembly diverso. Per indicare la traduzione di un frammento di codice `x` in assembly IA32, useremo la notazione `IA32(x)`.

4.2.1 Istruzioni condizionali

Le istruzioni ed espressioni condizionali vengono normalmente basate sulle istruzioni di salto condizionato. In alcuni casi è possibile usare l'istruzione di movimento dati condizionale (`CMOV`).

4.2.1.1 Istruzione `if`

Consideriamo lo schema generale di una istruzione `if`. Se il test effettuato è vero, viene eseguito il blocco e si riprende dall'istruzione successiva, altrimenti si prosegue direttamente con l'istruzione successiva.



L'istruzione `if` può essere tradotta come segue:

C da tradurre	C equivalente	Traduzione IA32
<pre>if (test) blocco istruzione</pre>	<pre>if (!test) goto L; blocco; L: istruzione;</pre>	<pre>IA32(test) Jcc L IA32(blocco) L: IA32(istruzione)</pre>

Si noti che l'`if` viene realizzato effettuando un salto che evita di eseguire il blocco dell'`if` se il test non è soddisfatto. Si salta quindi su `!test` e non su `test`.

Esempio.

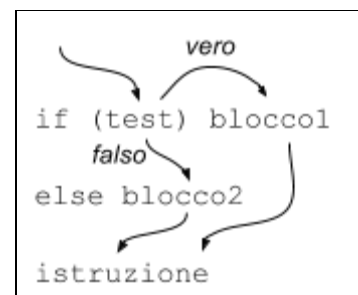
Consideriamo il seguente frammento di programma C con variabili intere senza segno e assumiamo che la variabile `a` sia tenuta nel registro `eax`, `b` in `ebx` e `c` in `ecx`:

C da tradurre	C equivalente	Traduzione IA32
<pre>if (a>b) c=10; c++;</pre>	<pre>if (a<=b) goto L; c=10; L: c++;</pre>	<pre>cmpl %ebx, %eax jbe L movl \$10, %ecx L: incl %ecx</pre>

Notiamo che il test `a<=b` su variabili senza segno viene realizzato calcolando la differenza `R[%eax]-R[%ebx]` con la `CMP` e saltando se il risultato è `<=0` (suffisso `be`=below or equal, confronto senza segno).

4.2.1.1 Istruzione `if...else`

Consideriamo lo schema generale di una istruzione `if...else`. Se il test effettuato è vero, viene eseguito il blocco 1 e si riprende dall'istruzione successiva all'`if...else`, altrimenti si esegue il blocco 2 e si riprende dall'istruzione successiva all'`if...else`.



L'istruzione `if...else` può essere tradotta come segue:

C da tradurre	C equivalente ⁶	Traduzione IA32
<pre>if (test) blocco1</pre>	<pre>if (!test) goto E; blocco1</pre>	<pre>IA32(test) Jcc E IA32(blocco1)</pre>

⁶ Si noti che in C il costrutto `if...else` può essere riscritto in termini di `if` e `goto`.

else blocco2 istruzione	goto F; E: blocco2 F: istruzione;	jmp F E: IA32(blocco2) F: IA32(istruzione)
----------------------------	---	--

Si noti che l'if...else viene realizzato effettuando un salto al blocco 2 che evita di eseguire il blocco 1 se il test non è soddisfatto. Alla fine del blocco 1 c'è un salto incondizionato che evita di eseguire il blocco 2 se il blocco 1 è stato eseguito. Questo realizza la mutua esclusione tra i blocchi eseguiti.

Esempio.

Consideriamo il seguente frammento di programma C con variabili intere con segno e assumiamo che la variabile a sia tenuta nel registro eax, b in ebx e c in ecx:

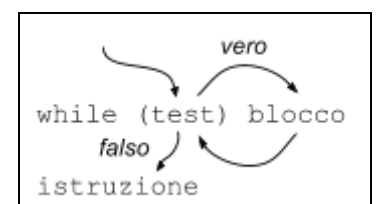
C da tradurre	C equivalente	Traduzione IA32
if (a<=b) c=10; else c=20; c++;	if (a>b) goto E; c=10; goto F; E: c=20; F: c++;	cmpl %ebx, %eax jg L movl \$10, %ecx jmp F E: movl \$20, %ecx F: incl %ecx

Notiamo che il test $a > b$ su variabili con segno viene realizzato calcolando la differenza $R[\text{eax}] - R[\text{ebx}]$ con la CMP e saltando se il risultato è > 0 (suffisso g=greater, confronto con segno).

4.2.2 Cicli

4.2.2.1 Istruzione while

Consideriamo lo schema generale di una istruzione while. Se il test effettuato è vero, viene eseguito il blocco e si ritorna al test, altrimenti si prosegue con l'istruzione successiva al while.



L'istruzione while può essere tradotta come segue:

C da tradurre	C equivalente ⁷	Traduzione IA32
while (test) blocco istruzione	L: if (!test) goto E; blocco; goto L; E: istruzione;	L: IA32(test) Jcc E IA32(blocco) jmp L E: IA32(istruzione)

⁷ Si noti che in C il costrutto while può essere riscritto in termini di if e goto.

Si noti che il `while` è del tutto simile all'`if`, tranne che dopo l'esecuzione del blocco non si prosegue all'istruzione successiva, ma si torna al test.

Esempio.

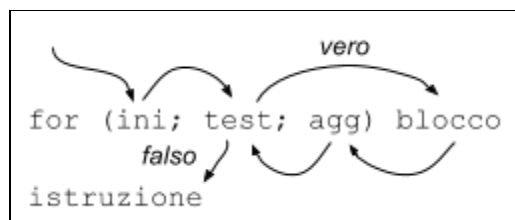
Consideriamo il seguente frammento di programma C con variabili intere senza segno e assumiamo che la variabile `a` sia tenuta nel registro `eax`, `b` in `ebx` e `c` in `ecx`:

C da tradurre	C equivalente	Traduzione IA32
<pre>a=1; c=0; while (a<=b) { c+=a; a++; }</pre>	<pre>a=1; c=0; L: if (a>b) goto E; c+=a; a++; goto L; E:</pre>	<pre>movl \$1, %eax movl \$0, %ecx L: cmpl %ebx, %eax ja E addl %eax, %ecx incl %eax jmp L E:</pre>

Il programma calcola in `c` la somma dei primi `b` interi, cioè $c \leftarrow 1+2+3+\dots+b$.

4.2.2.1 Istruzione `for`

Consideriamo lo schema generale di una istruzione `for`. Si esegue dapprima l'inizializzazione e poi si effettua il test. Se il test effettuato è vero, viene eseguito il blocco, si esegue l'aggiornamento, e si ritorna al test, altrimenti si prosegue con l'istruzione successiva al `for`.



L'istruzione `for` può essere tradotta come segue:

C da tradurre	C equivalente ⁸	Traduzione IA32
<pre>for (ini;test;agg) blocco istruzione</pre>	<pre>ini; L: if (!test) goto E; blocco; agg; goto L; E: istruzione</pre>	<pre>IA32(ini) L: IA32(test) Jcc E; IA32(blocco) IA32(agg) jmp L E: IA32(istruzione)</pre>

Esempio.

Consideriamo il seguente frammento di programma C con variabili intere senza segno e assumiamo che la variabile `a` sia tenuta nel registro `eax`, `b` in `ebx` e `c` in `ecx`:

⁸ Si noti che in C il costrutto `for` può essere riscritto in termini di `if` e `goto`.

C da tradurre	C equivalente	Traduzione IA32
<pre>c=0; for (a=1; a<=b; a++) c+=a;</pre>	<pre>c=0; a=1; L: if (a>b) goto E; c+=a; a++; goto L; E:</pre>	<pre>movl \$0, %ecx movl \$1, %eax L: cmpl %ebx, %eax ja E addl %eax, %ecx incl %eax jmp L E:</pre>

Il programma calcola in `c` la somma dei primi `b` interi, cioè $c \leftarrow 1+2+3+\dots+b$, ed è del tutto equivalente a quello visto come esempio per il `while`.

4.2.3 Funzioni

Una funzione C è normalmente tradotta in assembly IA32 come una sequenza di istruzioni terminate da una `RET` e viene invocata mediante l'istruzione `CALL`. Durante una chiamata a funzione, la funzione che ha effettuato l'invocazione viene detta **chiamante** (caller) e quella invocata viene detta **chiamato** (callee).

Le **convenzioni** relative alla traduzione delle funzioni, del passaggio dei parametri e della restituzione dei valori che vedremo in questo paragrafo non sono specificate dall'ISA IA32, ma sono conformi con la [System V Application Binary Interface](#) (ABI), che descrive uno standard diffuso (es. Mac OS X e Linux) usato nella creazione dei file oggetto e nell'orchestrazione dell'esecuzione dei programmi su piattaforme IA32.

Esempio.

Il seguente frammento di programma C mostra come la definizione di una funzione e la chiamata a funzione vengono tradotte in codice IA32:

C da tradurre	Traduzione IA32
<pre>void f(){ g(); h(); } void g(){ ... }</pre>	<pre>f: call g call h ret g: ... ret h: ... ret</pre>

4.2.3.1 Restituzione valore

Per convenzione, valori scalari come interi e puntatori⁹ vengono restituiti al chiamante dal chiamato scrivendoli nel registro `eax`.

Esempio.

Consideriamo il seguente frammento di programma C:

C da tradurre	C equivalente	Traduzione IA32
<pre>int f(){ return 7+g(); } int g(){ return 10; }</pre>	<pre>int f(){ int tmp = g(); tmp += 7; return tmp; } int g(){ tmp = 10; return tmp; }</pre>	<pre>f: call g addl \$7, %eax ret g: movl \$10, %eax ret</pre>

⁹ Non trattiamo il caso di come vengono restituiti valori in virgola mobile e strutture. Per approfondimenti si veda ad esempio la documentazione Apple su [IA-32 Function Calling Conventions](#).

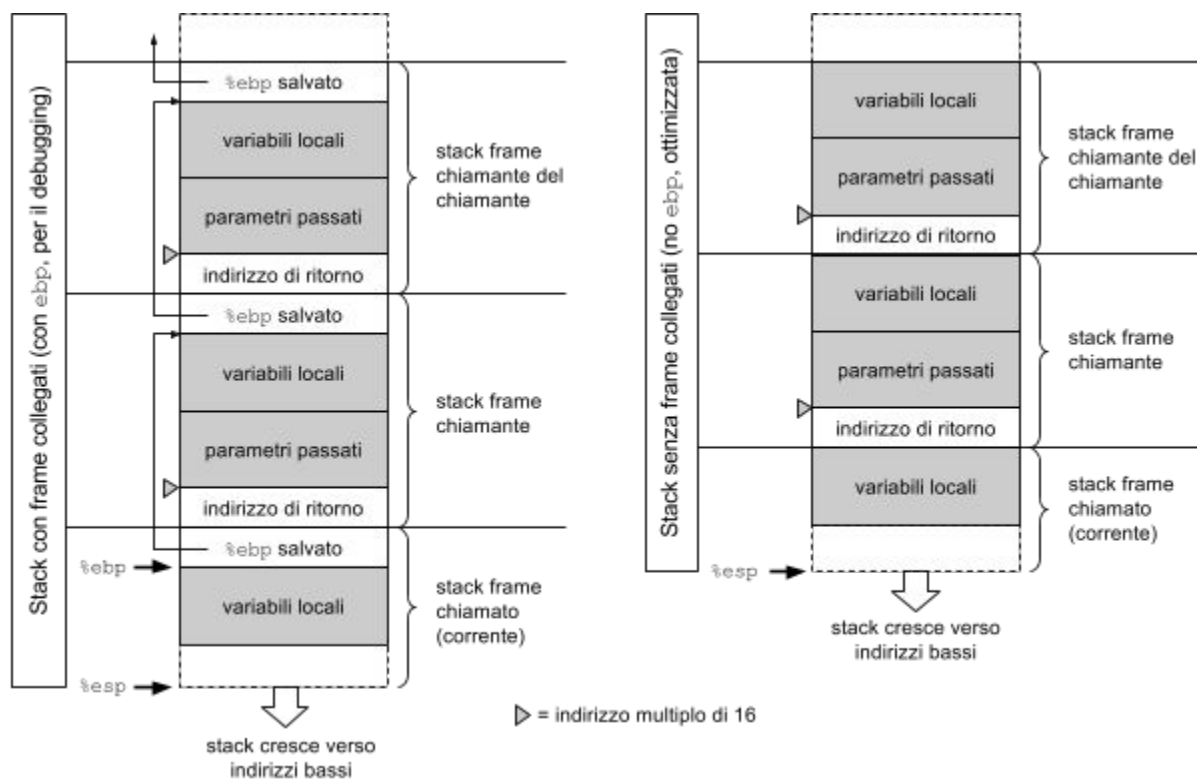
Si noti che *g* restituisce a *f* il valore 10 in *eax*, e a sua volta *f* restituisce al proprio chiamante il valore 17 in *eax*.

4.2.3.2 Stack frame e registro EBP

La stack è uno strumento essenziale per l'orchestrazione delle chiamate a funzione e per fornire spazio di memorizzazione locale alle chiamate. Ogni invocazione a funzione ha associato uno **stack frame** (o record di attivazione), che contiene spazio per memorizzare variabili locali, parametri passati ad altre funzioni, ecc.

Per consentire a un **debugger** di elencare in ogni istante le funzioni pendenti che portano dal main alla funzione correntemente eseguita, e quindi comprendere meglio il contesto in cui una funzione agisce, gli stack frame vengono organizzati a formare concettualmente una **lista collegata**, in cui il registro *ebp* punta al primo stack frame (quello della funzione correntemente eseguita). Ogni stack frame conterrà un puntatore allo stack frame del proprio chiamante.

Poiché il **collegamento fra stack frame usando *ebp* è opzionale**, illustriamo di seguito la struttura con cui viene organizzata la stack sia con che senza collegamento tra frame:

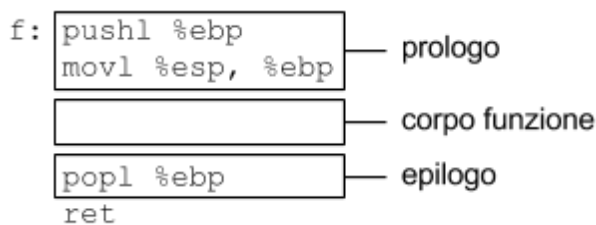


Per convenzione, **nel momento in cui si effettua un'istruzione CALL**, la **base della stack** puntata dal registro *%esp* deve essere sempre a un **indirizzo multiplo di 16**.¹⁰

¹⁰ Poiché questa convenzione ha motivazioni prestazionali e non ha implicazioni sulla correttezza di un programma, in alcuni degli esempi in questa dispensa è volutamente ignorata per rendere il codice più semplice da comprendere. Tenere a mente la convenzione è comunque utile per capire perché il codice generato da *gcc* contiene istruzioni apparentemente inutili il cui unico scopo è l'allineamento della stack a multipli di 16.

La lista di stack frame viene gestita mediante un codice di **prologo** all'inizio di una funzione e un codice di **epilogo** alla fine:

- Il **prologo** salva in stack il contenuto di `ebp` (che punta allo stack frame del chiamante) mediante l'istruzione `pushl %ebp`. Il registro base pointer `ebp` viene poi fatto puntare alla posizione corrente in stack contenuta nel registro stack pointer `esp` mediante l'istruzione `movl %esp, %ebp` (così facendo, registro `ebp` viene a puntare allo stack frame corrente).
- L'**epilogo** ripristina il valore di `ebp` che si aveva prima dell'attivazione della funzione corrente eseguendo `popl %ebp`. Il registro `ebp` verrà quindi a puntare nuovamente allo stack frame del chiamante.



In `gcc`, è possibile **omettere il collegamento fra stack frame** compilando con l'opzione `-fomit-frame-pointer`. In questo modo, non verranno generati prologo ed epilogo: la funzione sarà più veloce e compatta, ma il debugging potrebbe essere più difficoltoso.

Esempio.

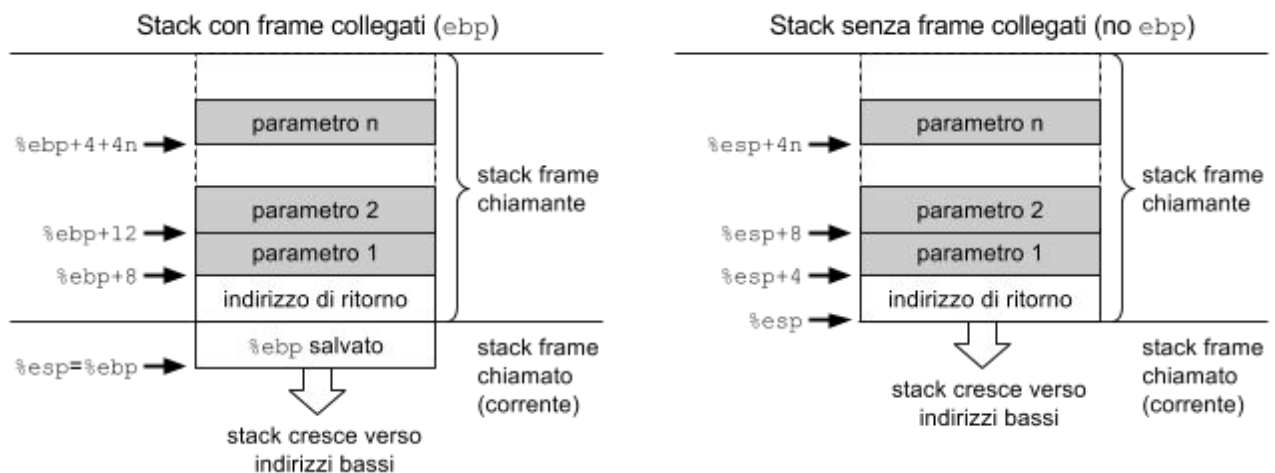
Il seguente esempio mostra come una funzione C viene compilata con e senza l'uso del registro base pointer `ebp`:

C da tradurre <code>test.c</code>	Traduzione IA32 (con <code>ebp</code>): <code>gcc -S test.c</code>	Traduzione IA32 (no <code>ebp</code>): <code>gcc -S -fomit-frame-pointer test.c</code>
<pre>int f() { return 10; }</pre>	<pre>f: pushl %ebp movl %esp, %ebp movl \$10, %eax popl %ebp ret</pre>	<pre>f: movl \$10, %eax ret</pre>

4.2.3.3 Passaggio dei parametri

I parametri di tipi primitivi¹¹ vengono passati dal chiamante al chiamato **sulla stack** e vengono disposti in memoria nello stack frame del chiamante **nello stesso ordine** in cui appaiono nell'intestazione della funzione. Parametri interi di 1 o 2 byte vengono **promossi** a 4 byte, in modo che ogni parametro passato sia di dimensione multiplo di 4 byte.

¹¹ Non trattiamo il passaggio per parametro di oggetti di tipo struttura. Per approfondimenti si veda ad esempio la documentazione Apple su [IA-32 Function Calling Conventions](https://developer.apple.com/library/ios/qa/qa1064/_index.html).



Esempio.

Il seguente esempio mostra come una funzione C con parametri viene compilata con e senza prologo/epilogo:

C da tradurre	Traduzione IA32 (ebp):	Traduzione IA32 (no ebp):
int f(int x, int y) {	f: pushl %ebp movl %esp, %ebp	f:
return x+y;	movl 8(%ebp), %eax addl 12(%ebp), %eax	movl 4(%esp), %eax addl 8(%esp), %eax
	popl %ebp	
}	ret	ret

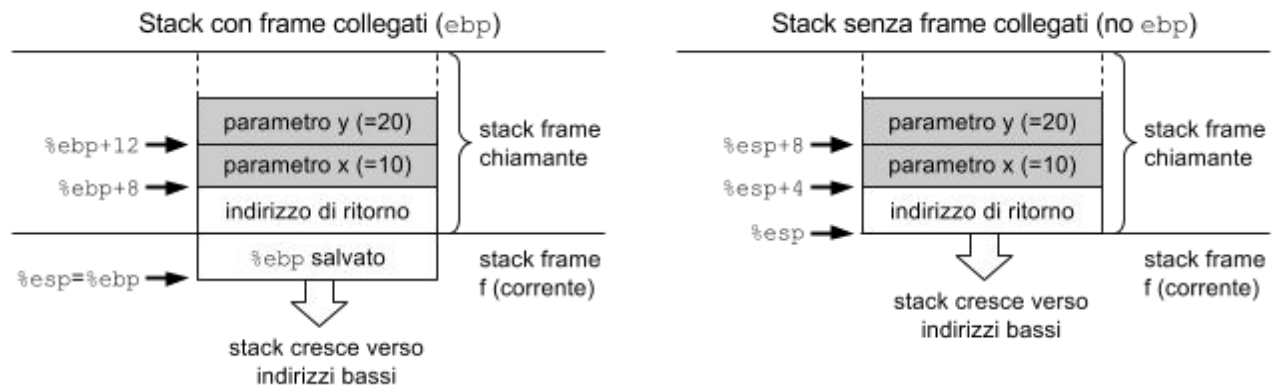
Si noti che il chiamato accede ai parametri passati dal chiamante usando `ebp` se i frame sono collegati. Se invece i frame non sono collegati, il chiamato accede ai parametri usando `esp`.

Vediamo ora come la funzione `f` può essere invocata mostrando il **passaggio dei parametri**. Assumiamo che la variabile locale `c` sia memorizzata nel registro `ecx`:

C da tradurre	Traduzione IA32 (no ebp)
c = f(10, 20);	pushl \$20 # passa il secondo parametro pushl \$10 # passa il primo parametro call f # chiama la funzione f addl \$8, %esp # toglie i due parametri dalla stack movl %eax, %ecx # assegna il risultato a c

Si noti che, poiché la stack cresce da indirizzi alti verso indirizzi bassi, le operazioni di push dei parametri su stack avvengono nell'ordine inverso in cui appaiono nella chiamata in modo che

risultino poi disposti in memoria nello stesso ordine. La seguente figura illustra lo stato della stack durante l'esecuzione del corpo della funzione `f`, con e senza frame collegati:



Osserviamo inoltre che i parametri passati sulla stack dal chiamante devono essere poi rimossi dal chiamante stesso dopo la chiamata. Nel nostro esempio, questo si ottiene incrementando lo stack pointer di 8 (`addl $8, %esp`), compensando le due push di 4 byte ciascuna effettuate prima della chiamata (`pushl $20` e `pushl $10`).

4.2.3.4 Registri caller-save e callee-save

L'esecuzione di una funzione potrebbe sovrascrivere i registri in uso al chiamante. Se il chiamante vuole avere la garanzia che il contenuto di un registro non verrà alterato a fronte dell'invocazione di una funzione, è necessario che il suo contenuto venga salvato da qualche parte, generalmente sulla stack. Si hanno due possibilità:

1. Il registro viene **salvato in stack dal chiamante (caller-save)** prima dell'invocazione e ripristinato subito dopo.
2. Il registro viene **salvato in stack dal chiamato (callee-save)** prima di eseguirne il corpo e ripristinato prima di ritornare al chiamato (il salvataggio avviene nel prologo e il ripristino nell'epilogo).

Per convenzione, alcuni registri vengono salvati dal chiamante, e altri dal chiamato:

1. **Registri caller-save:** A, C, D
2. **Registri callee-save:** B, DI, SI, SP, BP

I registri caller-save possono essere liberamente usati da una funzione senza dover essere salvati nel prologo e ripristinati nell'epilogo, ma devono essere salvati/ripristinati a fronte di una chiamata a funzione se serve mantenerne il contenuto dopo la chiamata. I registri callee-save, se usati da una funzione, devono essere salvati nel prologo e ripristinati nell'epilogo della funzione; si ha la garanzia che il loro contenuto sia preservato a fronte dell'invocazione di una funzione.

Esempio (caller-save).

Consideriamo il seguente frammento di programma C con variabili intere e assumiamo che la variabile `a` sia tenuta nel registro `eax` e la variabile `c` in `ecx`:

C da tradurre	C equivalente	Traduzione IA32 (no ebp)
<pre>int f() { return g()+h(); }</pre>	<pre>int f(){ int c = g(); int a = h(); a += c; return a; }</pre>	<pre>f: call g movl %eax, %ecx pushl %ecx call h popl %ecx addl %ecx, %eax ret</pre>

Si noti che il valore restituito da `g` viene scritto in `ecx`, che è un registro caller-save. Se non prendessimo provvedimenti, il suo valore potrebbe essere modificato dalla chiamata ad `h`, perdendo il valore restituito da `g`. Il registro `ecx` viene pertanto salvato in stack (`pushl %ecx`) prima della chiamata ad `h` e ripristinato subito dopo (`popl %ecx`).

Esempio (callee-save).

Vediamo lo stesso esempio di prima in cui usiamo un registro callee-save (`B`) invece che caller-save (`C`) per preservare il valore restituito da `g` a fronte della chiamata ad `h` (assumiamo che la variabile `a` sia tenuta nel registro `eax` e la variabile `b` in `ebx`):

C da tradurre	C equivalente	Traduzione IA32 (no ebp)
<pre>int f() { return g()+h(); }</pre>	<pre>int f(){ int b = g(); int a = h(); a += b; return a; }</pre>	<pre>f: pushl %ebx # prologo call g movl %eax, %ebx call h addl %ebx, %eax popl %ebx # epilogo ret</pre>

Si noti che il contenuto di `ebx` non viene alterato dalla chiamata ad `h` (se infatti `h` dovesse usarlo, dovrebbe salvarlo e poi ripristinarlo prima di terminare), e può quindi essere sommato al valore restituito da `h` (`addl %ebx, %eax`) per determinare il valore restituito da `f`. Il prezzo per usare `ebx` (callee-save) in `f` è che deve essere salvato nel prologo e ripristinato nell'epilogo di `f`.

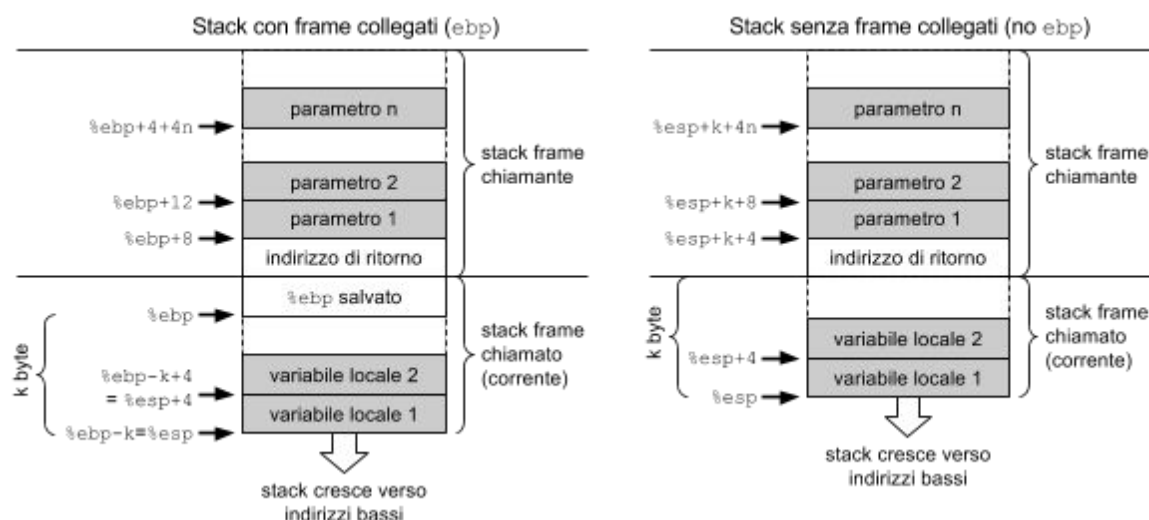
4.2.3.5 Variabili locali

Negli esempi visti finora abbiamo sempre assunto che le variabili locali venissero tenute nei registri. Questa è senz'altro la scelta più semplice e anche la migliore dal punto di vista prestazionale. Tuttavia, alle volte è necessario che le variabili locali abbiano un loro spazio riservato nello stack frame della funzione:

1. se la variabile è di tipo array o struttura e quindi non può essere memorizzata in un registro;
2. se la funzione usa più variabili locali di quanti siano i registri disponibili; oppure

- se la funzione usa l'operatore `&` su una variabile locale, che quindi deve possere un indirizzo in memoria.

Lo spazio per le variabili locali, normalmente allocato in stack nel prologo e deallocato nell'epilogo, è organizzato come segue:



Per accedere a una variabile locale, è possibile usare il registro `esp` con offset positivo. Se il registro `ebp` viene usato per puntare al frame corrente, è possibile usare equivalentemente `ebp` con offset negativo.

Esempio.

Consideriamo il seguente frammento di programma C:

C da tradurre	C equivalente	Traduzione IA32 (con ebp)
<pre>int f(int x){ int y; leggi(&y); return x+y; }</pre>	<pre>int f(int x){ int y; int* c=&y; leggi(c); int a = x; a += y; return a; }</pre>	<pre>f: pushl %ebp # prologo movl %esp, %ebp # prologo subl \$4, %esp # prologo leal -4(%ebp), %ecx # y in -4(%ebp) pushl %ecx # passa param. call leggi addl \$4, %esp # toglie param. movl 8(%ebp), %eax # x in 8(%ebp) addl -4(%ebp), %eax addl \$4, %esp # epilogo popl %ebp # epilogo ret</pre>

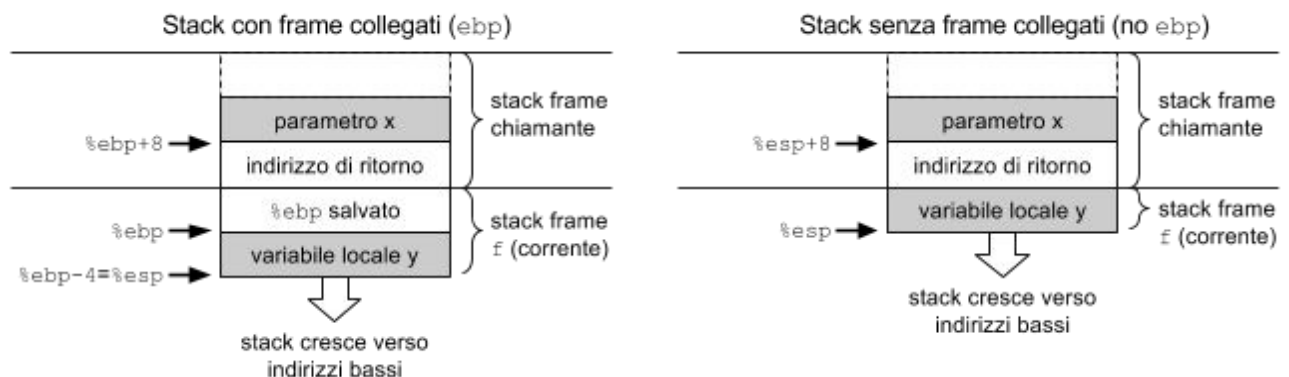
Osserviamo che la zona di memoria per le variabili locali ha dimensione $k=4$ byte. Si noti il modo in cui è compilata l'istruzione `c=&y`: viene usata una `leal` per scrivere nel registro `ecx` l'indirizzo effettivo di `y` (`%ebp-4`). Vediamo ora una versione equivalente del programma senza stack frame

collegati:

C da tradurre	C equivalente	Traduzione IA32 (senza ebp)
<pre>int f(int x){ int y; leggi(&y); return x+y; }</pre>	<pre>int f(int x){ int y; int* c=&y; leggi(c); int a = x; a += y; return a; }</pre>	<pre>f: subl \$4, %esp # prologo leal (%esp), %ecx # y in (%esp) pushl %ecx # passa param. call leggi addl \$4, %esp # toglie param. movl 8(%esp), %eax # x in 8(%esp) addl (%esp), %eax addl \$4, %esp # epilogo ret</pre>

In questo caso si accede al parametro formale *x* e alla variabile locale *y* usando lo stack pointer *esp* piuttosto che il base pointer *ebp*.

Nella figura seguente illustriamo la struttura (layout) della stack all'inizio dell'esecuzione del corpo della funzione *f* per entrambe le versioni (con e senza *ebp*):



4.2.4 Array e aritmetica dei puntatori

L'accesso alle celle di array con elementi di dimensione fino a 4 byte avviene normalmente sfruttando gli indirizzamenti a memoria della forma:

(base, indice, scala)

dove *base* è l'indirizzo del primo byte dell'array, *indice* è l'indice della cella dell'array che si vuole accedere, e *scala*=sizeof(elemento) è il numero di byte di ciascun elemento dell'array. Si noti che l'indirizzo effettivo *base+indice*scala* calcolato dall'operando (base, indice, scala) realizza l'**aritmetica dei puntatori**, scalando l'indice in base alla dimensione degli elementi dell'array.

Se l'indice *i* dell'elemento che si vuole accedere è noto a tempo di compilazione, è possibile usare la forma:

`disp(base)`

dove `base` è l'indirizzo del primo byte dell'array e `disp=i*sizeof(elemento)` è lo spiazzamento in byte rispetto alla base dell'array per arrivare all'*i*-esimo elemento dell'array.

Esempio 1.

Si consideri la scrittura della cella *c*-esima dell'array *a* di `int`, assumendo che la variabile *a* sia tenuta in `eax` e la variabile *c* in `ecx`:

C da tradurre	C equivalente	Traduzione IA32
<code>a[c]=10;</code>	<code>*(a+c)=10;</code>	<code>movl \$10, (%eax,%ecx,4)</code>

Si noti che la scala è 4 poiché l'array è di `int` e `sizeof(int)==4`.

Esempio 2.

La seguente funzione C calcola la somma degli elementi di un array di due `int` passato come parametro:

C da tradurre	C equivalente	Traduzione IA32 (no <code>ebp</code>)
<pre>int sum(int c[2]) { return c[0]+c[1]; }</pre>	<pre>int sum(int c[2]){ int a = c[0]; a += c[1]; return a; }</pre>	<pre>sum: movl 4(%esp), %ecx movl (%ecx), %eax addl 4(%ecx), %eax ret</pre>

Si noti che in questo caso gli indici 0 e 1 nell'array *c* sono noti a tempo di compilazione (costanti nel codice) ed è quindi possibile calcolare gli spiazzamenti delle rispettive celle che si vogliono accedere (0 e 4).

Esempio 3.

Generalizziamo la funzione vista sopra per sommare gli elementi di un array *v* di dimensione arbitraria *n*. Assumendo di tenere l'indirizzo *v* dell'array in `ecx`, la dimensione *n* dell'array in `edx`, la somma *s* degli elementi di *v* in `eax`, e l'indice *i* per scorrere l'array in `ebx`, possiamo scrivere:

C da tradurre	C equivalente	Traduzione IA32 (no ebp)
<pre> int sum(int* v, int n){ int i, s=0; for (i=0; i<n; i++) s += v[i]; return s; } </pre>	<pre> int sum(int* v, int n){ int s = 0; int i = 0; L: if (i>=n) goto E; s += v[i]; i++; goto L; E: return s; } </pre>	<pre> sum: pushl %ebx # prologo movl 8(%esp),%ecx # ecx=v movl 12(%esp),%edx # edx=n movl \$0, %eax # eax=s movl \$0, %ebx # ebx=i L: cmpl %edx, %ebx jge E addl (%ecx,%ebx,4),%eax incl %ebx jmp L E: popl %ebx # epilogo ret </pre>