Il linguaggio PC Assembly

Paul A. Carter

20 Marzo 2005

Traduzione italiana a cura di Giacomo Bruschi

Nota alla traduzione italiana: Nel testo i termini tecnici sono stati tradotti nel corrispondene italiano, ad eccezione di quei termini la cui traduzione avrebbe determinato una perdita di chiarezza, o di quei termini che sono comunemente usati e conosciuti in lingua inglese.

Prefazione

Scopo

Lo scopo di questo libro e' di dare al lettore una migliore comprensione di come i computer in realta' lavorano ad un piu' basso livello rispetto ai linguaggi di programmazione come il Pascal. Con l'approfondimento di come funzionano i computer, il lettore puo' essere molto piu' produttivo nello sviluppare software nei linguaggi di alto livello come il C e C++. Imparare a programmare in linguaggio assembly e' un eccellente modo per raggiungere questo scopo. Gli altri libri sul linguaggio assembly per PC ancora insegnano come programmare il processore 8086 che i primi PC usavano nel 1981! Il processore 8086 supporta solo la modalita' *remota*. In questa modalita', ogni programma puo' indirizzare ogni memoria o periferica di un computer. Questa modalita' non e' adatta per un sistema operativo multitasking sicuro. Questo libro invece discute su come programmare il processore 80386 e i successivi in modalita' *protetta* (la modalita' che usano Windows e Linux). Questa modalita' supporta le caratteristiche che i moderni sistemi operativi si aspettano come la memoria virtuale e la protezione della memoria. Ci sono diverse ragione per utilizzare la modalita' protetta:

1. E' piu' facile programmare in modalita' protetta piuttosto che in modalita' reale dell'8086, che gli altri libri usano.

2. Tutti i moderni PC lavorano in modalita' protetta.

3. Esiste software libero disponibile che gira in questa modalita'.

La mancanza di libri di testo per la programmazioni in linguaggio assembly in modalita' protetta e' la ragione principale che ha spinto l'autore a scrivere questo libro.

Come accennato prima, questo ttesto fa uso di software Free/Open Source: nella fattispecie, l'assembler NASM e il compilatore DJGPP per C/C++. Entrambi sono disponibili per il download su Internet. Il testo inoltre discute su come usare il codice assembly NASM nei sistemi operativi Linux e,

attraverso i compilatori per C/C++ di Borland e di Microsoft, sotto Windows. Gli esempi per tutte queste piattaforme possono essere trovati sul mio sito: `http://www.drpaulcarter.com/pcasm`. Tu *devi* scaricare il codice di sempio se vuoi assemblare ed eseguire i molti esempi di questo libro.

E' importante sottolineare che questo testo non tenta di coprire ogni aspetto della programmazione in assembly. L'autore ha provato a coprire gli aspetti piu' importante sui cui *tutti* i programmatori dovrebbero avere familiarita'.

Ringraziamenti

L'autore vuole ringraziare tutti i programmatori in giro per il mondo che hanno contribuito al movimento Free/Open Source. Tutti i programmi ed anche questo stesso libro sono stati prodotti utlizzando software libero. L'autore vuole ringraziare in modo speciale John S. Fine, Simon Tatham, Julian Hall e gli altri per lo sviluppo dell'assmbler NASM su cui sono basati tutti gli esempi di questo libro; DJ Delorie per lo sviluppo del compiliatore usato DJGPP per C/C++; le numerore persone che hanno contribuito al compilatore GNU gcc su cui e' basato DJGPP.Donald Knuth e gli altri per lo sviluppo dei linguaggi di composizione TEX e LATEX 2_ε che sono stati utilizzati per produrre questo libro; Richard Stallman (fondatore della Free Software Foundation), Linus Torvalds (creatore del Kernel Linux) e gli altri che hanno prodotto il sofware di base che l'autore ha usato per produrre questo lavoro.

Grazie alle seguenti persone per le correzioni:

- John S. Fine
- Marcelo Henrique Pinto de Almeida
- Sam Hopkins
- Nick D'Imperio
- Jeremiah Lawrence
- Ed Beroset
- Jerry Gembarowski
- Ziqiang Peng
- Eno Compton
- Josh I Cates

- Mik Mifflin
- Luke Wallis
- Gaku Ueda
- Brian Heward
- Chad Gorshing
- F. Gotti
- Bob Wilkinson
- Markus Koegel
- Louis Taber
- Dave Kiddell
- Eduardo Horowitz
- Sébastien Le Ray
- Nehal Mistry
- Jianyue Wang
- Jeremias Kleer
- Marc Janicki

Risorse su Internet

Pagina dell'Autore	`http://www.drpaulcarter.com/`
Pagine NASM SourceForge	`http://sourceforge.net/projects/nasm/`
DJGPP	`http://www.delorie.com/djgpp`
Linux Assembly	`http://www.linuxassembly.org/`
The Art of Assembly	`http://webster.cs.ucr.edu/`
USENET	`comp.lang.asm.x86`
Documentazione Intel	`http://developer.intel.com/design/Pentium4/documentation.htm`

Feedback

L'autore accetta volentieri ogni ritorno su questo lavoro.

E-mail: `pacman128@gmail.com`
WWW: `http://www.drpaulcarter.com/pcasm`

Capitolo 1

Introduzione

1.1 Sistemi Numerici

La memoria in un computer consiste di numeri. La memoria di un computer non immagazzina questi numeri come decimali (base 10). Per la semplificazione dell'hardware, il computer immagazzina tutte le informazioni in formato binario (base 2). Prima di tutto ripassiamo il sistema decimale.

1.1.1 Decimale

I numeri a base 10 sono composti da 10 possibili cifre (0-9). Ogni cifra del numero ha una potenza del dieci il cui esponenete e' dato dalla sua posizione nel numero. Per esempio:

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

1.1.2 Binario

I numeri a base 2 sono composti di due possibili valori (0 e 1). Ogni cifra del numero ha una potenza del 2 il cui esponente e' dato dalla sua posizione nel numero. (Una singola cifra binaria e' chiamata bit) Per esempio:

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 1 \\ &= 25 \end{aligned}$$

Questo esempio mostra come i numeri binari possono essere convertiti in numeri decimali. La Tabella 1.1 mostra come i primi numeri decimali sono rappresentati in binario.

La Figura 1.1 mostra come le singole cifre binarie (*i.e.*,i bit) sono sommate. Ecco un'esempio:

Decimale	Binario		Decimale	Binario
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

Tabella 1.1: Rappresentazione dei Decimali da 0 a 15 in Binario

nessun precedente riporto				precedente riporto			
0	0	1	1	0	0	1	1
+0	+1	+0	+1	+0	+1	+0	+1
0	1	1	0	1	0	0	1
			c		c	c	c

Figura 1.1: Somma Binaria (c sta per *riporto*)

$$\begin{array}{r} 11011_2 \\ +10001_2 \\ \hline 101100_2 \end{array}$$

Considerando la divisione decimale sia ha:

$$1234 \div 10 = 123\ r\ 4$$

si puo' vedere che la divisione taglia la cifra decimale piu' a destra del numero e sposta le altre cifre decimali di una posizione a destra. Dividendo per due si ottiene la stessa operazione, ma in questo caso per le cifre binarie del numero. Consideriamo la seguente divisione binaria[1]:

$$1101_2 \div 10_2 = 110_2\ r\ 1$$

Cio' puo' essere usato per convertire un numero decimale nel sua rappresentazione binaria equivalente, come dimostra la Figura 1.2. Questo metodo trova innazitutto la cifra piu' a destra, chiamata *bit meno significativo* (lsb). La cifra piu' a sinistra e' invece chiamata *bit piu' significativo* (msb). L'unita' base della memoria e' costituita da 8 bit ed e' chiamata *byte*.

[1] Il 2 accanto ai numeri (pedice) indica che quel numero e' in rappresentazione binaria, non decimale.

Decimale	Binario
$25 \div 2 = 12\ r\ 1$	$11001 \div 10 = 1100\ r\ 1$
$12 \div 2 = 6\ r\ 0$	$1100 \div 10 = 110\ r\ 0$
$6 \div 2 = 3\ r\ 0$	$110 \div 10 = 11\ r\ 0$
$3 \div 2 = 1\ r\ 1$	$11 \div 10 = 1\ r\ 1$
$1 \div 2 = 0\ r\ 1$	$1 \div 10 = 0\ r\ 1$

Cosi' $25_{10} = 11001_2$

Figura 1.2: Conversione decimale

1.1.3 Esadecimale

I numeri esadecimali usano la base 16. Gli esadecimali (o *hex* per semplicita') possono essere usati come una scorciatoia per i numeri binari. Gli esadecimali hanno 16 possibili cifre. Cio' crea dei problemi dal momento che non ci sono simboli addizionali per le cifre oltre il 9. Per convenzione si e' deciso di usare le lettere per queste cifre. Cosi' le 16 cifre esadecimali sono i numeri da 0 a 9 e le lettere A,B,C,D,E e F. La cifra A e' equivalente a 10 in decimale, B a 11, e cosi' via. Ogni cifra in un numero esadecimale ha una potenza del 16 il cui esponente e' dato dalla posizione della cifra nel numero stesso. Esempio:

$$\begin{aligned} 2\mathrm{BD}_{16} &= 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0 \\ &= 512 + 176 + 13 \\ &= 701 \end{aligned}$$

Per convertire un decimale in un esadecimale, si usa la stessa procedura usata per i numeri binari, ma si divide per 16. La Figura 1.3 ne mostra un'esempio.

La ragione per cui gli esadecimali sono utili e' che esiste un modo molto semplice di conversione tra gli esadecimali e i binari. I numeri binari diventano grandi ed ingombranti molto velocemente. Gli esadecimali forniscono cosi' un modo piu' compatto per rappresentare i numeri binari. Per convertire un numero esadecimale in un numero binario, basta convertire ogni cifra esadecimale in un numero binario a 4 bit. Per esempio, $24\mathrm{D}_{16}$ viene convertito in $0010\ 0100\ 1101_2$. E' importante notare che gli 0 in testa di

$$
\begin{aligned}
589 \div 16 &= 36\ r\ 13 \\
36 \div 16 &= 2\ r\ 4 \\
2 \div 16 &= 0\ r\ 2
\end{aligned}
$$

Cosi' $589 = 24\mathrm{D}_{16}$

Figura 1.3:

ogni gruppo di 4 bit sono importanti! Se, ad esempio, viene eliminato lo zero di testa del gruppo centrale, nella conversione precedente, il risultato e' sbagliato. La conversione da binario a esadecimale e' altrettanto semplice. Occorre applicare la procedura inversa: spezzare il numero binario in gruppi di 4 bit e convertire ciascun gruppo nella sua rappresentazione esadecimale. Occorre spezzare il numero partendo da destra, non da sinistra. Cio' assicura che il processo di conversione utilizzi i gruppi di 4 bit corretti.[2] Esempio:

110	0000	0101	1010	0111	1110_2
6	0	5	A	7	E_{16}

Un numero a 4 bit e' chiamato *nibble* . Ogni cifra esadecimale corrisponde cosi' ad un nibble. Due nibble fanno un byte. Un byte puo' essere rappresentato da un numero esadecimale a 2 cifre. Il valore di un byte va da 0 a 11111111 in binario, da 0 a FF in esadecimale e da 0 a 255 in decimale.

1.2 Organizzazione del Computer

1.2.1 Memoria

La memoria e' misurata in unita' di kilobyte ($2^{10} = 1,024$ byte), megabyte ($2^{20} = 1,048,576$ bytes) e gigabyte ($2^{30} = 1,073,741,824$ byte).

L'unita' di base della memoria e' il byte. Un computer con 32 megabyte di memoria puo' immagazzinare approssimativamente 32 milioni di byte. Ogni byte in memoria e' contrassegnato da un numero unico che e' il suo indirizzo come mostra la Figura 1.4.

Indirizzo	0	1	2	3	4	5	6	7
Memoria	2A	45	B8	20	8F	CD	12	2E

Figura 1.4: Indirizzi di memoria

[2]Se non e' chiaro perche' il punto di partenza e' importante, provate a convertire il numero partendo da sinistra.

word	2 byte
double word	4 byte
quad word	8 byte
paragraph	16 byte

Tabella 1.2: Unita' di memoria

Spesso la memoria e' usata in pezzi piu' grandi di un singolo byte. Nell'architettura del PC, sono stati dati dei nomi a queste sezioni piu' grandi, come mostra la Tabella 1.2.

Tutti i dati in memoria sono numerici. I Caratteri sono memorizzati utilizzando un *codice carattere* che associa un numero a ciascun carattere. Uno dei sistemi di codifica piu' comune e' conosciuto come *ASCII* (American Standard Code for Information Interchange). Un nuovo, piu' completo sistema di codifica che sta sostituendo l'ASCII e' UNICODE. Una differenza chiave nei due sistemi e' che ASCII usa un byte per codificare un carattere mentre UNICODE usa 2 byte (o una *word*) per carattere.

Per esempio, ASCII mappa il byte 41_{16} (65_{10}) al carattere *A* maiuscolo,mentre UNICODE usa la word 0041_{16}. Dal momento che ASCII usa un byte, e' limitato a solo 256 diversi caratteri[3]. Unicode estende il valore ASCII a una word (2 byte) che permettono di rappresentare molti piu' valori. Cio' diventa importante per la rappresentazione dei caratteri di tutte le lingue del mondo.

1.2.2 La CPU

L' unita' centrale di calcolo (o CPU - Central Processing Unit) e' l'unita' fisica che elabora le istruzioni. Le istruzioni che sono elaborate dalla CPU sono in genere molto semplici. Queste istruzioni possono aver bisogno che i dati sui cui lavorano, si trovino memorizzate in particolari locazioni di memoria all'interno della CPU stessa, *i registri*. La CPU puo' accedere ai dati nei registri in modo molto piu' veloce rispetto ai dati in memoria. Putroppo, il numero dei registri in una CPU e' limitato, cosi' il programmatore deve stare attento a memorizzare nei registri solo le informazioni che sta attualmente utilizzando.

Le istruzioni che un tipo di CPU esegue costituiscono il *linguaggio macchina* di quella CPU. I linguaggi macchina hanno molte piu' strutture di base rispetto ai linguaggi di alto livello. Le istruzioni del linguaggio macchina sono codificate come semplici numeri, non in formato testo piu' leggibile. La CPU deve essere capace di interpretare una istruzione velocemente per

[3]Infatti ASCII usa i 7 bit meno significativi, cosi' ha solo 128 valori possibili.

eseguirla in maniera efficiente. I linguaggi macchina sono stati definiti e concepiti per raggiungere questo obbiettivo e non per essere decifrati facilmente dalle persone. I programmi scritti in altri linguaggi devono essere convertiti nel linguaggio macchina nativo della CPU che si trova sul computer. Un *compilatore* e' un programma che traduce i programmi scritti in linguaggi di programmazione di alto livello nel linguaggio macchina per una particolare architettura hardware. In genere ogni CPU ha il suo particolare linguaggio macchina. Questa e' una ragione per cui i programmi scritti per Mac non possono girare su PC tipo IBM.

GHz *sta per gigahertz o 1 miliardo di cicli al secondo. Una CPU da 1.5 GHz ha 1.5 miliardi di cicli.*

I Computer usano un'*orologio* per sincronizzare l'esecuzione delle istruzioni. L'orologio pulsa ad una frequenza prefissata (conosciuta come *velocita' del processore*). Quando acquisti un computer a 1.5 GHz, 1.5 GHz indica la frequenza dell'orologio. L'orologio non tiene conto dei minuti o dei secondi. Semplicemente pulsa ad una frequenza costante. L'elettronica della CPU usa le pulsazioni per elaborare le istruzioni correttamente, cosi' come il battito di un metronomo aiuta chi fa musica a seguire il ritmo giusto. Il numero di battiti (o come sono definiti comunemente, *il numero di clici*) di cui un'istruzione necessita dipendono dalla generazione della CPU e dal suo modello. Il numero dei cicli dipende dalle istruzioni prima di questa e da altri fattori.

1.2.3 La famiglia delle CPU 8086

I PC tipo IBM contengono una CPU della famiglia Intel 80x86 (o un loro clone): Le CPU di questa famiglia hanno tutte le stesse caratteristiche incluso il linguaggio macchina di base. Comunque i processori piu' recenti hanno migliorato molto queste caratteristiche.

8088,8086: Queste CPU, dal punto della programmazione sono identiche. Queste sono le CPU che erano usate nei primi PC. Forniscono diversi registri a 16 bit: AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS. Riescono a supportare fino ad un Megabyte di memoria ed operano in *modalita' reale*. In questa modalita' un programma puo' accedere ad ogni indirizzo di memoria, anche indirizzi utilizzati da altri programmi! Cio' rende il debug e la sicurezza molto difficili da realizzare! Inoltre la memoria del programma e' stata divisa in *segmenti*. Ogni segmento non puo' essere piu' grande di 64K.

80286: Queste CPU era usate nei PC di classe AT. Aggiunge alcune nuove istruzioni al linguaggio base delle 8088/86. La piu' importante novita' e' l'introduzione della *modalita' protetta a 16 bit*. In questa modalita', la CPU puo' accedere fino a 16 megabyte di memoria e protegge i programmi da accessi di altri programmi. Ancora pero' i programmi sono divisi in segmenti che non possono essere piu' grandi di 64K.

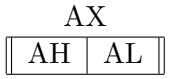


Figura 1.5: Il registro AX

80386: Questa CPU migliora fortemente l'80286. Prima di tutto estende molti dei registri portandoli a 32 bit (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) e aggiunge due nuovi registri a 16 bit, FS e GS. Inoltre aggiunge una nuova *modalita' protetta a 32 bit*. In questa modalita' possono essere allocati fino a 4 Gigabyte di memoria. I programmi sono ancora divisi in segmenti, ma ora questi segmenti possono arrivare fino ad una dimensione di 4 gigabyte!

80486/Pentium/Pentium Pro: Queste CPU della famiglia 80x86 aggiungono pochissime novita'. Principalmente velocizzano l'esecuzione delle istruzioni.

Pentium MMX: Questo processore aggiunge delle istruzioni specifiche per la multimedialita' (MMX sta per MultiMedia eXtensions). Queste istruzioni migliorano le performance per le operazioni di grafica.

Pentium II: Questo e' un processore Pentium Pro con le istruzioni MMX. (Il Pentium III e' essenzialmente un po' piu' veloce del Pentium II.)

1.2.4 Registri a 16 bit dell' 8086

Le prime CPU 8086 fornivano 4 registri generali a 16 bit: AX, BX, CX e DX. Ognuno di questi registri puo' essere scomposto in 2 regitri da 8 bit. Per esempio, il registro AX puo' essere scomposto nei due registri AH e AL come mostra la Figura 1.5. Il registro AH contiene gli 8 bit superiori (o alti) di AX, mentre AL contiene gli 8 inferiori (o bassi). Spesso AH e AL sono usati come registri indipendenti ad un byte. E' comunque importante capire che questi registri non sono indipendenti da AX. Cambiando il valore di AX, si cambia anche il valore di AH e AL, e *viceversa*. Questi registri generali sono usati frequentemente per spostamento di dati e istruzioni aritmentiche.

Ci sono poi 2 registri indice a 16 bit: SI e DI. Spesso sono usati come puntatori ma possono anche essere usati come i registri generici, per le stesse operazioni. A differenze di questi ultimi pero', non possono essere decomposti in registri da 8 bit. I registri BP e SP, entrambi a 16 bit, sono usati come puntatori allo stack del linguaggio macchina e sono chiamati Base Pointer e Stack Pointer, rispettivamente. Di questi discuteremo piu' avanti.

I registri a 16 bit CS, DS, SS e ES sono definiti registri di *segmento*. Il loro scopo principale e' infatti quello di indicare quale memoria e' usata

dalle varie parti del programma. CS sta per Code Segment (segmento del Codice), DS sta per Data Segment (segmento dei dati), SS sta per Stack Segment (segmento dello stack) ed ES sta per Extra Segment. ES e' usato principalmente come un registro di segmento temporaneo. Maggiori dettagli su questi segmenti verrano dati nelle Sezioni 1.2.6 e 1.2.7.

Il registro puntatore dell'Istruzione (IP - Instruction Pointer) e' usato insieme al registro CS per tenere traccia dell'indirizzo del'istruzione successiva che deve essere eseguita dalla CPU. Normalmente, quando l'istruzione e' terminata, il registro IP e' modificato per puntare alla successiva istruzione in memoria.

Il registro FLAGS memorizza importanti informazioni sui risultati di una istruzione precedente. Questi risultati sono memorizzati come singoli bit nel registro. Per esempio, il bit Z e' 1 se il risultato di una precedente operazione era zero, oppure 0 se il risultato non era zero. Non tutte le istruzioni modificano i bit in FLAGS. Puoi consultare la tabella in appendice per vedere come le singole istruzioni modificano il registro FLAGS.

1.2.5 I Registri a 32 bit dell' 80386

Il processore 80386 e i successivi hanno esteso i registri. Per esempio il registro AX a 16 bit, e' stato esteso a 32 bit. Per essere compatibile con il passato, AX ancora indica il registro a 16 bit, mentre EAX si riferisce ora al registro a 32 bit. AX rappresenta i 16 bit bassi di EAX, cosi' come AL rappresenta gli 8 bit bassi di AX(e EAX). Non c'e' modo per accedere direttamente ai 16 bit alti di EAX. Gli altri registri estesi sono EBX, ECX, EDX, ESI ed EDI.

Molti degli altri registri sono estensi nello stesso modo. BP diventa EBP; SP diventa ESP; FLAGS diventa EFLAGS e IP diventa EIP. C'e' da notare che, diversamente dai registri indice e generali, in modalita' protetta a 32 bit (discussa piu' avanti), questi registri vengono usati solo nella loro versione estesa.

I registri segmento sono ancora a 16 bit nell' 80386. Ci sono inoltre due nuovi registri: FS e GS. Il loro nome non indica nulla, ma come ES sono dei registri segmento temporanei.

Una delle definizioni del termine *word* si riferisce alla dimensione dei registri dei dati della CPU. Per la famiglia degli 80x86, questo termine genera un po' di confusione. Nella Tabella 1.2, si puo' vedere che word e' definita come 2 byte (o 16 bit). Questa definizione era nata con i primi 8086. Con lo sviluppo dell'80386 fu deciso di lasciare definizione di *word* invariata, anche se la dimensione dei registri non era piu' la stessa.

1.2.6 Modalita' Reale

Da dove arriva, quindi, il tristemente famoso limite di 640K del DOS? Il BIOS richiede una parte di 1 Mb per il suo codice e per le periferiche hardware, come lo schermo video.

In modalita' reale la memoria e' limitata ad un solo megabyte (2^{20} byte). Indirizzi di memoria validi vanno da (in Hex) 00000 a FFFFF. Questi indirizzi richiedono un numero a 20 bit. Ovviamente, un numero di questa dimensione non puo' stare in nessuno dei registri a 16 bit dell'8086. Intel ha risolto il problema utilizzano due valori da 16 bit per rappresentare l'indirizzo. . Il primo valore e' chiamato *selettore*. I valori del selettore devono essere memorizzati nei registri di segmento. Il secondo valore da 16 bit e' chiamato *offset*. . L'indirizzo fisico referenziato dalla coppia (32 bit) *selettore:offset* e' ricavato dalla formula:

$$16 * \text{selettore} + \text{offset}$$

Moltiplicare per 16 in esadecimale e' semplice, basta aggiungere 0 a destra del numero. Per esempio, l'indirizzo fisico referenziato da 047C:0048 e' calcolato cosi':

$$\begin{array}{r} 047C0 \\ +0048 \\ \hline 04808 \end{array}$$

In effetti, il valore del selettore e' un paragrafo. (vedi Tabella 1.2).

Questa segmentazione degli indirizzi ha degli svantaggi:

- Il valore di un singolo selettore puo' referenziare solo 64 K di memoria. Cosa succede se un programma ha piu' di 64K di codice? Il singolo valore di CS non puo' essere usato per l'intera esecuzione del programma. Il programma deve essere diviso in sezioni (chiamate *segmenti*) piu' piccoli di 64K. Quando l'esecuzione si sposta da un segmento ad un'altro, il valore di CS deve cambiare. Problemi simili si hanno, con grandi quantita' di dati, con il registro DS. Cio' puo' essere veramente scomodo!

- Ogni byte in memoria non ha un'unico indirizzo di segmento. L'indirizzo fisico 04808 puo' essere referenziato da 047C:0048, 047D:0038, 047E:0028 o 047B:0058. Cio' puo' complicare la comparazione di indirizzi di segmento.

1.2.7 Modalita' protetta a 16-bit

Nella modalita' protetta a 16 bit degli 80286, i valori del selettore sonon interpretati in maniera completamente differente rispetto alla modalita' reale. In modalita' reale, il valore del selettore e' un paragrafo di memoria fisica. In modalita' protetta , il valore di un selettore e' un' *indice* in una

tabella dei descrittori. In entrambe le modalita', i programmi sono divisi in segmenti. In modalita' reale questi segmenti sono in posizione fissa nella memoria fisica e il valore del selettore indica un numero all'inizio del segmento. In modalita' protetta , i segmenti non sono in posizione fissa nella memoria fisica. Infatti i settori non sono per nulla in memoria!

La modalita' protetta usa una particolare tecnica chiamata *memoria virtuale* . L'idea che sta alla base di questa tecnica e' quella per cui il codice e i dati vengono tenuti in memoria solo quando sono in uso. I dati ed il codice che non e' in uso e' tenuto temporaneamente sul disco, finche' non diventano necessari di nuovo. Nella modalita' protetta a 16 bit , i segmenti sono spostati tra memoria e disco solo all'occorenza. Quando un segmento viene caricato in memoria dal disco, e' molto probabile che venga memorizzato in un'area diversa rispetto quella in cui era caricato prima di essere trasferito sul disco. Tutto cio' e' fatto in maniera trasparente dal sistema operativo. Il programma non deve essere scritto diversamente per girare in modalita' protetta.

In modalita' protetta, ad ogni segmento e' assegnata una entry nella tavola dei descrittori. Questa entry contiene tutte le informazioni che il sistema ha bisogno di sapere sul segmento. Queste includono: se il segment e' attualmente in memoria; se in memoria, dove; permessi di accesso (*es.*, sola lettura). L'indice di questa entry del segmento e' il valore del selettore che viene memorizzato nei registri di segmento.

Un ben noto giornalista informatico ha definiro la CPU 286 "la morte del cervello"

Un grande svantaggio della modalita' protetta a 16 bit e' che gli offset sono ancora a 16 bit. Di conseguenza, la dimensione dei segmenti e' ancora limitata a 64K. Cio' rende problematico l'uso di array di grandi dimensioni.

1.2.8 Modalita' protetta a 32 bit

L'80386 introduce la modalita' protetta a 32 bit. Ci sono due principali novita' tra la modalita' protetta a 32 bit del 386 rispetto a quella a 16 bit del 286:

1. Gli Offset sono estesi a 32 bit. Cio' permette ad un offset di spaziare da 0 a 4 miliardi. Di conseguenza, i segmenti arrivano ad avere una dimensione massima di 4 gigabyte.

2. I Segmenti possono essere divisi in pezzi piu' piccoli, di 4 K, chiamati *Pagine*. La memoria virtuale opera adesso con queste pagine, anziche' con i segmenti. Cio' significa che solo alcune parti del segmento si possono trovare in memoria in un dato momento. In modalita' protetta a 16 bit , o tutto il segmento era in memoria oppure nessuna parte lo era. Questo approcio non e' comodo con i segmenti piu' grandi che permette la modalita' protetta a 32 bit.

In Windows 3.x, *la modalita' standard* si riferisce alla modalita' protetta a 16 bit, mentre la *modalita' avanzata* indica quella a 32 bit. Windows 9X,Windows NT/2000/XP,OS/2 e Linux usano tutte la modalita' protetta a 32 bit.

1.2.9 Interrupts

A volta il flusso ordinario di un programma deve essere interrotto per elaborare eventi che richiedono una pronta risposta. L'hardware di un computer fornisce un meccanismo chiamato *Interrupts* per gestire questi eventi. Per esempio, quando il mouse e' mosso, l'hardware del mouse interrompe il programma in esecuzione per gestire il movimento del mouse (per spostare il puntatore del mouse,*ecc.*). L'interrupts fa si' che il controllo sia passato al *gestore di interrupts*. I Gestori di interrupts sono routine che elaborano l'Interrupts. Ad ogni tipo di Interrupts e' assegnato un numero intero. All'inizio della memoria fisica risiede una tabella dei *vettori di Interrupts* che contiene gli indirizzi di segmento dei gestori di interrupts. Il numero di Interrupts e' essenzialmente un'indice dentro questa tabella.

Interrupts esterni sono generati al di fuori della CPU. (Il mouse e' un'esempio di questo tipo) . Molte periferiche di I/O generano interrupts (*Es.*, la tastiera, il timer, i dischi driver, CD-ROM e schede audio). Gli Interrupts interni sono generati dentro la CPU o per un'errore oppure per una istruzione di interrupts. Gli Interrupts di errore sono chimati *traps*. Gli interrupts generati da istruzioni sono chiamati *Interrupts software*. DOS usa questi tipi di interrupts per implementare le sue API (Application Programming Interface). I sistemi operativi piu' moderni (come Windows o Linux) usano un'interfaccia basata su C.[4]

Molti gestori di Interrupts ritornano il controllo al programma sospeso dopo aver finito le operazioni. Inoltre aggiornano tutti i registri con i valori che questi avevano prima della chiamata dell'interrupts. Cosi' il programma riprende l'esecuzione come se nulla fossa successo (ad eccezione del fatto che perde alcuni cicli della CPU). I Traps generalmente non ritornano il controllo. Spesso terminano l'esecuzione del programma.

1.3 Il Linguaggio Assembly

1.3.1 Linguaggio Macchina

Ogni tipo di CPU capisce solo il suo linguaggio macchina . Le istruzioni in linguaggio macchina sono numeri memorizzati come byte in memoria.

[4]Comunque, potrebbero usare una interfaccia di piu' basso livello, il livello kernel.

Ogni istruzione ha il suo codice numerico univoco chiamato *codice dell'operazione* o, abbreviato, *opcode* . Le istruzioni dei processori della famiglia 80x86 possono variare in dimensione. L'Opcode e' sempre all'inizio dell'istruzione. Molte istruzioni possono includere dati (*Es.*, costanti o indirizzi) usati dall'istruzione stessa.

Il Linguaggio macchina e' molto difficile da programmare direttamente. Decifrare il significato delle istruzione in codifica numerica e' molto tedioso per le persone. Per esempio l'istruzione che dice di sommare i registri EAX ed EBX e mettere il risultato nel registro EAX e' codificata con il seguente codice Hex:

```
03 C3
```

Questo e' terribilmente ovvio. Fortunatamente, un programma, *l'assembler* fa questo lavoro noioso al posto del programmatore.

1.3.2 Il Linguaggio Assembly

Un programma in linguaggio assembly e' memorizzato come testo (proprio come succede per i programmi in linguaggio di alto livello). Ogni istruzione assembly rappresenta esattamente una istruzione in linguaggio macchina. Per esempio, l'istruzione di addizione descritta sopra sarebbe rappresentata in linguaggio assembly come:

```
add eax, ebx
```

Qui il significato dell'istruzione e' *molto* piu' chiaro rispetto al linguaggio macchina. La parola `add` e' lo *mnemonico* per l'istruzione di addizione. La forma generale di una istruzione in linguaggio assembly e':

```
mnemonico operando(i)
```

Ci sono voluti diversi anni prima che i ricercatori informatici riuscissero a capire come scrivere un compilatore!

Un *assembler* e' un programma che legge un file di testo con le istruzioni in assembly e converte l'assembly in codice macchina. I *Compilatori* sono programmi che operano una conversione simile per i linguaggi di alto livello. Un assembler e' molto piu' semplice di un compilatore. Ogni istruzione del linguaggio assembly rappresenta direttamente una singola istruzione in linguaggio macchina. Le istruzioni dei linguaggi di alto livello sono *molto* piu' complesse e possono richiedere molte istruzioni in linguaggio macchina.

Un'altra differenza importante fra l'assembly e i linguaggi di alto livello e' che, dal momento che ogni CPU ha il suo linguaggio macchina, essa ha anche il suo proprio linguaggio assembly. La portabilita' di programmi in assembly su piattaforme diverse e' *molto piu' difficile* rispetto ai linguaggi di alto livello.

Gli esempi di questo libro utilizzano il Netwide Assembler o NASM. Esso e' disponibile liberamente in Internet (vedi la prefazione per l'URL).

Assembler molto comuni sono il Assembler Microsoft (MASM) o il Assembler Borland (TASM). Ci sono alcune differenze nella sintassi dell'assembly tra MASM/TASM e NASM.

1.3.3 Operandi delle istruzioni

Le istruzioni in codice macchina hanno tipi e numeri variabili di operandi. In generale comunque ogni istruzione in codice macchina avra' un numero fisso di operandi (da 0 a 3). Gli operandi possono avere i seguenti tipi:

registro: Questi operandi si riferiscono direttamente al contenuto dei registri della CPU.

memoria: Questi invece si riferiscono a dati nella memoria. L'indirizzo dei dati puo' essere una costante codificata nell'istruzione o puo' essere calcolata utilizzando i valori dei registri. Gli indirizzi sono sempre offset dall'inizio di un segmento.

immediati: Questi sono valori fissi contenuti nell'istruzione stessa. Sono memorizzati nell'istruzione (nel segmento di codice) non nel segmento dei dati.

impliciti: Questi operandi non sono esplicitamente mostrati. Per esempio l'istruzione di incremento aggiunge uno al registro o alla memoria. Il valore 1 e' implicito.

1.3.4 Istruzioni di base

L'istruzione base principale e' l'istruzione `MOV` . Questa istruzione sposta i dati da una locazione ad un'altra. (come l'operatore di assegnamento nei linguaggi di alto livello). Questa istruzione ha due operandi:

```
mov destinazione, sorgente
```

I dati specificati in *sorgente* sono copiati in *destinazione*. Una restrizione e' che entrambi gli operandi non possono essere operandi di memoria. Cio' porta alla luce un'altra stranezza dell'assembly. Ci sono spesso delle regole in qualche modo arbitrarie di come vengono usate le varie istruzioni. Gli operandi devono inoltre avere la stessa dimensione: Il valore di AX non puo' essere memorizzato in BL.

Ecco un 'esempio (il punto e virgola ";" inizia un commento):

```
mov    eax, 3   ; sposta 3 nel registro EAX (3 e' un'operando immediato)
mov    bx, ax   ; sposta il valore di AX nel registro BX
```

L'istruzione `ADD` e' utilizzata per addizionare interi.

```
add    eax, 4   ; eax = eax + 4
add    al, ah   ; al = al + ah
```

L'istruzione `SUB` sottrae interi.

```
sub    bx, 10   ; bx = bx - 10
sub    ebx, edi ; ebx = ebx - edi
```

Le istruzioni `INC` e `DEC` incrementano o decrementano i valori di uno. Dal momento che uno e' un'operando implicito, il codice macchina per `INC` e `DEC` e' piu' piccolo rispetto al codice delle equivalenti istruzioni `ADD` e `SUB`.

```
inc    ecx      ; ecx++
dec    dl       ; dl--
```

1.3.5 Direttive

Una *direttiva* e' un artefatto dell'assembler, non della CPU. Le direttive generalmente sono usate per istruire l'assembler o di fare qualcosa o di informarlo di qualcosa. Non sono tradotte in codice macchina. Usi comuni delle direttive sono:

- definire costanti
- definire memoria per memorizzare dati
- raggruppare memoria in segmenti
- includere codice in maniera condizionata
- includere altri file

Come succede per il C, il codice NASM, prima di essere assemblato passa un preprocessore. Questo ha infatti molti degli stessi comandi del preprocessore del C. Comunque le direttive del preprocessore NASM iniziano con % anziche' con # del C.

La direttiva equ

La direttiva `equ` e' utilizzata per definire *simboli*. I simboli sono costanti etichettate che possono essere usati nel programma assembly. Il formato e':

simbolo `equ` *valore*

I valori dei simboli *non* possono essere ridefiniti.

Unita'	Lettera
byte	B
word	W
double word	D
quad word	Q
10 byte	T

Tabella 1.3: Lettere per le direttive `RESX` e `DX`

La direttiva %define

Questa direttiva e' simile alla direttiva del C `#define`. E' comunemente usata per definire macro costanti come in C.

```
%define SIZE 100
      mov    eax, SIZE
```

Il codice sopra definisce una macro chiamata `SIZE` e mostra il suo utilizzo in una istruzione `MOV`. Le macro sono piu' flessibili dei simboli per due motivi: possono essere ridefinite e possono essere piu' di una semplice costante numerica

Direttive dati

Le direttive dei dati sono usate nei segmenti di dati per definire spazi di memoria. Ci sono due modi con cui la memoria puo' essere riservata. Il primo modo definisce solo lo spazio per la memoria; il secondo modo definisce lo spazio e il suo valore iniziale. Il primo metodo utilizza una delle direttive `RES`*X*. La *X* e' sostituita con una lettera che determina la dimensione dell'oggetto (o degli oggetti) che vi saranno memorizzati. La tabella 1.3 mostra i possibili valori.

Il secondo metodo (che definisce anche il valore iniziale) usa una delle direttive `D`*X*. La lettera *X* ha lo stesso uso della direttiva `RES`*X* e puo' assumere gli stessi valori.

E' molto comune indicare locazioni di memoria con *etichette*. Sotto sono riportati alcuni esempi:

```
L1    db     0         ; byte etichettato L1 con valore iniziale 0
L2    dw     1000      ; word etichettata L2 con valore iniziale 1000
L3    db     110101b   ; byte inizializzato al valore binario 110101 (53 in decimale)
L4    db     12h       ; byte inizializzato al valore hex 12 (18 in decimale)
L5    db     17o       ; byte inizializzato al valore oct 17 (15 in decimale)
L6    dd     1A92h     ; double word inizializzato al valore hex 1A92
L7    resb   1         ; 1 byte non inizializzato
L8    db     "A"       ; byte inizializzato con il codice ASCII per A (65)
```

Le doppie virgolette e le virgolette singole sono trattate nello stesso modo. Definizioni consecutive di dati sono allocate in memoria sequenzialmente. Cosi', la word L2 e' memorizzata subito dopo L1. Le sequenze di memoria possono anche essere definite.

```
L9    db     0, 1, 2, 3              ; definisce 4 byte
L10   db     "w", "o", "r", 'd', 0   ; definisce una stringa di tipo C = ``word''
L11   db     'word', 0               ; stesso risultato della riga 10 (L10)
```

La direttiva `DD` puo' essere usata sia con interi che con costanti in virgola mobile a singola precisione[5]. La direttiva `DQ` puo' invece essere usata solo per definire costanti in virgola mobile a doppia precisione.

Per sequenze piu' grandi, la direttiva del Nasm, `TIMES` puo' essere spesso molto utile. Questa direttiva ripete il suo operando un numero di volte specificato. Per esempio,

```
L12   times 100 db 0                 ; equivale a 100 (db 0)
L13   resw   100                     ; riserva 100 locazioni di dimensione word
```

Ricorda che le etichette possono essere usate per riferirsi a dati nel codice. Ci sono due modi in cui le etichette possono essere usate. Se e' utilizzata una etichetta piana, questa e' interpretata come indirizzo (o offset) dei dati. Se il nome dell'etichetta e' inserito tra parentesi quadre(`[]`), allora questa e' interpretata come valore dei dati a quell'indirizzo. In altra parole, si puo' pensare ad una label come ad un *puntatore* ad un dato e le parentesi quadre dereferenziano il puntatore come fa l'asterisco in C. (MASM/TASM seguono diverse convensioni). In modalita' a 32 bit gli indirizzi sono a 32 bit. Ecco alcuni esempi:

```
1       mov    al, [L1]      ; copia il byte in L1 dentro AL
2       mov    eax, L1       ; EAX = indirizzo del byte in L1
3       mov    [L1], ah      ; copia AH in L1
4       mov    eax, [L6]     ; copia la double word in L6 dentro EAX
5       add    eax, [L6]     ; EAX = EAX + double word in L6
6       add    [L6], eax     ; double word in L6 += EAX
7       mov    al, [L6]      ; copia il primo byte della double word in L6 dentro AL
```

La riga 7 dell'esempio mostra una importante proprieta' del NASM. L'assembler *non* tiene traccia del tipo di dato a cui si riferisce l'etichetta. Spetta al programmatore utilizzare correttamente l'etichetta. Piu' tardi sara' uso comune di memorizzare gli indirizzi nei registri ed usare questi ultimi come puntatori in C. Di nuovo, non c'e' nessun controllo che il puntarore sia usato correttamente. Da cio' deriva che l'assembly e' molto piu' incline all'errore rispetto al C.

Consideriamo le seguenti istruzioni:

[5] virgola mobile a singola precisione e' equivalente ad una variabile di tipo `float` in C.

```
mov     [L6], 1             ; memorizza 1 in L6
```

Questo comando produce un'errore di `operation size not specified`. Perche'? Perche' l'assembler non sa se memorizzare 1 come byte, come word oppure come double word. Per correggere l'errore si aggiunge uno specificatore di dimensione:

```
mov     dword [L6], 1       ; memorizza 1 in L6
```

Questo dice all'assembler di memorizzare 1 alla double word che inizia all'indirizzo di L6. Altri specificatori sono: `BYTE`, `WORD`, `QWORD` e `TWORD`[6]

1.3.6 Input e Output

Le attivita' di input ed output sono molto dipendenti dal sistema. Cio' implica l'interfacciamento con l'hardware di sistema. Linguaggi di alto livello, come il C, forniscono librerie standard che forniscono una interfaccia di programmazione semplice ed uniforme per l'I/O. I linguaggi assembly non forniscono librerie standard. Questi devono accedere direttamente all'hardware (che una operazione privilegiata in modalita' protetta) oppure usare qualche routine di basso livello fornita dal sistema operativo.

E' molto comune per le routine in assembly essere interfacciate con il C. Un vantaggio di questo approccio e' che il codice assembly puo' utilizzare le routine della libreria I/O standard del C. Comnque, il programmatore deve conoscere le regole per passare informazioni tra le routine che usa il C. Queste regole sono troppo complicate per essere affrontate qui. (Saranno riprese piu' tardi). Per semplificare l'I/O, l'autore ha sviluppato le sue proprie routine che nascondono le complesse regole del C e forniscono una interfaccia piu' semplice. La Tabella 1.4 descrive le queste routine.Tutte le routine preservano il valore di tutti i registri ad eccezione delle routine di lettura. Queste routine modificano il valore del registro EAX. Per utilizzare queste routine e' necessario includere un file con le informazioni di cui l'assembler necessita per poterle usare. Per includere un file in NASM, si usa la direttiva `%include`. Le righe seguenti includono il file necessario per le routine dell'autore[7]:

```
%include "asm_io.inc"
```

[6] `TWORD` definisce un'area di memoria di 10 byte. Il coprocessore in virgola mobile utilizza questo tipo di dati.

[7] Il file `asm_io.inc` (e il file oggetto `asm_io` necessario per `asm_io.inc`) sono nella sezione di download del codice esempio alla pagina web di questa guida, `http://www.drpaulcarter.com/pcasm`

print_int	stampa a video il valore dell'intero memorizzaro in EAX
print_char	stampa a video il carattere ASCII corrispondente al valore memorizzato in AL
print_string	stampa a video il contenuto della stringa all'*indirizzo* memorizzato in EAX. La stringa deve essere di tipo C. (*i.e.* terminata da null).
print_nl	stampa a video un carattere di nuova riga.
read_int	legge un intero dalla tastiera e lo memorizza nel registro EAX.
read_char	legge un carattere dalla tastiera e memorizza il suo codice ASCII nel registro EAX.

Tabella 1.4: Routine di I/O Assembly

Per utilizzare le routine di stampa, occorre caricare in EAX il valore corretto e utilizzare l'istruzione `CALL` per invocare la routine. L'istruzione `CALL` e' equivalente ad una chiamata di funzione in un linguaggio di alto livello. `CALL` sposta l'esecuzione ad un'altra sezione di codice, e ritorna all'origine della chiamata quando la routine e' completata. Il programma di esempio riportato di seguito mostra alcuni esempi di chiamate a queste routine.

1.3.7 Debugging

La libreria dell'autore contiene anche alcune utili routine per il debug dei programmi. Queste routine visualizzano informazioni circa lo stato del computer senza modificarlo. Queste routine sono in realta' delle *macro* che preservano lo stato attuale della CPU e poi effetuano una chiamata ad una subroutine. Le macro sono definite nel file `asm_io.inc` discusso prima. Le macro sono usate come istruzioni ordinarie. Gli operandi delle macro sono separati da virgole.

Ci sono 4 routine per il debug: `dump_regs`, `dump_mem`, `dump_stack` e `dump_math`; Queste rispettivamente visualizzano lo stato dei registri, della memoria, dello stack e del coprocessore matematico.

dump_regs Questa macro stampa il valore dei registri (in esadecimale) del computer nella `stdout` (*es.* lo schermo). Visualizza inoltre i bit settati nel registro FLAGS[8]. Per esempio, se il flag zero e' 1, *ZF* e' visualizzato. Se e' 0, *ZF* non e' visualizzato. Accetta come argomento un'intero che viene stampato. Puo' essere usato per distinguere l'output di diversi comandi `dump_regs`.

[8]Il Capitolo 2 approfondisce questo registro

dump_mem Questa macro stampa i valori di una regione di memoria (in esadecimale e in codifica ASCII). Accetta tre argomenti separati da virgola. Il primo argomento e' un'intero che e' usato per etichettare l'output (come per il comando `dump_regs`). Il secondo e' l'indirizzo da visualizzare (Puo' essere una etichetta). L'ultimo argomento e' il numero di paragrafi di 16 bit da visualizare dopo l'indirizzo. La memoria visualizzata iniziera' dal limite del primo paragrafo prima del'indirizzo richiesto.

dump_stack Questa macro stampa il valore dello stack della CPU . (Lo stack sara' affrontato nel Capitolo 4.). Lo stack e' organizzato come una pila di double word e questa routine lo visualizza in questo modo. Accetta tre argomenti separati da virgole. Il primo argomento e' un intero (come per `dump_regs`). Il secondo argomento e' il numero di double word da visualizzare *prima* dell'indirizzo memorizzato nel registro `EBP` e il terzo argomento e' il numero di double word da visualizzare *dopo* l'indirizzo in `EBP`.

dump_math Questa macro stampa il valore dei registri del coprocessore matematico. Accetta come argomento un numero intero che, come per il comando `dump_ regs` viene usato per etichettare la stampa.

1.4 Creare un Programma

Oggi, e' molto raro creare un programma stand alone scritto completamente in linguaggio assembly. Assembly e' spesso usato per creare alcune routine critiche. Perche'? E' *piu' semplice* programmare in un linguaggi di alto livello rispetto all'assembly. Inoltre , utilizzare l'assembly rende un programma molto difficile da portare su altre piattaforme. Infatti e' raro usare completamente l'assembly.

Vi chiederete quindi perche' qualcuno dovrebbe imparare l'assembly?

1. Qualche volta il codice scritto in assembly e' piu' piccolo e veloce del codice generato da un compilatore.

2. L'assembly permette di accedere direttamente alle caratteristiche hardware del sistema e questo, con un linguaggio di alto livello potrebbe essere molto difficile se non impossibile.

3. Imparare a programmare in assembly aiuta a capire in maniera piu' approfondita come i computer lavorano.

4. Imparare a programmare in assembly aiuta a capire meglio come i compilatori e i linguaggi di alto livelllo come il C funzionano.

```
int main()
{
  int ret_status ;
   ret_status = asm_main();
  return ret_status ;
}
```

Figura 1.6: Codice di `driver.c`

Questi due ultimi punti dimostrano che imparare l'assembly puo' essere utile anche se non si programmera' mai in assembly in futuro. Infatti, l'autore raramente programma in assembly, ma usa le idee che ha imparato da esso ogni giorno.

1.4.1 Il Primo Programma

I prossimi programmi nel testo partiranno da un piccolo programma guida in C mostrato in Figura 1.6. Questo programma semplicemente chiama un'altra funzione, `asm_main`. Questa e' la routine scritta in assembly. Ci sono molti vantaggi nell'utilizzare una routine guida scritta in C. Primo, permette all'archiettura del C di impostare correttamente il programma per girare in modalita' protetta. Tutti i segmenti e i loro corrispondenti registri segmento saranno inizializati dal C. Il codice assembly non si deve occupare e preoccupare di questa operazione. Secondo, la libreria C sara' disponibile anche da codice assembly. Le routine I/O dell'autore traggono vantaggio da cio'. Infatti utilizzano le funzioni di I/O del C (*es.* `printf`). Il codice sotto mostra un semplice programma in assembly:

first.asm

```
1  ; file: first.asm
2  ; Primo programma in assembly. Questo programma richiede due interi come
3  ; ingresso e stampa come uscita la loro somma.
4  ;
5  ; per creare l'eseguibile con djgpp:
6  ; nasm -f coff first.asm
7  ; gcc -o first first.o driver.c asm_io.o
8
9  %include "asm_io.inc"
10 ;
11 ; i dati inizializzati sono messi nel segmento .data
12 ;
13 segment .data
```

```
14 ;
15 ; queste etichette si riferiscono alle stringhe usate per l'output
16 ;
17 prompt1 db    "Enter a number: ", 0     ; non dimenticare il terminatore null
18 prompt2 db    "Enter another number: ", 0
19 outmsg1 db    "You entered ", 0
20 outmsg2 db    " and ", 0
21 outmsg3 db    ", the sum of these is ", 0
22
23 ;
24 ; i dati non inizializzati sono posti nel segmento .bss
25 ;
26 segment .bss
27 ;
28 ; Queste etichette si riferiscono a double word utilizzate per memeorizzare gli ingressi
29 ;
30 input1  resd 1
31 input2  resd 1
32
33 ;
34 ; Il codice e' posto nel segmento .text
35 ;
36 segment .text
37         global  _asm_main
38 _asm_main:
39         enter   0,0               ; routine di setup
40         pusha
41
42         mov     eax, prompt1      ; stampa il prompt
43         call    print_string
44
45         call    read_int          ; legge l'intero
46         mov     [input1], eax     ; lo memorizza in input1
47
48         mov     eax, prompt2      ; stampa prompt
49         call    print_string
50
51         call    read_int          ; legge l'intero
52         mov     [input2], eax     ; lo memorizza in input2
53
54         mov     eax, [input1]     ; eax = dword in input1
55         add     eax, [input2]     ; eax += dword in input2
```

```
56          mov     ebx, eax          ; ebx = eax
57
58          dump_regs 1                ; stampa i valori del registro
59          dump_mem  2, outmsg1, 1    ; stampa la memroria
60  ;
61  ; il codice seguente stampa i risultati in sequenza
62  ;
63          mov     eax, outmsg1
64          call    print_string      ; stampa il primo messaggio
65          mov     eax, [input1]
66          call    print_int         ; stampa input1
67          mov     eax, outmsg2
68          call    print_string      ; stampa il secondo messaggio
69          mov     eax, [input2]
70          call    print_int         ; stampa input2
71          mov     eax, outmsg3
72          call    print_string      ; stampa il terzo messaggio
73          mov     eax, ebx
74          call    print_int         ; stampa la somma (ebx)
75          call    print_nl          ; stampa una nuova riga
76
77          popa
78          mov     eax, 0            ; ritorna al codice C
79          leave
80          ret
```

first.asm

La riga 13 del programma definisce una sezione del programma che specifica la memoria utilizzata nel segmento dati (il cui nome e' `.data`). Solo memoria inizializzata puo' essere definita in questo segmento. Dalla riga 17 alla riga 21 sono dichiarate alcune stringhe. Queste saranno stampate utilizzando la libreria C e quindi dovranno terminare con il carattere *null* (codice ASCII 0). Ricorda che c'e' una grande differenza tra `0` e `'0'`.

I dati non inizializzati dovranno essere dichiarati nel segmento bss (chiamato `.bss` alla riga 26). Questo segmento prende il suo nome da un'originario operatore dell'assembler UNIX che sta per "block started by symbol." C'e' anche un segmento dello stack, che verra' discusso piu' avanti.

Il segmento codice e' storicamente chiamato `.text`. E' qui che sono messe le istruzioni. Nota che il nome della routine principale (riga 38) ha come prefisso il carattere underscore. Questo fa parte della *convenzione di chiamata del C*. Questa convenzione specifica le regole che il C usa quando compila il codice. E' molto importante conoscere questa convenzione quando si interfacciano il C e l'assembly. Piu' avanti sar presentata l'intera conven-

zione. Comnuque, per ora, basta sapere che tutti i simboli C (*es.* funzioni e variabili globali) hanno un underscore prefisso aposto dal compilatore C (Questa regola e' specificatamente per DOS/Windows, il compilatore C di Linux non aggiunge nessun prefisso).

La direttiva `global` alla riga 37 dice all'assemblatore di rendere globale l'etichetta `_asm_main`. Diversamente dal C, le etichette hanno un *campo di visibilit interna* per default. Cio' significa che solo il codice nello stesso modulo puo' usare quelle etichette. La direttiva `global` da' all'etichetta (o alle etichette) specificata un *campo di visibilit esterno*. Questo tipo di etichette possono essere accedute da ogni modulo del programma. Il modulo `asm_io` dichiara l'etichette `print_ int` *ecc.* , come globali. Per cio' possono essere richiamate nel modulo `first.asm`.

1.4.2 Dipendenze del Compilatore

Il codice assembly sotto e' specifico per il compilatore C/C++ DJGPP[9] basato su GNU[10]. Questo compilatore puo' essere liberamente scaricato da internet. Richiede un computer con un processore 386 o superiore e gira sotto DOS/Windows 95/98 o NT. Questo compilatore usa file oggetto nel formato COFF (Common Object File Format). Per assemblare il codice in questo formato si usa l'opzione`-f coff` con il `NASM` (come mostrato nei commenti del codice precedente). L'estensione dell'oggetto risultante sara' `o`.

Anche il Compilatore C per Linux e' un compilatore GNU. Per convertire il codice precedente perche' giri anche sotto Linux, occorre rimuovere i prefissi underscore alle righe 37 e 38. Linux usa il formato ELF (Executable and Linkable Format) per i file oggetto. Si usa l'opzione `-f elf` per Linux. Anche con questa opzione viene generato un file di estensione `o`.

file di esempio disponibili nella sito web dell'autore sono gia' stati modificati per funzionare con il compilatore appropriato.

Il Borland C/C++ e' un'altro compilatore molto popolare. Utilizza il formato Microsoft OMF per i file oggetto. In questo caso si usa l'opzione `-f obj`. L'estensione di un file oggetto sara' `obj`. Il formato OMF usa diverse direttive di `segmento` rispetto agli altri formati. Il segmento dati (riga 13) deve essere cambiato in:

```
segment _DATA public align=4 class=DATA use32
```

Il segmento bss (riga 26) deve essere cambiato in:

```
segment _BSS public align=4 class=BSS use32
```

Il segmento Text (riga 36) deve essere cambiato in:

```
segment _TEXT public align=1 class=CODE use32
```

[9] `http://www.delorie.com/djgpp`

[10] GNU e' un progetto della Free Software Foundation (`http://www.fsf.org`)

Inoltre dovrebbe essere aggiunta una nuova riga prima della riga 36:

```
group DGROUP _BSS _DATA
```

Il compilatore C/C++ Microsoft usa sia il formato OMF che il formato Win32 per i file oggetto. (se e' passato un file di tipo OMF, questo e' convertito internamente dal compilatore). Il formato Win32 permette ai segmenti di essere definiti come per DJGPP e Linux. Si usa l'opzione `-f win32` per questo formato. L'estensione del file sar `obj`.

1.4.3 Assemblare il Codice

Il primo passo e' assemblare il codice. Dalla riga di comando digitare:

```
nasm -f object-format first.asm
```

dove *object-format* e' *coff* o *elf* o *obj* oppure *win32* dipendentemente dal Compilatore C usato. (Ricorda che il codice sorgente deve essere modificato sia per Linux che Borland)

1.4.4 Compilare il Codice C

Il `driver.c` si compila utilizzando un Compilatore C. Per DJGPP, si usa:

```
gcc -c driver.c
```

L'opzione `-c` indica di fare solo la compilazione e non ancora il collegamento. La stessa opzione e' valida pure per Linux,Borland e Microsoft.

1.4.5 Collegare i File Oggetto

Il collegamento e' un processo di combinazione del codice macchina e dai dati dei file oggetto e dei file di libreria per creare un file eseguibile. Come sara' dimostrato sotto, questo processo e' complicato.

Il codice C necessita della libreria C standard e di uno speciale *codice di avvio* per funzionare. E' *molto* piu' semplice lasciare al compilatore la chiamata al Linker con i parametri appropriati piuttosto che provare a chiamare direttamente il Linker . Per esempio, per collegare il codice per il primo programma, con DJGPP si usa:

```
gcc -o first driver.o first.o asm_io.o
```

Questo comando crea un file eseguibile chiamato `first.exe` (o solo `first` sotto Linux)

Con Borland si usa:

```
bcc32 first.obj driver.obj asm_io.obj
```

Borland usa lo stesso nome del primo file elencato come nome dell'eseguibile. In questo caso il programma e' chiamato first.exe. E' possibile combinare i passi di compilazione e di collegamento. Per esempio,

```
gcc -o first driver.c first.o asm_io.o
```

In questo modo gcc compilera' `driver.c` e poi fara' il collegamento.

1.4.6 Capire un listing file in assembly

L'opzione `-l` *`listing-file`* puo' essere usata per dire al `Nasm` di creare un file di listing di un dato nome. Questo file mostra come il codice e' stato assemblato. Ecco come le righe 17 e 18 appaiono nel file di listing. (I numeri di riga sono nel file di listing; comunque nota che I numeri di riga nel file sorgente potrebbero non essere corrispondenti ai numeri di riga del file di listing.)

```
48 00000000 456E7465722061206E-     prompt1 db    "Enter a number: ", 0
49 00000009 756D6265723A2000
50 00000011 456E74657220616E6F-     prompt2 db    "Enter another number: ", 0
51 0000001A 74686572206E756D62-
52 00000023 65723A2000
```

La prima colonna di ogni riga e' il numero di riga, mentre la seconda e' l'offset (in Hex) dei dati nel segmento. La terza colonna mostra i valori esadecimali che saranno memorizzati. In questo caso i valori esadecimali corrispondono ai codice ASCII. Infine, il testo del codice sorgente corrispondente. Gli offset della seconda colonna in realt *non* sono i veri offset a cui saranno memorizzati i dati nel programma definitivo. Ogni modulo puo' definire le proprie etichette nel segmento dati (ed anche in altri segmenti). Nella fase di collegamento (vedi sezione 1.4.5), tutte queste definizioni di etichette nei segmenti dati saranno combinate per formare un unico segmento dati. Il nuovo definitivo offset sara' a quel punto calcolato dal linker.

Ecco una piccola sezione (righe dalla 54 alla 56 del file sorgente) del segmento text dal file di listing:

```
94 0000002C A1[00000000]          mov     eax, [input1]
95 00000031 0305[04000000]        add     eax, [input2]
96 00000037 89C3                  mov     ebx, eax
```

La terza colonna mostra il codice macchina generato dall'assembly. Spesso il codice completo dell'istruzione non puo' essere ancora generato. Per esempio, alla riga 94 l'offset (o indirizzo) di `input1` non e' conosciuto finche' il

codice non e' collegato. L'assembler puo' calcolare l'opcode per l'istruzione `mov` (che dal listing e' A1) , ma scrive l'offset tra parentesi quadre perche' l'esatto valore non puo' ancora essere calcolato. In questo caso e' usato un offset temporaneo di 0 in quanto `input1` e' all'inizio del segmento bss definito nel file. Ricorda che cio' *non* significa che `input1` sara' all'inizio del segmento bss definitivo del programma. Quando il codice e' collegato il linker inserira' il valore corretto dell'offset nella posizione. Altre istruzioni, come la riga 96, non referenziano nessuna etichetta. Qui l'assembler puo' calcolare il codice macchina completo.

Rappresentazione Big e Little Endian

Guardando attentamente la riga 95, qualcosa appare molto strano circa l'offset tra parentesi quadre nel codice macchina. L'etichetta `input2` si trova all'offset 4 (come definito in questo file); invece, l'offset che appare in memoria non e' 00000004, ma 04000000. Perche'? I diversi processori allocano in memoria interi su piu' byte in ordine diverso. I due modi piu' diffusi per memorizzare interi sono: *big endian* e *little endian*. Big endian e' il metodo che appare piu' naturale. Il byte piu' grande (il piu' significativo) e' memorizzato per primo, il secondo piu' grande per secondo, e cosi' via. Per esempio la dword 00000004 sar memorizzata come 4 byte 00 00 00 04. I mainframe IBM, molti processori RISC e I processori motorola usano il metodo big endian. I processori tipo intel invece usano il metodo little endian! Qui il byte meno significativo viene memorizzato per primo. Cosi' 00000004 e' memorizzato come 04 00 00 00. Questo formato e' integrato nella CPU e non puo' essere modificato. Normalmente i programmatori non hanno la necessita' di preoccuparsi di quale formato e' usato. Ci sono invece delle circostanze in cui cio' e' importante.

Endian e' pronunciato come indian.

1. Quando dati binari sono trasferiti fra computer differenti (attraverso un file oppure una rete).

2. Quando il dato binario e' scritto in memoria come un intero multibyte ed e' invece letto come singolo byte, o *viceversa*.

Queste modalit non si applicano all'ordine degli elementi degli array. Il primo elemento di un array e' sempre quello con l'indirizzo piu' basso. Stesso discorso vale per le stringhe, che sono array di caratteri. Queste modalit invece si applicano ai singoli elementi dell'array.

1.5 File Modello

La Figura 1.7 mostra uno scheletro di file che puo' essere usato come punto di partenza per scrivere programmi in assembly.

skel.asm

```
1  %include "asm_io.inc"
2  segment .data
3  ;
4  ; I dati inizializzati sono messi nel segmento dati qui
5  ;
6
7  segment .bss
8  ;
9  ; I dati non inizializzati sono messi nel segmento bss
10 ;
11
12 segment .text
13         global  _asm_main
14 _asm_main:
15         enter   0,0               ; routine di setup
16         pusha
17
18 ;
19 ; Il codice e' messo nel segmento text. Non modificare il codice prima
20 ; o dopo questo commento
21 ;
22
23         popa
24         mov     eax, 0            ; ritorna il controllo al C
25         leave
26         ret
```

skel.asm

Figura 1.7: File modello

Capitolo 2

Linguaggio Assembly di Base

2.1 Lavorare con gli Interi

2.1.1 Rappresentazione degli Interi

Gli interi possono essere di due tipi: unsigned e signed. Gli interi unsigned (che non sono negativi) sono rappresentati direttamente in binario. Il numero 200, come byte intero unsigned sarebbe rappresentato come 11001000 (o C8 in esadecimale).

Gli interi Signed (che possono essere positivi o negativi) sono rappresentati in maniera piu' complessa. Per esempio, consideriamo -56. $+56$ come byte sarebbe rappresentato da 00111000. Sulla carta, si potrebbe rappresentare -56 come -111000, ma come potrebbe essere rappresentato questo numero, come byte, nella memoria del computer? Come potrebbe essere memorizzato il meno?

Ci sono tre tecniche generali che sono state usate per rappresentare gli interi signed nella memoria del computer. Tutti e tre i metodi usano il bit piu' significativo dell'intero come *bit di segno*. Questo bit e' 0 se il numero e' positivo, 1 se negativo.

Signed Magnitude

Il primo metodo e' il piu' semplice ed e' chiamato *signed magnitude*. Esso rappresenta l'intero in due parti. La prima parte e' il bit del segno, mentre la seconda rappresenta la magnitudine dell'intero: 56 sarebbe rappresentato come byte come $\underline{0}$0111000 (il bit del segno e' sottolineato) e -56 come $\underline{1}$0111000. Il piu' grande valore rappresentabile in un byte sarebbe $\underline{0}$1111111 o $+127$ mentre il piu' piccolo sarebbe $\underline{1}$1111111 o -127. Per negare un valore, il bit di segno e' invertito. Questo metodo e' diretto, ma ha i suoi svantaggi. Innanzitutto, ci sono due possibili valori di zero, $+0$ ($\underline{0}$0000000) e -0 ($\underline{1}$0000000). Dal momento che zero non e' ne' positivo ne' negativo,

entrambe queste rappresentazioni dovrebbero comportarsi nello stesso modo. Questo complica la logica dell'aritmetica della CPU. Secondariamente, anche l'aritmetica in generale viene complicata. Se 10 viene sommato a -56, questa operazione deve essere convertita come la sottrazione di 10 da 56. Di nuovo, la logica della CPU ne viene complicata.

Complemento a Uno

Il secondo metodo e' conosciuto come rappresentazione in *(*complemento a 1). Il complemento a 1 di un numero e' trovato invertendo ogni bit del numero (oppure , in altro modo, con la sottrazione $1 -$ numeroinbit.) Per esempio il complemento a 1 di $\underline{0}0111000$ ($+56$) e' $\underline{1}1000111$. Nel notazione del complemento a 1, il complemento a 1 equivale alla negazione. Cosi', $\underline{1}1000111$ e' la rappresentazione per -56. Nota che il bit del segno e' automaticamente cambiato dal complemento a 1 e che come ci si puo' aspettare, l'utilizzo del complemento a 1 due volte porta al numero originale. Come per il primo metodo, ci sono due rappresentazioni dello zero: $\underline{0}0000000$ ($+0$) e $\underline{1}1111111$ (-0). L'aritmetica, con i numeri in completmento a 1 e' complicata.

C'e' un semplice trucco per trovare il complemento a 1 di un numero in esadecimale senza convertirlo in binario. Il trucco e' di sottrarre ogni cifra esadecimale da F (o 15 in decimale). Questo metodo assume che il numero di bit nel numero sia un multiplo di 2. Ecco un'esempio: $+56$ e' rappresentato da 38 in esadecimale. Per trovare il complemento a 1, si sottrae ogni cifra da F per ottenere C7 (in Hex). Questo concorda con il risultato precedente.

Complemento a 2

I primi due metodi descritti erano usati nei primi computer. I moderni computer usano un terzo metodo chiamato rappresentazione in *complemento a 2*. Il complemento a 2 di un numero e' trovato con il seguente algoritmo:

1. Si trova il complemento a uno del numero
2. Si aggiunge 1 al risultato dello step 1

Ecco un'esempio utilizzando $\underline{0}0111000$ (56). Prima si trova il complemnto a 1: $\underline{1}1000111$. Poi si aggiunge 1:

$$\begin{array}{rr} & \underline{1}1000111 \\ + & \qquad\quad 1 \\ \hline & \underline{1}1001000 \end{array}$$

Nella notazione in complemento a 2, trovare il complemento a 2 equivale alla negazione del numero. Cosi', il numero $\underline{1}1001000$ e' la rappresentazione

Numero	Rappresentazione Hex
0	00
1	01
127	7F
-128	80
-127	81
-2	FE
-1	FF

Tabella 2.1: Rappresentazione in complemento a due

in complemento a 2 di -56. Due negazioni dovrebbero riprodurre il numero originale. Sorprendentemente il complemento a 2 soddisfa questo requisito. Proviamo ad trovare il complemento a 2 di $\underline{1}1001000$ aggiungendo 1 al complemento a 1.

$$\begin{array}{rr} & \underline{0}0110111 \\ + & 1 \\ \hline & \underline{0}0111000 \end{array}$$

Quando si esegue l'addizione nella operazioni di complemento a 2, l'addizione del bit piu' a sinistra puo' produrre un riporto. Questo riporto *non* e' usato. Ricorda che tutti i dati di un computer sono di dimensione fissa (in termini di numero di bit). La somma di 2 byte produce sempre un byte come risultato (cosi' come la somma di 2 word produce una word, *ecc.*) Questa proprieta' e' importante per la notazione in complemento a due. Per esempio, consideriamo zero come un byte in complemento a due ($\underline{0}0000000$) Il calcolo del suo complemento a due produce la somma:

$$\begin{array}{rr} & \underline{1}1111111 \\ + & 1 \\ \hline c & \underline{0}0000000 \end{array}$$

dove c rappresenta il riporto. (Piu' tardi sara' mostrato come individuare questo riporto, che non e' memorizzato nel risultato.) Cosi', nella notazione in complemento a due, c'e' solo uno zero. Cio' rende l'aritmetica del complemento a due molto piu' semplice rispetto a quella dei precedenti metodi.

Nella notazione in complemento a due, un byte con segno puo' essere utilizzato per rappresentare i numeri da -128 a $+127$. La Tabella 2.1 mostra alcuni valori selezionati. Se sono usati 16 bit, possono essere rappresentati i numeri da $-32,768$ a $+32,767$. $-32,768$ e' rappresentato da 7FFF, mentre $+32,767$ da 8000, -128 come FF80 e -1 come FFFF. I numeri a 32 bit

in complemento a due vanno approssimativamente da -2 miliardi a $+2$ miliardi.

La CPU non sa cosa un particolare byte (o una word o una double word) rappresenta. Assembly non ha idea dei tipi, a differenza dei linguaggi di alto livello. Come viene intepretato un dato, dipende dalla istruzione che e' usata su quel dato. Considerare il valore esadecimale FF come -1 (signed) o come $+255$(unsigned) dipende dal programmatore. Il linguaggio C definisce i tipi interi signed e unsigned. Questo permette al compilatore del C di determinare correttamente quale istruzioni usare con i dati.

2.1.2 Estensione del Segno

In assembly, tutti i dati hanno una dimensione specificata. E' spesso necessario aver bisogno di cambiare la dimensione dei dati, per utilizzarli con altri dati. Decrementare la dimensione e' piu' semplice.

Decrementare la dimensione di un dato

Per decrementare la dimensione di un dato vengono rimossi i bit piu' significativi. Ecco un'esempio:

```
mov    ax, 0034h      ; ax = 52 (stored in 16 bits)
mov    cl, al         ; cl = lower 8-bits of ax
```

Naturalmente, se il numero non puo' essere rappresentato correttamente in una dimensione minore, il decremento di dimensione non funziona. Per esempio, se `AX` fosse 0134h (o 308 in decimale), nel codice sopra `CL` sarebbe ancora impostato a 34h. Questo metodo funziona sia con i numeri con segno che con i numeri senza segno. Considerando i numeri con segno, se `AX` fosse FFFFh (-1 come word), allora `CL` sarebbe FFh (-1 as a byte). E' da notare che questo non sarebbe stato corretto se il valore di `AX` fosse stato senza segno!

La regola per i numeri senza segno e' che tutti i bit che vengono rimossi devono essere 0 per far si' che la conversione sia corretta. La regola per i numeri con segno e' che tutti i bit che vengono rimossi devono essere o tutti 1 o tutti 0. Inoltre, il primo bit non rimosso deve avere lo stesso valore dei bit rimossi. Questo sara' il nuovo bit di segno del valore piu' piccolo. E' importante che sia lo stesso del bit di segno originale.

Incrementare la dimensione dei dati

Incrementare la dimensione dei dati e' piu' complicato che decrementare. Consideriamo il byte FF. Se viene esteso ad una word, quale valore dovrebbe avere la word? Dipende da come FF e' interpretato. Se FF e' un byte senza

segno (255 in decimale), allora la word dovrebbe essere 00FFh; Invece, se fosse un byte con segno (−1 in decimale), la word allora sarebbe FFFFh.

In generale, per estendere un numero senza segno, occorre impostare a 0 tutti i nuovi bit del numero espanso. Cosi', FF diventa 00FF. Invece, per estendere un numero con segno, occorre *estendere* il bit del segno. Cio' significa che i nuovi bit diventano una copia del bit di segno. Dal momento che il bit di segno di FF e' 1, i nuovi bit dovranno essere tutti 1, per produrre FFFF. Se il numero con segno 5A(90 in decimale) fosse esteso, il risultato sarebbe 005A.

L'80386 fornisce diverse istruzioni per l'estensione dei numeri. Ricorda i computer non conoscono se un numero e' con segno o senza segno. Sta' al programmatore usare l'istruzione corretta.

Per i numeri senza segno, occorre semplicemente mettere a 0 i bit piu' alti utilizzando l'istruzione `MOV`. Per esempio per estendere il byte in AL ad una word senza segno in AX:

```
mov    ah, 0   ; Imposta a 0 gli 8 bit superiori
```

Invece non e' possibile usare l'istruzione `MOV` per convertire la word senza segno in AX nella double word senza segno in EAX. Perche' no? Non c'e' modo di impostare i 16 bit superiori in EAX con una `MOV`. L'80386 risolve il problema fornendo una nuova istruzione `MOVZX`. Questa istruzione ha due operandi. La destinazione (primo operando) deve essere un registro a 16 o 32 bit. La sorgente (secondo operando) deve essere un registro a 8 o 16 bit oppure un byte o una word di memoria. L'altra restrizione e' che la destinazione deve essere piu' grande della sorgente (Molte istruzioni richiedono invece che la destinazione e la sorgente siano della stessa dimensione.) Ecco alcuni esempi:

```
movzx  eax, ax      ; estende ax in eax
movzx  eax, al      ; estende al in eax
movzx  ax, al       ; estende al in ax
movzx  ebx, ax      ; estende ax in ebx
```

Per i numeri con segno, non ci sono metodi semplici per usare l'istruzione `MOV` in tutti i casi. L'8086 forniva diverse istruzioni per estendere i numeri con segno. L'istruzione `CBW` (Converte Byte in Word) estende il segno del registro AL in AX. Gli operandi sono impliciti. L'istruzione `CWD` (Converte Word in Double word) estende il segno del registro AX in DX:AX. La notazione DX:AX indica di pensare ai registri DX e AX come ad un'unico registro a 32 bit con i 16 bit superiori in DX e 16 bit inferiori in AX. (Ricorda che l'8086 non aveva i registri a 32 bit!) L'80386 aggiungeva diverse nuove istruzioni. L'istruzione `CWDE` (Converte word in double word) estende il segno di AX in EAX. L'istruzione `CDQ`(Converte Double word in Quad

```
unsigned char uchar = 0xFF;
signed char   schar = 0xFF;
int a = (int) uchar;      /* a = 255 (0x000000FF) */
int b = (int) schar;      /* b = -1 (0xFFFFFFFF) */
```

Figura 2.1:

```
char ch;
while( (ch = fgetc(fp)) != EOF ) {
  /* fai qualcosa con ch */
}
```

Figura 2.2:

word) estende il segno di EAX in EDX:EAX (64 bit!). Infine, L'istruzione `MOVSX` , che funziona come `MOVZX` , ma utilizza le regole per i numeri con segno.

Applicazione alla programmazione in C

ANSI C non definisce se il tipo `char` *e' con segno o senza. Sta ad ogni compilatore decidere. Per questo in Figura 2.1 e' esplicitamente definito il tipo.*

L'estensione degli interi con segno e senza segno si verifica anche in C. Le variabili in C possono essere dichiarate con o senza segno (`int` e' con segno). Consideriamo il codice in Figura 2.1. Alla riga 3 la variabile `a` e' estesa utilizzando le regole per i valori senza segno (utilizzando `MOVZX`), mentre alla riga 4, le regole per i numeri con segno sono utilizzate per `b` (utilizzando `MOVSX`).

Esiste un bug comune nella programmazione in C che riguarda direttamente questo punto. Consideriamo il codice in Figura 2.2. Il prototipo di `fgetc()`e':

```
int fgetc( FILE * );
```

Ci si potrebbe chiedere perche' la funzione ritorna un `int` dal momento che legge un carattere? La ragione e' che normalmente questa funzione ritorna un `char` (esteso ad `int` utilizzando l'estensione a zero). Invece, c'e' un unico valore che viene restituito che non e' un carattere, `EOF`. Questo e' una macro che generalmente e' definita -1. Quindi, `fgetc()` ritorna o un `char` esteso ad `int` (`000000`xx in hex) oppure `EOF` (`FFFFFFFF` in hex).

Il problema di base col programma in Figura 2.2 e' che `fgetc()` ritorna un `int`, ma il suo valore e' memorizzato in un `char`. C tronchera' i bit di ordine piu' alto per aggiustare il valore dell' `int` nel `char`. L'unico problema e' che i numeri (in Hex) `000000FF` e `FFFFFFFF` saranno entrambi troncati ad

un byte `FF`. In questo modo pero' il loop while non riesce a distinguere tra la lettura del carattere `FF` dal file oppure la fine del file.

Cosa fa esattamente il codice in questo caso, dipende se `char` e' un numero con segno oppure senza segno. Perche'? Perche' alla riga 2, `ch` e' comparato a `EOF`. Dal momento che `EOF` e' un valore `int` [1], `ch` sara' esteso ad un `int` in modo che i due valori da comparare sono della stessa dimensione. [2]. Come mostra la Figura 2.1, e' importante se una variabile e' con segno oppure senza segno.

Se il `char` e' senza segno, `FF` e' esteso a `000000FF`. Questo e' comparato con `EOF` (`FFFFFFFF`) e non viene trovato uguale. Cosi' il loop non finisce mai!.

Se il `char` e' con segno, `FF` e' esteso a `FFFFFFFF`. In questo caso, la comparazione da' uguaglianza e il loop termina. Putroppo, dal momento che il byte `FF` potrebbe essere letto anche dentro il file, il loop potrebbe terminare prematuramente.

La soluzione per questo problema e' definire la variabile `ch` come un `int`, non come un `char`. In questo modo, nessun troncamento viene fatto alla riga 2. Dentro il loop, e' sicuro troncare il valore dal momento che `ch` *deve* essere un byte la' dentro.

2.1.3 Aritmetica del complemento a 2

Come visto prima, L'istruzione `add` effettua somme mentre l'istruzione `sub` esegue sottrazioni. Due dei bit del registro FLAGS che queste istruzioni impostano sono i flag di *overflow* e di *carry*. Il flag di overflow e' impostato quando il risultato di una operazione e' troppo grande per stare nella destinazione, nella aritmetica con segno. Il flag di carry e' impostato se una addizione ha generato un riporto nel suo msb oppure se una sottrazione ha generato un resto nel suo msb. Cio' puo' essere usato per determinare l'overflow per l'aritmetica con segno. L'uso del flag di carry per l'aritmetica con segno sara' visto tra breve. Uno dei grossi vantaggi del complemento a due e' che le regole per l'addizione e la sottrazione sono esattamente le stesse come per gli interi senza segno. Ne deriva che `add` e `sub` possono essere usati per gli interi con e senza segno.

$$\begin{array}{rr} 002C & 44 \\ +\ \text{FFFF} & +\ (-1) \\ \hline 002B & 43 \end{array}$$

Questo e' un carry generato,ma non fa parte della risposta.

[1] E' un comune credenza che i file abbiano un carattere EOF alla loro fine. Questo *non* e' vero!

[2] La ragione di questo requisito sara' visto dopo.

dest	source1	source2	Action
	reg/mem8		AX = AL*source1
	reg/mem16		DX:AX = AX*source1
	reg/mem32		EDX:EAX = EAX*source1
reg16	reg/mem16		dest *= source1
reg32	reg/mem32		dest *= source1
reg16	immed8		dest *= immed8
reg32	immed8		dest *= immed8
reg16	immed16		dest *= immed16
reg32	immed32		dest *= immed32
reg16	reg/mem16	immed8	dest = source1*source2
reg32	reg/mem32	immed8	dest = source1*source2
reg16	reg/mem16	immed16	dest = source1*source2
reg32	reg/mem32	immed32	dest = source1*source2

Tabella 2.2: Istruzioni `imul`

Ci sono due diverse istruzioni di moltiplicazione e divisione. Innanzitutto per moltiplicare si usa sia l'istruzione `MUL` che l'istruzione `IMUL`. L'istruzione e' usata per moltiplicare numeri senza segno e l'istruzione `IMUL` e' utilizzata per moltiplicare gli interi con segno. Perche' c'e' bisogno di istruzioni diverse? Le regole per la moltiplicazione sono diverse per i numeri senza segno e quelli con segno in complemento a 2. Perche'? Consideriamo la moltiplicazione del byte FF con se stessa che abbia come risultato una word. Utilizzando la moltiplicazione senza segno, si ha 255 per 255, o 65025 (o FE01 in hex). Utilizzando la moltiplicazione con segno, sia ha -1 per -1 o 1 (0001 in hex).

Ci sono diverse forme per le istruzioni di moltiplicazione. La piu' vecchia e' la seguente:

```
mul   sorgente
```

La *sorgente* puo' essere sia un registro che un riferimento alla memoria. Non puo' essere un valore immediato. La dimensione dell'operando sorgente determina come e' eseguita la moltiplicazione. Se l'operando e' un byte, viene moltiplicato per il byte nel registro AL e il risultato e' memorizzato nei 16 bit di EAX. Se la sorgente e' a 16 bit, viene moltiplicato per la word in AX e il risultato a 32 bit e' memorizzato in DX:AX. Se la sorgente e' a 32 bit, viene moltiplicato per EAX e il risultato a 64 bit e' memorizzato in EDX:EAX.

L'istruzione `IMUL` ha gli stessi formati di `MUL`, ed inoltre aggiunge i formati di altre istruzioni. Ci sono formati a due e tre operandi:

```
imul   dest, source1
imul   dest, source1, source2
```

La tabella 2.2 mostra le possibili combinazioni.

I due operatori di divisione sono `DIV` e `IDIV`.Questi eseguono la divisione tra interi senza segno e con segno rispettivamente. Il formato generale e':

```
div   source
```

Se l'operando source e' a 8 bit, allora AX e' diviso per l'operando. Il quoziente e' memorizzato in AL e il resto in AH. Se source e' a 16 bit, allora DX:AX e' diviso per l'operatore. Il quoziente e' memorizzato in AX e il resto in DX. Se source e' a 32 bit, infine, EDX:EAX e' diviso per l'operando e il quoziente e' memorizzato in EAX e il resto in EDX. L'istruzione `IDIV` funziona nello stesso modo. Non ci sono istruzioni speciali `IDIV` come quella `IMUL`. Se il quoziente e' troppo grande per stare nel suo registro o se il divisore e' 0, il programma e' interrotto e terminato. Un'errore molto comune e' dimenticare di inizializzare DX o EDX prima della divisione.

L'istruzione `NEG` nega il suo singolo operando, calcolando il suo complemento a 2. L'operando puo' essere un registro o una locazione di memoria di 8,16,o 32 bit.

2.1.4 Programma esempio

math.asm

```
1  %include "asm_io.inc"
2  segment .data         ; stringhe di Output
3  prompt          db    "Inserisci un numero: ", 0
4  square_msg      db    "Il quadrati dell'input e' ", 0
5  cube_msg        db    "Il cubo dell'input e' ", 0
6  cube25_msg      db    "Il cubo dell'input per 25 e' ", 0
7  quot_msg        db    "Il quoziente dell'input/100 e' ", 0
8  rem_msg         db    "Il resto del cubo/100 e' ", 0
9  neg_msg         db    "La negazione del resto e' ", 0
10
11 segment .bss
12 input   resd 1
13
14 segment .text
15         global  _asm_main
16 _asm_main:
17         enter   0,0               ; setup routine
18         pusha
19
```

```
20          mov     eax, prompt
21          call    print_string
22
23          call    read_int
24          mov     [input], eax
25
26          imul    eax               ; edx:eax = eax * eax
27          mov     ebx, eax          ; memorizza la risposta in ebx
28          mov     eax, square_msg
29          call    print_string
30          mov     eax, ebx
31          call    print_int
32          call    print_nl
33
34          mov     ebx, eax
35          imul    ebx, [input]      ; ebx *= [input]
36          mov     eax, cube_msg
37          call    print_string
38          mov     eax, ebx
39          call    print_int
40          call    print_nl
41
42          imul    ecx, ebx, 25      ; ecx = ebx*25
43          mov     eax, cube25_msg
44          call    print_string
45          mov     eax, ecx
46          call    print_int
47          call    print_nl
48
49          mov     eax, ebx
50          cdq                       ; inizializza edx con l'estenzione di segno
51          mov     ecx, 100          ; non si puo' dividere per un valore immediato
52          idiv    ecx               ; edx:eax / ecx
53          mov     ecx, eax          ; salva il quoziente in ecx
54          mov     eax, quot_msg
55          call    print_string
56          mov     eax, ecx
57          call    print_int
58          call    print_nl
59          mov     eax, rem_msg
60          call    print_string
61          mov     eax, edx
```

```
62          call    print_int
63          call    print_nl
64
65          neg     edx             ; nega il resto
66          mov     eax, neg_msg
67          call    print_string
68          mov     eax, edx
69          call    print_int
70          call    print_nl
71
72          popa
73          mov     eax, 0          ; ritorna il controllo al C
74          leave
75          ret
```
math.asm

2.1.5 Aritmetica con precisione estesa

Il linguaggio assembly fornisce le istruzioni che permettono di eseguire somme e sottrazioni di numeri piu' grandi di una double word. Queste istruzioni usano il carry flag. Come visto prima, sia l'istruzione `ADD` che l'istruzione `SUB` modificano questo flag se viene generato un riporto o un resto rispettivamente. L'informazione memorizzata nel carry flag puo' essere usata per sommare o sottrarre grandi numeri spezzando l'operazione in pezzi piu' piccoli (double word o piu' piccoli).

Le istruzioni `ADC` e `SBB` utilizzano questa informazione del carry flag. L'istruzione `ADC` esegue l'operazione seguente:

operando1 = *operando1* + carry flag + *operando2*

L'istruzione `SBB` invece esegue questa:

operando1 = *operando1* - carry flag - *operando2*

Come sono usate? Consideriamo la somma di interi a 64 bit in EDX:EAX e in EBX:ECX. Il codice seguente memorizza la somma in EDX:EAX:

```
1       add     eax, ecx        ; somma i 32 bit bassi
2       adc     edx, ebx        ; somma i 32 bit alti e il riporto dalla somma precedente
```

La sottrazione e' molto simile. Il codice seguente sottrae EBX:ECX da EDX:EAX:

```
1       sub     eax, ecx        ; sottrae i 32 bit bassi
2       sbb     edx, ebx        ; sottraw i 32 bit alti e il resto
```

Per numeri *davvero* grandi, puo' essere usato un ciclo (vedi la sezione 2.2). Per un ciclo di somma potrebbe essere conveniente usare l'istruzione `ADC` per ogni iterazione (ad eccezione della prima). Per questo si puo' utilizzare l'istruzione `CLC` (Clear Carry), prima di iniziare il ciclo per inizializzare il flag di carry a 0. Se il flag di carry e' 0, non c'e' differenza fra le istruzioni `ADD` e `ADC`. Lo stesso principio puo' essere usato anche per le sottrazioni.

2.2 Strutture di controllo

I linguaggi di alto livello forniscono strutture di controllo (*es.* i comandi *if* e *while*) che controllano il processo di esecuzione. Il linguaggio assembly non fornisce questo tipo di strutture complesse: Usa invece il comando *goto*, tristemente noto che se usato impropriamente puo' generare *spaghetti code*![3] *E'* comunque possibile scrivere programmi strutturati in linguaggio assembly. La procedura di base e' quella di definire la logica del programma utilizzando un linguaggio di alto livello conosciuto e tradurre il risultato nell'appropriato linguaggio assembly (un po' come fanno i compilatori).

2.2.1 Comparazioni

Le strutture di controllo decidono cosa fare in base alla comparazione dei dati. In assembly, il risultato di una comparazione e' memorizzata nel registro FLAGS per essere utilizzato piu' tardi. L'80x86 fornisce l'istruzione `CMP` per eseguire comparazioni. Il registro FLAGS e' impostato in base alla differenza dei due operandi della istruzione `CMP`. Gli operandi sono sottratti e il FLAGS e' impostato sulla base del risultato, ma il risultato *non* e' memorizzato da nessuna parte. Se occorre utilizzare il risultato, si utilizzi l'istruzione `SUB` invece di `CMP`.

Per gli interi senza segno, ci sono 2 flag (bit nel registro FLAGS) che sono importanti: lo Zero flag (ZF) e il carry flag . Lo Zero flag e' settato (1) se la differenza risultante e' 0. Il carry flag e' usato come flag di prestito nella sottrazione. Considera la comparazione seguente:

```
cmp    vleft, vright
```

Viene calcolata la differenza di `vleft - vright` e i flag sono impostati di conseguenza. Se la differenza del `CMP` e' zero, cioe' `vleft = vright`, allora ZF e' impostato (*i.e.* 1) e il CF no (*i.e.* 0). Se `vleft > vright` allora sia ZF che CF non sono impostati(nessun resto). Se `vleft < vright` allora ZF non e' settato mentre CF e' settato (resto).

Per gli interi con segno, ci sono tre flag che sono importanti: lo zero (ZF) flag, l' overflow (OF) flag e lo sign (SF) flag. L'overflow flag e' impostato se il

Perche' SF = OF se `vleft > vright`*? Se non c'e' overflow, la differenza avra' un risultato corretto e dovra' non essere negativa. Cosi', SF = OF = 0. Invece, se c'e' un overflow, la differenza non avra' un valore corretto (ed infatti sara' negativa). Cosi', SF = OF = 1*

[3] Spaghetti code e' un termine gergale non traducibile che indica codice scritto in maniera incomprensibile (ndt)

risultato di una operazione "trabocca"(overflow) o e' troppo piccolo (underflow). Il sign flag e' impostato se il risultato di una operazione e' negativo. Se `vleft = vright`, ZF e' impostato (come per gli interi senza segno). Se `vleft > vright`, ZF non e' impostato e SF = OF. Se `vleft < vright`, ZF non e' impostato e SF $\neq$ OF.

Non dimenticare che altre istruzioni possono cambiare il registro FLAGS, non solo `CMP`.

2.2.2 Istruzioni di salto

Le istruzioni di salto possono trasferire l'esecuzione a punti arbitrari di un programma. In altre parole, si comportano come un *goto*. Ci sono due tipi di salto: incondizionati e condizionati. Un salto condizionato puo' eseguire o no un percorso dipendentemente dai flag del registro FLAGS. Se il salto condizionato non esegue quel percorso, il controllo passa alla successiva istruzione.

L'istruzione `JMP` (abbreviazione di *jump*) crea ramificazioni incondizionate. Il suo solo operando e' solitamente una *etichetta di codice* alla istruzione a cui saltare. L'assembler o il linker sostituiranno l'etichetta con il corretto indirizzo della istruzione. Questa e' un'altra delle operazioni noiose che l'assembler esegue per rendere piu' facile la vita del programmatore. E' importante capire che il comando immediatamente dopo l'istruzione `JMP` non sara' mai eseguito a meno che un'altra istruzione non salti a esso!

Ci sono diverse varianti dell'istruzione di salto:

SHORT Questo salto e' molto limitato come raggio. Puo' spostare l'esecuzione solo di 128 bytes. Il vantaggio di questo salto e' che usa meno memoria rispetto agli altri. Usa un solo byte con segno per memorizzare il *dislocamento* del salto. Il dislocamento e' la misura che indica di quanti byte occorre muoversi avanti o indietro. (Il dislocamento e' sommano al registro EIP). Per specificare un salto di tipo short, si usa la parola chiave `SHORT` subito prima dell'etichetta nell'istruzione `JMP`.

NEAR Questo salto e' il tipo di default per le istruzioni di salto condizionate e non condizionate e puo' essere usato per saltare a qualunque punto all'interno del segmento. L'80386 supporta due tipi di salto near. Il primo usa 2 byte per il dislocamento. Questo permette di spostarsi in avanti o indietro di 32.000 byte. L'altro tipo usa 4 byte per il dislocamento, che naturalmente permette di spostarsi in qualsiasi punto del segmento. Il tipo a 4 byte e' quello di default in modalita' protetta 386. Il tipo a 2 byte puo' essere usato, inserendo la parola chiave `WORD` prima della etichetta nella istruzione `JMP`.

JZ	salta solo se ZF e' impostato
JNZ	salta solo se ZF non e' impostato
JO	salta solo se OF e' impostato
JNO	salta solo se OF non e' impostato
JS	salta solo se SF e' impostato
JNS	salta solo se SF non e' impostato
JC	salta solo se CF e' impostato
JNC	salta solo se CF non e' impostato
JP	salta solo se PF e' impostato
JNP	salta solo se PF non e' impostato

Tabella 2.3: Salti condizionati semplici

FAR Questo salto permette di spostarsi da un segmento di codice ad un'altro. Questa e' una operazione molto rara da fare in modalita' protetta 386.

Le etichette di codice valide seguono le stesse regole delle etichette dei dati. Le etichette di codice sono definite posizionandole nel segmento codice di fronte al comando che contrassegnano. Al nome dell'etichetta, nella definizione, vengono aggiunti i due punti (":"). I due punti *non* fanno parte del nome.

Ci sono molte istruzioni differenti di salto condizionato. Anche queste hanno come unico operando una etichetta di codice. Il piu' semplice controlla un solo flag nel registro FLAGS per determinare se effettuare il salto o meno. Vedi la Tabella 2.3 per la lista di queste istruzioni. (PF e' il *flag di parita'* che indica la parita' o meno del numero di bit impostati negli 8 bit bassi del risultato.)

Il seguente Pseudo codice:

```
if ( EAX == 0 )
  EBX = 1;
else
  EBX = 2;
```

potrebbe essere scritto in assembly come:

```
1        cmp    eax, 0            ; imposta i flags (ZF impostato se eax - 0 = 0)
2        jz     thenblock         ; se ZF e' 1, salta a thenblock
3        mov    ebx, 2            ; parte ELSE dell'IF
4        jmp    next              ; salta alla parte THEN di IF
5  thenblock:
6        mov    ebx, 1            ; parte THEN di IF
7  next:
```

Con Segno		Senza Segno	
JE	salta se `vleft = vright`	JE	salta se `vleft = vright`
JNE	salta se `vleft` $\neq$ `vright`	JNE	salta se `vleft` $\neq$ `vright`
JL, JNGE	salta se `vleft < vright`	JB, JNAE	salta se `vleft < vright`
JLE, JNG	salta se `vleft` $\leq$ `vright`	JBE, JNA	salta se `vleft` $\leq$ `vright`
JG, JNLE	salta se `vleft > vright`	JA, JNBE	salta se `vleft > vright`
JGE, JNL	salta se `vleft` $\geq$ `vright`	JAE, JNB	salta se `vleft` $\geq$ `vright`

Tabella 2.4: Istruzioni di comparazione con segno e senza segno

Altre comparazione non sono cosi' semplici utilizzando i salti condizionati della Tabella 2.3. Per illustrarle, consideriamo il seguente Pseudo codice:

```
if ( EAX >= 5 )
  EBX = 1;
else
  EBX = 2;
```

Se EAX e' maggiore o uguale a 5, lo ZF potrebbe essere impostato o meno e SF sarebbe uguale a OF. Ecco il codice assembly che prova queste condizioni (assumiamo che EAX sia con segno):

```
1        cmp    eax, 5
2        js     signon            ; va a signon se SF = 1
3        jo     elseblock         ; va a elseblock se OF = 1 e SF = 0
4        jmp    thenblock         ; va a thenblock se SF = 0 e OF = 0
5   signon:
6        jo     thenblock         ; va a thenblock se SF = 1 e OF = 1
7   elseblock:
8        mov    ebx, 2
9        jmp    next
10  thenblock:
11       mov    ebx, 1
12  next:
```

Il codice sopra e' molto contorto. Fortunatamente, l'80x86 fornisce ulteriori istruzioni di salto per rendere questo tipo di prove *piu'* semplici da realizzare. Ci sono versioni di ogni comando per numeri con segno e per i numeri senza segno. La Tabella 2.4 mostra queste istruzioni. Le istruzioni di uguale o non uguale (JE e JNE) sone le stesse sia per gli interi con segno e che senza segno. (Infatti, JE e JNE sono in realta' identici rispettivamente a JZ e JNZ). Ognuna delle altre istruzioni ha due sinonimi. Per esempio, si puo' vedere JL (jump less than, salta a se minore di) e JNGE (Jump not

greater than or equal to, salta se non maggiore o uguale di). Queste sono la stessa istruzione perche':

$$x < y \Longrightarrow \mathbf{not}(x \geq y)$$

Le istruzioni senza segno usano A per *sopra* e B per *sotto* invece di L e G.

Utilizzando queste nuove istruzioni di salto, lo pseudo codice puo' essere tradotto in assembly piu' facilmente.

```
1        cmp    eax, 5
2        jge    thenblock
3        mov    ebx, 2
4        jmp    next
5  thenblock:
6        mov    ebx, 1
7  next:
```

2.2.3 Le istruzioni iterative

L'80x86 fornisce diverse istruzioni definite per implementare cicli di tipo *for*. Ognuna di queste istruzioni accetta come operando una etichetta di codice.

LOOP Decrementa ECX, se ECX $\neq 0$, salta alla etichetta

LOOPE, LOOPZ Decrementa ECX (il registro FLAGS non e' modificato), se ECX $\neq 0$ e ZF $= 1$, salta

LOOPNE, LOOPNZ Decrementa ECX (il FLAGS non modificato), se ECX $\neq 0$ e ZF $= 0$, esegue il salto

Le due ultime istruzioni iterative sono utili per cicli di ricerca sequenziale. Lo pseudo codice seguente:

```
sum = 0;
for( i=10; i >0; i-- )
  sum += i;
```

puo' essere tradotto in assembly cosi':

```
1        mov    eax, 0          ; eax e' sum
2        mov    ecx, 10         ; ecx e' i
3  loop_start:
4        add    eax, ecx
5        loop   loop_start
```

2.3 Tradurre strutture di controllo standard

Questa sezione si occupa di capire come le strutture dei linguaggi di alto livello possono essere implementate in linguaggio assembly.

2.3.1 Istruzione If

Il seguente pseudo:

if (condition)
  then_block;
else
  else_block ;

puo' essere implementato cosi':

```
1        ; codice per impostare FLAGS
2        jxx    else_block    ; seleziona xx in maniera da saltare se la condizione e' falsa
3        ; codice per il blocco then
4        jmp    endif
5  else_block:
6        ; codice per il blocco else
7  endif:
```

Se non c'e' il blocco else, il ramo `else_block` puo' essere sostituito con il ramo `endif`.

```
1        ; codice per impostare FLAGS
2        jxx    endif    ; seleziona xx in maniera da saltare se la condizione e' falsa
3        ; codice per il blocco then
4  endif:
```

2.3.2 Ciclo While

Il ciclo di *while* e' un ciclo a controllo iniziale:

while(condizione) {
  corpo del ciclo ;
}

Questo puo' essere tradotto cosi':

```
1  while:
2        ; codice per impostare FLAGS in base alla condizione
3        jxx    endwhile     ; seleziona xx in maniera che salti se la condizione e' falsa
4        ; body of loop
5        jmp    while
6  endwhile:
```

```
unsigned guess;   /* attuale ipotesi per il numero primo */
unsigned factor;  /* possibile fattore di guess          */
unsigned limit;   /* trova i numeri primi da questo valore */

printf("Find primes up to: ");
scanf("%u", &limit);
printf("2\n");    /* tratta i primi due numeri primi come */
printf("3\n");    /* casi speciali */
guess = 5;        /* ipotesi */
while ( guess <= limit ) {
  /* cerca il fattore di guess */
  factor = 3;
  while ( factor*factor < guess &&
          guess % factor != 0 )
    factor += 2;
  if ( guess % factor != 0 )
    printf("%d\n", guess);
  guess += 2;   /* cerca sono i numeri dispari */
}
```

Figura 2.3:

2.3.3 Ciclo Do..while

Il ciclo *do..while* e' a controllo finale:

```
do {
  corpo del loop ;
} while( condizione );
```

Questo puo' essere tradotto cosi':

```
1 do:
2       ; corpo del ciclo
3       ; codice per impostare FLAGS in base alla condizione
4       jxx    do      ; seleziona xx in maniera che salti se la condizione e' vera
```

2.4 Esempio: Trovare i numeri primi

Questa sezione analizza un programma che trova i numeri primi. Ricordiamo che i numeri primi sono divisibili solo per 1 o per se stessi. Non ci sono formule per fare cio'. L'algoritmo che usa questo programma trova i

fattori di tutti i numeri dispari[4] sotto un dato limite. La Figura 2.3 mostra l'algoritmo scritto in C.

Ecco la versione in assembly:

prime.asm

```
%include "asm_io.inc"
segment .data
Message         db      "Find primes up to: ", 0

segment .bss
Limit           resd    1               ; trova i numeri primi fino a questo limite
Guess           resd    1               ; l'ipotesi corrente del numero primo

segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; routine di setup
        pusha

        mov     eax, Message
        call    print_string
        call    read_int            ; scanf("%u", & limit );
        mov     [Limit], eax

        mov     eax, 2              ; printf("2\n");
        call    print_int
        call    print_nl
        mov     eax, 3              ; printf("3\n");
        call    print_int
        call    print_nl

        mov     dword [Guess], 5    ; Guess = 5;
while_limit:                        ; while ( Guess <= Limit )
        mov     eax,[Guess]
        cmp     eax, [Limit]
        jnbe    end_while_limit     ; usa jnbe visto che i numeri sono senza segno

        mov     ebx, 3              ; ebx e' il fattore = 3;
while_factor:
        mov     eax,ebx
        mul     eax                 ; edx:eax = eax*eax
        jo      end_while_factor    ; if answer won't fit in eax alone
```

[4] 2 e' l'unico numeri primo pari.

```
38          cmp     eax, [Guess]
39          jnb     end_while_factor     ; if !(factor*factor < guess)
40          mov     eax,[Guess]
41          mov     edx,0
42          div     ebx                  ; edx = edx:eax % ebx
43          cmp     edx, 0
44          je      end_while_factor     ; if !(guess % factor != 0)
45
46          add     ebx,2                ; factor += 2;
47          jmp     while_factor
48  end_while_factor:
49          je      end_if               ; if !(guess % factor != 0)
50          mov     eax,[Guess]          ; printf("%u\n")
51          call    print_int
52          call    print_nl
53  end_if:
54          add     dword [Guess], 2     ; guess += 2
55          jmp     while_limit
56  end_while_limit:
57
58          popa
59          mov     eax, 0            ; ritorna al C
60          leave
61          ret
```

prime.asm

Capitolo 3

Operazioni sui Bit

3.1 Operazioni di Shift

Il linguaggio assembly permette al programmatore di manipolare i singoli bit di un dato. Una operazione comune sui bit e' chiamata *shift*. Una operazione di shift sposta la posizione dei bit di alcuni dati. Gli shift possono essere verso sinistra (*i.e.* verso i bit piu' significativi) o verso destra (verso i bit meno significativi).

3.1.1 Shift Logici

Uno shift logico e' il piu' semplice tipo di shift: sposta i bit in maniera diretta. La Figura 3.1 mostra un'esempio di shift su un singolo byte.

Originale	1	1	1	0	1	0	1	0
shift a sinistra	1	1	0	1	0	1	0	0
shift a destra	0	1	1	1	0	1	0	1

Figura 3.1: Shift Logici

Nota che i nuovi bit aggiunti sono sempre zero. Le istruzioni `SHL` e `SHR` sono usate per eseguire shift logici a sinistra e a destra rispettivamente. Queste istruzioni permettono di spostare di qualunque numero di posizioni. Il numero di posizioni da spostare puo' essere sia una costante che un numero memorizzato nel registro `CL`. L'ultimo bit spostato fuori da un dato e' memorizzato nel carry flag. Ecco alcuni esempi di codice:

```
1  mov    ax, 0C123H
2  shl    ax, 1          ; sposta 1 bit a sinistra,   ax = 8246H, CF = 1
3  shr    ax, 1          ; sposta 1 bit a destra,  ax = 4123H, CF = 0
4  shr    ax, 1          ; sposta 1 bit a destra,  ax = 2091H, CF = 1
5  mov    ax, 0C123H
```

```
6       shl    ax, 2          ; sposta 2 bits a sinistra,  ax = 048CH, CF = 1
7       mov    cl, 3
8       shr    ax, cl         ; sposta 3 bits a destra, ax = 0091H, CF = 1
```

3.1.2 Uso degli Shift

[1]

L'uso piu' comune delle operazioni di shift sono le moltiplicazioni e divisioni veloci. Ricorda che nel sistema decimale, la moltiplicazione e la divisione per una potenza del 10 sono semplici, si tratta solo di spostare le cifre. Lo stesso e' vero per le potenze del 2 in binario. Per esempio, per raddoppiare il numero binario 1011_2 (o 11 in decimale) , basta spostare di una posizione a sinistra per ottenere 10110_2 (o 22 in decimale). Il quoziente della divisione per una potenza del 2 e' il risultato di uno shift a destra. Per dividere per 2, si usa un singolo shift a destra; per dividere per 4 (2^2), si usa uno shift a destra di 2 posizioni; per dividere per 8 (2^3), uno shift a destra di 3 posizioni, *ecc.* Le istruzioni di shift sono molto semplici e sono *molto* piu' veloci delle corrispondenti istruizioni `MUL` e `DIV` !

In realta', gli shift logici possono essere usati per moltiplicare e dividere valori senza segno. Questi non funzionano generalmente con i valori con segno. Consideriamo il valore a 2 byte FFFF (-1 con segno). Se questo viene spostato di una posizione a destra, il risultato e' 7FFF, cioe' $+32,767$! Per i valori con segno si usa un'altro tipo di shift.

3.1.3 Shift aritmetici

Questi shift sono progettati per permettere la moltiplicazione e la divisione per potenze del 2 in maniera piu' veloce dei numeri con segno. Questi si assicurano che il bit di segno sia trattato correttamente.

SAL Shift Aritmetico a sinistra - Questa istruzione e' un sinonimo per `SHL`. E' assemblato nello stesso codice macchina di `SHL`. Da momento che il bit di segno non e' cambiato, il risultato e' corretto.

SAR Shift Aritmetico a destra - Questa e' una nuova istruzione che non sposta il bit di segno (*i.e.* il msb[2]) del suo operando. Gli altri bit sono spostati come al solito ad eccezione del fatto che i nuovi bit che entrano a sinistra sono copie del bit di segno (Cosi' se il bit di segno e' 1, i nuovi bit saranno 1). In questo modo, se un byte e' soggetto a shift con questa istruzione, solo i 7 bit piu' bassi sono spostati. Come

[1] In questo capitolo e nei successivi il termine *spostamento* puo' essere usato al posto di *shift* e rappresenta un suo sinonimo(ndt).

[2] Most Significant Bit - il bit piu' significativo

per gli altri shift, l'ultimo bit spostato fuori e' memorizzato nel carry flag.

```
1  mov    ax, 0C123H
2  sal    ax, 1          ; ax = 8246H, CF = 1
3  sal    ax, 1          ; ax = 048CH, CF = 1
4  sar    ax, 2          ; ax = 0123H, CF = 0
```

3.1.4 Shift di rotazione

Le istruzioni di shift di rotazione funzionano come quelli logici ad eccezione del fatto che i bit che escono da una parte del numero sono spostati nell'altra parte. I dati, cioe', sono trattati come se avessero una struttura circolare. Le due istruzioni di shift di rotazione piu' semplici sono `ROL` e `ROR` che effettuano rotazioni a sinistra e a destra, rispettivamente. Come per gli altri shift, questi shift lasciano una copia dell'ultimo bit ruotato nel carry flag.

```
1  mov    ax, 0C123H
2  rol    ax, 1          ; ax = 8247H, CF = 1
3  rol    ax, 1          ; ax = 048FH, CF = 1
4  rol    ax, 1          ; ax = 091EH, CF = 0
5  ror    ax, 2          ; ax = 8247H, CF = 1
6  ror    ax, 1          ; ax = C123H, CF = 1
```

Ci sono due ulteriori istruzioni di shift di rotazione che spostano i bit nel dato e nel carry flag, chiamate `RCL` e `RCR`. Per esempio, se il registro `AX` e' ruotato con queste istruzioni, i 17 bit composti dal registro AX e dal carry flag sono ruotati.

```
1  mov    ax, 0C123H
2  clc                   ; clear the carry flag (CF = 0)
3  rcl    ax, 1          ; ax = 8246H, CF = 1
4  rcl    ax, 1          ; ax = 048DH, CF = 1
5  rcl    ax, 1          ; ax = 091BH, CF = 0
6  rcr    ax, 2          ; ax = 8246H, CF = 1
7  rcr    ax, 1          ; ax = C123H, CF = 0
```

3.1.5 Semplice applicazione

Ecco un piccolo pezzo di codice che conta i numeri di bit che sono "on" (*cioe'* 1) nel registro EAX.

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Tabella 3.1: L'operazione AND

	1	0	1	0	1	0	1	0
AND	1	1	0	0	1	0	0	1
	1	0	0	0	1	0	0	0

Figura 3.2: AND di un byte

```
1       mov    bl, 0            ; bl conterra' il numero dei bit ON
2       mov    ecx, 32          ; ecx e' il contatore di ciclo
3  count_loop:
4       shl    eax, 1           ; sposta i bit nel carry flag
5       jnc    skip_inc         ; se CF == 0, vai a skip_inc
6       inc    bl
7  skip_inc:
8       loop   count_loop
```

Il codice qua sopra distrugge il valore originale di `EAX`(`EAX` e' zero alla fine del ciclo). Se si volesse mantenere il valore di `EAX`, la riga 4 dovrebbe essere cambiato con `rol eax, 1`.

3.2 Operazioni booleane a livello di bit

Ci sono 4 operatori booleani: *AND*, *OR*, *XOR* e *NOT*. Una *tavola delle verita'* mostra il risultato di ogni operazione per ogni possibile valore del suo operando.

3.2.1 L'operazione *AND*

Il risultato di un *AND* di due bit e' sempre 1 se entrambi i bit sono 1 altrimenti sara' sempre 0 come mostra la tavola delle verita' in Tabella 3.1.

I processori gestiscono queste operazioni come istruzioni che lavorano indipendentemente su tutti i bit dei dati in parallelo. Per esempio, se il contenuto di `AL` e di `BL` vengono sottoposti ad *AND* l'un l'altro, l'operazione base *AND* e' applicata ad ognuna delle 8 coppie dei corrispondenti bit nei due registri come mostra la Figura 3.2. Sotto il codice di esempio:

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Tabella 3.2: L'operazione OR

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 3.3: L'operazione XOR

```
1        mov    ax, 0C123H
2        and    ax, 82F6H          ; ax = 8022H
```

3.2.2 L'operazione *OR*

L'*OR* inclusivo di 2 bit e' 0 se entrambi i bit sono 0, altrimenti e' 1 come mostra la tavola delle verita' in Tabella 3.2. Ecco il codice esempio:

```
1        mov    ax, 0C123H
2        or     ax, 0E831H          ; ax = E933H
```

3.2.3 L'operazione *XOR*

L'*OR* esclusivo di 2 bit e' 0 se e solo se entrambi i bit sono uguali, altrimenti il risultato e' 1, come e' mostrato nella tavola delle verita' in Figura 3.3. Ecco il codice esempio:

```
1        mov    ax, 0C123H
2        xor    ax, 0E831H          ; ax = 2912H
```

3.2.4 L'operazione *NOT*

L'operazione *NOT* e' una operazione *unaria* (*i.e.* questa agisce su di un solo operando, non due come le operazioni *binarie* come *AND*). Il *NOT* di un bit e' il suo valore opposto come mostra la tavola delle verita' in Figura 3.4. Ecco il codice esempio:

```
1        mov    ax, 0C123H
2        not    ax                  ; ax = 3EDCH
```

X	NOT X
0	1
1	0

Tabella 3.4: L'operazione NOT

metti a on il bit i	esegue OR del numero con 2^i (che e' il il numero binario con cui metto a on il bit i)
metti a off il bit i	esegue AND del numero con il numero binario con il solo bit i off. Questo operando e' spesso chiamato una *maschera*
metti a complemento il bit i	esegui XOR del numero con 2^i

Tabella 3.5: Uso delle operazioni booleane

Nota che NOT trova il complemento a 1. Diversamente dalle altre operazioni a livello di bit, l'istruzione NOT non modifica nessuno dei bit del registro `FLAGS`.

3.2.5 L'istruzione `TEST`

L'istruzione `TEST` esegue una operazione AND, ma non memorizza il risultato. Si limita a settare il registro `FLAGS` sulla base di quello che sarebbe il risultato (come fa l'istruzione `CMP` che esegue una sottrazione ma setta solamente `FLAGS`. Per esempio, se il risultato fosse zero, `ZF` sarebbe settato.

3.2.6 Uso delle operazioni sui bit

Le operazioni sui bit sono molto utili per la manipolazione individuale dei bit di una dato senza modificare gli altri bit. La tabella 3.5 mostra tre usi comuni di queste operazione. Sotto il codice esempio che sviluppa questa idea.

```
1  mov    ax, 0C123H
2  or     ax, 8          ; a on il bit 3,     ax = C12BH
3  and    ax, 0FFDFH     ; a off il bit 5,    ax = C10BH
4  xor    ax, 8000H      ; inverte il bit 31, ax = 410BH
5  or     ax, 0F00H      ; a on il nibble,    ax = 4F0BH
6  and    ax, 0FFF0H     ; a off il nibble,   ax = 4F00H
7  xor    ax, 0F00FH     ; inverte il nibbles,ax = BF0FH
8  xor    ax, 0FFFFH     ; complemento a 1,    ax = 40F0H
```

L'operazione AND puo' essere usata anche per trovare il resto di una divisione per una potenza del 2. Per trovare il resto di una divisione per 2^i,

occorre eseguire una *AND* del numero con una maschera uguale a $2^i - 1$. Questa maschera conterra' 1 dal bit 0 al bit $i - 1$. E sono appunto questi bit che contengono il resto. Il risultato dell'*AND* manterra' questi bit e mettera' a zero tutti gli altri. Di seguito un piccolo pezzo di codice che trova il quoziente ed il resto della divisione di 100 per 16.

```
1  mov    eax, 100         ; 100 = 64H
2  mov    ebx, 0000000FH   ; mask = 16 - 1 = 15 or F
3  and    ebx, eax         ; ebx = remainder = 4
```

Utilizzando il registro *CL* e' possibile modificare arbitrariamente i bit dei dati. Di seguito un esempio che imposta (mette a on) un bit arbitrario nel registro `EAX`. Il numero del bit da impostare e' memorizzato in `BH`.

```
1  mov    cl, bh           ; costruisce il numero per l'OR
2  mov    ebx, 1
3  shl    ebx, cl          ; sposta a sinistra cl volte
4  or     eax, ebx         ; mette a on il bit
```

Mettere a off un bit e' un po' piu' difficile.

```
1  mov    cl, bh           ; costruisce il numero per l'AND
2  mov    ebx, 1
3  shl    ebx, cl          ; sposta a sinistra cl volte
4  not    ebx              ; inverte i bit
5  and    eax, ebx         ; mette a off il bit
```

Il codice per il complemento di un bit arbitrario e' lasciato al lettore come esercizio.

Non e' molto raro vedere la seguente istruzione "strana" in un programma per 80x86:

```
xor    eax, eax          ; eax = 0
```

Uno *XOR* di un numero con se stesso produce sempre un risultato di zero. Questa istruzione e' usata perche' il corrispondente codice macchina e' piu' piccolo della corrispondente istruzione `MOV`.

3.3 Evitare i salti condizionati

I moderni processori usano tecniche molto sofisticate per eseguire codice il piu' veloce possibile. Una tecnica comune e' conosciuta come *esecuzione speculativa*. Questa tecnica usa le capacita' della CPU di processare in parallelo l'esecuzione di molte istruzioni alla volta. I percorsi condizionati presentano da questo punto di vista dei problemi. Il processore, in generale,

```
1       mov    bl, 0           ; bl conterra' il numero di bit ON
2       mov    ecx, 32         ; ecx e' il contatore del ciclo
3  count_loop:
4       shl    eax, 1          ; sposta i bit nel carry flag
5       adc    bl, 0           ; somma il carry flag a bl
6       loop   count_loop
```

Figura 3.3: Conteggio dei bit con `ADC`

non sa se il percorso condizionato sara' preso o no. E questo comporterra' l'esecuzione di due diversi gruppi di istruzioni, a seconda della scelta. I processori provano a predire se il percorso verra' preso. Se la predizione e' sbagliata, il processore ha perso il suo tempo eseguendo codice sbagliato.

Un modo per evitare questo problema e' evitare di utilizzare i percorsi condizionali quando possibile. Il codice in 3.1.5 fornisce un semplice esempio di come si possa fare. Nell'esempio precedente, sono contati i bit "on" del registro EAX. Usa un salto per saltare l'istruzione `INC`. La figura 3.3 mostra come i percorsi possono essere rimossi utilizzando l'istruzione `ADC` per sommare il carry flag direttamente.

Le istruzioni `SET`*xx* forniscono un modo per rimuovere i percorsi in alcuni casi. Queste istruzioni impostano il valore di un registro byte o di una locazione di memoria a zero o uno sulla base dello stato del registro FLAGS. I caratteri dopo `SET` sono gli stessi caratteri usati per i salti condizionati. Se la condizione corrispondente di `SET`*xx* e' vera, il risultato memorizzato e' uno, se falso e' memorizzato zero. Per esempio,

```
setz   al        ; AL = 1 se il flag Z e' 1, else 0
```

L'utilizzo di queste istruzioni permette di sviluppare alcune tecniche brillanti per calcolare i valori senza istruzioni di salto.

Per esempio, consideriamo il problema di trovare il massimo tra due valori. L'approccio standard per la risoluzione di questo problema e' quello di usare una istruzione `CMP` e una istruzione di salto per trovare il valore piu' alto. Il programma di esempio mostra come puo' essere trovato il valore massimo senza nessuna istruzione di salto.

```
1  ; file: max.asm
2  %include "asm_io.inc"
3  segment .data
4
5  message1 db "Enter a number: ",0
```

```
6  message2 db "Enter another number: ", 0
7  message3 db "The larger number is: ", 0
8
9  segment .bss
10
11 input1  resd    1        ; primo numero inserito
12
13 segment .text
14         global  _asm_main
15 _asm_main:
16         enter   0,0              ; setup routine
17         pusha
18
19         mov     eax, message1    ; stampa il primo messaggio
20         call    print_string
21         call    read_int         ; inserisci i primo numero
22         mov     [input1], eax
23
24         mov     eax, message2    ; stampa il secondo messaggio
25         call    print_string
26         call    read_int         ; inserisce il secondo numero (in eax)
27
28         xor     ebx, ebx         ; ebx = 0
29         cmp     eax, [input1]    ; compara il primo e il secondo numero
30         setg    bl               ; ebx = (input2 > input1) ?          1 : 0
31         neg     ebx              ; ebx = (input2 > input1) ? 0xFFFFFFFF : 0
32         mov     ecx, ebx         ; ecx = (input2 > input1) ? 0xFFFFFFFF : 0
33         and     ecx, eax         ; ecx = (input2 > input1) ?     input2 : 0
34         not     ebx              ; ebx = (input2 > input1) ?          0 : 0xFFFFFFFF
35         and     ebx, [input1]    ; ebx = (input2 > input1) ?          0 : input1
36         or      ecx, ebx         ; ecx = (input2 > input1) ?     input2 : input1
37
38         mov     eax, message3    ; stampa il risultato
39         call    print_string
40         mov     eax, ecx
41         call    print_int
42         call    print_nl
43
44         popa
45         mov     eax, 0           ; ritorna al C
46         leave
47         ret
```

Il trucco e' creare una maschera di bit che puo' essere usata per selezionare il corretto valore per il massimo. L'istruzione `SETG` alla riga 30 imposta BL a 1 se il secondo input e' il massimo, altrimenti a 0. Questa non e' comunque la maschera di cui abbiamo bisogno. Per creare la maschera desiderata, la riga 31 utilizza l'istruzione `NEG` sull'intero registro EBX. (Nota che EBX era stato azzerato poco prima.) Se EBX e' 0, questo non fa nulla; Invece, se EBX e' 1, il risultato e' la rappresentazione in complemento a 2 di -1 o 0xFFFFFFFF. Questa e' appunto la maschera di bit desiderata. Il codice rimanente utilizza questa maschera per selezionare il corretto input come massimo.

Un trucco alternativo consiste nell'utilizzare il comando `DEC`. Nel codice sopra, se `NEG` e' sostituito con `DEC`, di nuovo il risultato sara' 0 o 0xFFFFFFFF. I valori sono quindi invertiti quando si utilizza l'istruzione `NEG`.

3.4 Manipolazione di bit in C

3.4.1 Gli operatori a livello di bit del C

Diversamente da altri linguaggi di alto livello, il C fornisce operatori per le operazioni a livello di bit. L'operazione *AND* e' rappresentata dell'operatore `&`[3]. L'operazione di *OR* e' rappresentata dall'operatore binario `|`. L'operazione di *XOR* e' rappresentata dall'operatore binario `^`. Infine, l'operazione *NOT* e' rappresentata dall'operatore unario `~` .

Le operazioni di shift sono eseguite in C con gli operatori binari `<<` e `>>`. L'operatore `<<` effettua lo spostamento a sinistra, mentre l'operatore `>>` esegue quello a destra. Questi operatori accettano due operandi. Il primo operando e' il valore su cui effettuare l'operazione, mentre il secondo e' il numero di bit da spostare. Se il valore da shiftare e' di tipo unsigned allora viene eseguito uno shift logico. Se il valore e' di tipo signed (come `int`), viene utilizzato uno shift aritmetico. Sotto sono riportati alcuni esempi in C che utilizzano questi operatori:

```
short int s;          /* si assume che short int e' 16 bit */
short unsigned u;
s = -1;               /* s = 0xFFFF (complemento a 2) */
u = 100;              /* u = 0x0064 */
u = u | 0x0100;       /* u = 0x0164 */
s = s & 0xFFF0;       /* s = 0xFFF0 */
s = s ^ u;            /* s = 0xFE94 */
u = u << 3;           /* u = 0x0B20 (shift logico) */
s = s >> 2;           /* s = 0xFFA5 (shift aritmetico) */
```

[3]Questo operatore e' diverso dall'operatore binario `&&` ed unario `&`!

Macro	Significato
S_IRUSR	utente puo' leggere
S_IWUSR	utente puo' scrivere
S_IXUSR	utente puo' eseguire
S_IRGRP	gruppo puo' leggere
S_IWGRP	gruppo puo' scrivere
S_IXGRP	gruppo puo' eseguire
S_IROTH	altri possono leggere
S_IWOTH	altri possono scrivere
S_IXOTH	altri possono eseguire

Tabella 3.6: Macro di permission sui File POSIX

3.4.2 Utilizzo degli operatori a livello di bit in C

Gli operatori a livello di bit sono usati in C con lo stesso scopo con cui vengono utilizzati in linguaggio assembly. Permettono di manipolare i dati a livello di singolo bit e sono utilizati per eseguire velocemente divisioni e moltiplicazioni. Infatti, un compilatore C brillante utilizzera' automaticamente uno shift per una moltiplicazione come `x *= 2`.

Le API[4] di molti sistemi operativi (come *POSIX*[5] e Win32) contengono funzioni che lavorano su operandi codificati come bit. Per esempio, i sistemi POSIX mantengono i permessi sui file per tre diversi tipi di utenti: *utente*(un nome migliore sarebbe *proprietario*),*gruppo* e *altri*. Ad ogni tipo di utente puo' essere concesso il permesso di leggere, scrivere e/o eseguire un file. Un programmatore C che vuole cambiare questi permessi su di un file deve manipolare i singoli bit. POSIX definisce diverse macro per venire in aiuto (vedi la Tabella 3.6). La funzione `chmod` puo' essere usata per questo scopo. Accetta due parametri, una stringa con il nome del file su cui agire e un intero[6] con i bit appropriati impostati per il permesso desiderato. Per esempio, il codice sotto imposta i permessi in modo che il proprietario del file possa leggerlo e scrivere su di esso, gli utenti del gruppo possano leggere il file, mentre gli altri non possano avere nessun accesso al file.

```
chmod("foo", S_IRUSR | S_IWUSR | S_IRGRP );
```

La funzione POSIX `stat` puo' essere usata per trovare i bit di permesso correnti di un file. Usata con la funzione `chmod`, permette di modificare alcuni permessi senza toccare gli altri. Sotto e' riportato un'esempio che

[4]Application Programming Interface

[5]E'l'acronimo di Portable Operating System Interface for Computer Environments. Si tratta di uno standard basato su UNIX sviluppato da IEEE

[6]In realta' un parametro di tipo `mode_t` che e' un typedef di un tipo integrale.

rimuove i permessi di scrittura agli altri e aggiunge il permesso di lettura al proprietario del file. Gli altri permessi non sono alterati.

```
struct stat file_stats ;      /* struct usata da stat() */
stat("foo", & file_stats );   /* legge le info del file .
                                 file_stats .st_mode contiene i bit dei permessi */
chmod("foo", ( file_stats .st_mode & ~S_IWOTH) | S_IRUSR);
```

3.5 Rappresentazioni Big e Little Endian

Il capitolo 1 ha introdotto il concetto di rappresentazioni big e little endian per dati multibyte. L'autore si e' comunque accorto che questo concetto confonde molte persone. Questa sezione approfondisce ulteriormente l'argomento.

Il lettore si ricordera' che l'*endianess* si riferisce all'ordine con cui i singoli byte (*non* bit) di un dato multibyte sono memorizzati. Il metodo Big endian e' quello piu' diretto. Esso memorizza il byte piu' significativo come primo, quello successivo piu' significativo come secondo e cosi via. In altre parole, i bit *grandi* sono memorizzati per primi. Il metodo Little endian fa l'opposto (quelli meno significativi per primi). La famiglia dei processori x86 usa la rappresentazione little endian.

Come esempio, consideriamo il valore 12345678_{16} come double word. Nella rappresentazione big endian, i byte sarebbero memorizzati come 12 34 56 78. Nella rappresentazione in little endian, i byte sarebbero memorizzati come 78 56 34 12.

Il lettore molto probabilmente si chiedera' ora perche' un designer di chip sano di mente dovrebbe usare la rappresentazione little endian? Gli ingegneri della Intel erano cosi' sadici da infliggere questa rappresentazione cosi' confondente alle moltitudini di programmatori? In questo modo sembrerebbe che la CPU debba fare del lavoro in piu' per memorizzare i byte alla rovescia (e di nuovo rovesciare i byte quando questi sono letti dalla memoria). La risposta e' che la CPU non fa nessun lavoro in piu' per leggere e scrivere la memoria utilizzando la rappresentazione little endian. Occorre capire che la CPU e' composta di diversi circuiti elettronici che semplicemente lavorano sui valori dei bit. I bit (e i byte) non sono necessariamente in nessun ordine nella CPU.

Consideriamo il registro a 2 byte `AX`. Esso puo' essere scomposto in registri ad un byte:`AH` e `AL`. Ci sono circuiti nella CPU che mantegono il valore di `AH` e `AL`. Non c'e' nessun ordine tra questi circuiti nella CPU. Cioe', i circuiti per `AH` non sono ne' prima ne' dopo i circuiti per `AL`. Una istruzione `mov` che copia il valore di `AX` in memoria, copia il valore di `AL` e poi il valore

```
unsigned short word = 0x1234; /* si assume sizeof(short) == 2 */
unsigned char * p = (unsigned char *) &word;

if ( p[0] == 0x12 )
  printf ("Macchina Big Endian\n");
else
  printf ("Macchina Little Endian\n");
```

Figura 3.4: Come determinare l'*Endianness*

di `AH`. Questa operazione per la CPU non e' piu' lavoriosa che copiare prima `AH`.

Lo stesso discorso si applica ai singoli bit di un byte. Questi non sono assolutamente in nessun ordine nei circuiti della CPU (o della memoria). Dal momento che i singoli bit non possono essere allocati nella CPU o in memoria, non c'e' nessun modo per sapere (o preoccuparsi) in quale ordine questi sono tenuti internamente dalla CPU.

Il codice C in Figura 3.4 mostra come la *endianness* di una CPU puo' essere determinata. Il puntatore `p` tratta la variabile `word` come un array di due caratteri. Cosi', `p[0]` valuta il primo byte della `word` in memoria che dipende dalla *endianness* della CPU.

3.5.1 Quando preoccuparsi di Little e Big Endian

Per la programmazione tipica, la *endianness* della CPU non e' significativa. Il momento piu' tipico in cui invece diventa importante e' quando dati binari vengono trasferiti fra diversi sistemi di computer. Questo accade di solito utilizzando alcuni tipi di supporti fisici di massa (come un disco) oppure una rete. Dal momento che i dati in ASCII usano un solo byte, l'*endianness* per questi non e' un problema.

*con l'avvento di set di caratteri in multibyte, come UNICODE, l'*endianness *e' diventata importante anche per i dati di testo. UNICODE supporta entrambe le modalita' ed ha un meccanismo per specificare che tipo di endianness e' utilizzata per rappresentare i dati.*

Tutti gli header interni TCP/IP memorizzano gli interi nel formato big endian (chiamato *ordine dei byte della rete*). Le librerie TCP/IP forniscono delle funzioni in C per trattare il problema della *endianness* in modo portabile. Per esempio, la funzione `htonl()` converte una double word (o long integer) dal formato dell'*host* a quello della *rete*. La funzione `ntohl()` esegue la trasformazione opposta.[7] Per i sistemi big endian, le due funzioni semplicemente ritorno il loro parametro non modificato. Questo permette di scrivere programmi per la rete che possano essere compilati ed eseguiti su qualunque sistema a dispetto del suo formato di rappresentazione. Per

[7] In realta', l'inversione della *endianness* di un intero semplicemente inverte i byte; In questo modo, la conversione da big a little o da little a big sono la stessa cosa. In definitiva, entrambe le funzioni fanno la stessa cosa.

```
unsigned invert_endian( unsigned x )
{
  unsigned invert;
  const unsigned char * xp = (const unsigned char *) &x;
  unsigned char * ip = (unsigned char *) & invert;

  ip[0] = xp[3];    /* inverte i singoli byte */
  ip[1] = xp[2];
  ip[2] = xp[1];
  ip[3] = xp[0];

  return invert;    /* ritorna i byte invertiti */
}
```

Figura 3.5: Funzione invert_endian

maggiori informazioni, sulla *endianness* e sulla programmazione delle rete si veda l'eccellente libro di W. Richard Steven, *UNIX Network Programming*.

La Figura 3.5 mostra una funzione C che inverte l'*endianness* di una double word. Il processore 486 ha introdotto una nuova istruzione macchina chiamata `BSWAP` che inverte i byte si qualunque registro a 32 bit. Per esempio,

```
bswap   edx          ; inverte i byte di edx
```

L'istruzione non puo' essere usata sui registri a 16 bit. Al suo posto, puo' essere usata l'instruzione `XCHG` per invertire i byte dei registri a 16 bit che possono essere scomposti in registri a 8 bit. Per esempio:

```
xchg    ah,al        ; inverte i byte di ax
```

3.6 Conteggio di Bit

Precedentemente era stata fornita una tecnica diretta per il conteggio del numero di bit che sono "on" in una double word. Questa sezione si occupa di altri metodi meno diretti per fare cio' come un'esercizio utilizzando le operazioni sui bit discusse in questo capitolo.

3.6.1 Metodo Uno

Il primo metodo e' molto semplice, ma non ovvio. La Figura 3.6 mostra il codice.

```
int count_bits( unsigned int data )
{
  int cnt = 0;

  while( data != 0 ) {
    data = data & (data − 1);
    cnt++;
  }
  return cnt;
}
```

Figura 3.6: Conteggio di Bit: Metodo Uno

Come funziona? In ogni iterazione del ciclo, un bit in `data` e' messo a off. Quando tutti i bit sono off (*i.e.* quando `data` e' zero), il ciclo termina. Il numero di iterazioni che occorrono per rendere `data` zero e' uguale al numero di bit nel valore originale di `data`

La riga 6 e' dove i bit di `data` sono messi a off. Come funziona? Consideriamo la forma generale della rappresentazione binaria di `data` e l'1 piu' a destra in questa rappresentazione. Per definizione ogni bit dopo questo 1 deve essere 0. Ora, quale sara' la rappresentazione binaria di `data -1`? I bit a sinistra del bit piu' a destra saranno gli stessi come per `data`, ma dall'1 in poi i bit a destra saranno il complemento dei bit originali di `data`. Per esempio:

```
data       =  xxxxx10000
data - 1   =  xxxxx01111
```

dove le x sono le stesse per entrambi i numeri. Quando `data` e' messo in *AND* con `data -1`, il risultato azzerera' l'1 piu' a destra in `data` e lasciera' inalterati gli altri bit.

3.6.2 Metodo Due

Una tabella di controllo puo' anche essere usata per contare i bit di una double word arbitraria. L'approccio diretto sarebbe di precalcolare il numero di bit di ciascuna double word e memorizzare questo dato in un array. Con questo approccio, pero', ci sono due problemi. Ci sono circa *4 miliardi* di valori di double word! Questo significa che l'array sarebbe veramente grande e che la sua inizializzazione richiederebbe davvero molto tempo. (Infatti, a meno che non si vada ad usare realmente l'array per piu' di 4 miliardi di volte, occorrerebbe piu' tempo per l'inizializzazione dell'array che per il calcolo del conto dei bit utilizzando il metodo 1!)

Un metodo piu' realistico precalcolera' i conteggi dei bit per tutte i pos-

```
static unsigned char byte_bit_count [256];  /* tabella di controllo */

void  initialize_count_bits  ()
{
  int cnt, i, data;

  for( i = 0; i < 256; i++ ) {
    cnt = 0;
    data = i;
    while( data != 0 ) {          /* metodo uno */
      data = data & (data - 1);
      cnt++;
    }
    byte_bit_count [i] = cnt;
  }
}

int  count_bits ( unsigned int data )
{
  const unsigned char * byte = ( unsigned char *) & data;

  return  byte_bit_count [byte [0]]  +  byte_bit_count [byte [1]]  +
          byte_bit_count [byte [2]]  +  byte_bit_count [byte [3]];
}
```

Figura 3.7: Metodo Due

sibili valori di un byte e li memorizzera in un array. Poi la double word verra' divisa in 4 valori da un byte. I conteggi dei bit di questi quattro valori sara' cercato nell'array e sommati per trovare il conteggio dei bit del valore originale della double word. La Figura 3.7 mostra l'implementazione del codice per questo approcio.

La funzione `initialize_count_bits` deve essere chiamata prima della prima chiamata della funzione `count_bits`. Questa funzione inizializza l'array globale `byte_bit_count`. La funzione `count_bits` controlla la variabile `data` non come una double word ma come ad un array di 4 byte. Il puntatore `dword` agisce come un puntatore a questo array di 4 bytes. Cosi', `dword[0]` rappresenta uno dei byte in `data` (il byte meno significativo o quello piu' significativo dipendono dal fatto che l'hardware usi rispettivamente little endian o big endian.) Naturalmente si potrebbe usare un costrutto del tipo:

(data >> 24) & 0x000000FF

```
int count_bits(unsigned int x )
{
  static unsigned int mask[] = { 0x55555555,
                                 0x33333333,
                                 0x0F0F0F0F,
                                 0x00FF00FF,
                                 0x0000FFFF };
  int i;
  int shift;    /* numero di posizioni da shiftare a destra */

  for( i=0, shift=1; i < 5; i++, shift *= 2 )
    x = (x & mask[i]) + ( (x >> shift) & mask[i] );
  return x;
}
```

Figura 3.8: Metodo 3

per trovare il valore del byte piu' significativo, ed uno simile per gli altri byte; in ogni modo questo approccio e' piu' lento rispetto al riferimento ad un array.

Un ultimo punto, un ciclo `for` potrebbe essere usato facilmente per calcolare la somma alle righe 22 e 23. Questo pero', includerebbe un sovraccarico per l'inizializzazione dell'indice del ciclo, la comparazione dell'indice dopo ogni iterazione e l'incremento dell'indice. Calcolare la somma come una somma esplicita di 4 valori e' sicuramente piu' veloce. Infatti, un compilatore brillante convertira' la versione con il ciclo `for` in una somma esplicita. Questo processo di riduzione od eliminazione delle iterazioni di un ciclo e' una tecnica di ottimizzazione dei compilatori conosciuta come *loop unrolling*[8].

3.6.3 Metodo Tre

Questo e' un'altro metodo intelligente di contare i bit che sono a on nel dato. Questo metodo letteralemente somma gli uno e gli zero del dato insieme. Questa somma deve essere uguale al numero di 1 nel dato. Per esempio, consideriamo il conteggio degli uno nel byte memorizzato in una variabile di nome `data`. Il primo passo e' quello di eseguire questa operazione:

```
data = (data & 0x55) + ((data >> 1) & 0x55);
```

Come funziona? La costante hex `0x55` e' 01010101 in binario.Nel primo operando della addizione, `data` e' posto in AND con questo valore e sono estratti i bit nelle posizioni dispari. Il secondo operando `((data`

[8] letteralmente, *svolgimento del ciclo* (ndt)

`>> 1) & 0x55)` muove prima i bit nelle posizioni pari in posizioni dispari ed usa la stessa maschera per estrarre questi stessi bit. Ora, il primo operando contiene i bit dispari di `data` mentre il secondo quelli pari. Quando questi due operandi sono sommati, i bit pari e quelli dispari sono sommati insieme. Per esempio, se `data` fosse 10110011_2, allora:

`data &` 01010101_2					00	01	00	01
`+ (data >> 1) &` 01010101_2	or	+			01	01	00	01
					01	10	00	10

La somma sulla destra mostra i bit correntemente sommati insieme. I bit del byte sono divisi in 4 campi da 2 bit per mostrare che ci sono 4 diverse ed indipendenti addizioni da eseguire. Da momento che il valore massimo di queste somme puo' essere 2, non ci sono possibilita' che la somma crei un overflow e vada a corrompere una delle altre somme di campi.

Naturalmente, il numero totale di bit non e' ancora stato calcolato. Comunque, la tecnica che e' stata usata puo' essere estesa al calcolo totale in una serie di passi simili. Il passo successivo sara':

```
data = (data & 0x33) + ((data >> 2) & 0x33);
```

Continuando con l'esempio sopra (ricorda che `data` ora e' 01100010_2):

`data &` 00110011_2					0010	0010
`+ (data >> 2) &` 00110011_2	or	+			0001	0000
					0011	0010

Ora ci sono due campi di 4 bit che sono sommati fra loro

Il passo successivo e' quello di sommare queste due somme di bit per formare il risultato finale:

```
data = (data & 0x0F) + ((data >> 4) & 0x0F);
```

Utilizzando sempre l'esempio di prima (con `data` uguale a 00110010_2):

`data &` 00001111_2			00000010
`+ (data >> 4) &` 00001111_2	or	+	00000011
			00000101

Ora `data` e' 5, cioe' il risultato corretto. La Figura 3.8 mostra una implementazione di questo metodo che conta i bit in una double word. Esso usa un ciclo `for` per calcolare la somma. Sarebbe piu' veloce svolgendo il ciclo; comunque, il ciclo rende piu' chiaro come il metodo generalizza le diverse dimensioni del dato.

Capitolo 4

Sottoprogrammi

Questo capitolo si occupa dell'utilizzo di sottoprogrammi per creare programmi modulari e per interfacciarsi con linguaggi di alto livello (come il C). Le funzioni e le procedure nei linguaggi di alto livello sono esempi di sottoprogrammi.

Il codice che chiama il sotto programma e lo stesso sottoprogramma devono avere accordo su come verranno trasferiti i dati tra di loro. Queste regole sono chiamate *convenzioni di chiamata*. Una buona parte di questo capitolo trattera' le convenzioni di chiamata standard del C che potranno essere usate per interfacciare sottoprogrammi in assembly con i programmi scritti in C. Questa (ed altre convenzioni) spesso passano degli indirizzi (*i.e.* puntatori) che permettono ai sottoprogrammi di avere accesso ai dati in memoria.

4.1 Indirizzamento Indiretto

L'indirizzamento indiretto permette ai registri di comportarsi come variabili puntatore. Per indicare che un registro viene usato indirettamente come un puntatore, viene racchiuso tra parentesi quadre (`[]`). Per esempio:

```
1  mov    ax, [Data]     ; indirizzamento diretto normale alla memoria
2
3  mov    ebx, Data      ; ebx = & Data
4  mov    ax, [ebx]      ; ax = *ebx
```

Poiche' la dimensione di AX e' una word, la riga 3 legge una word a partire dall'indirizzo memorizzato in EBX. Se AX e' sostituito con AL, verrebbe letto solo un singolo byte. E' importante capire che i registri non hanno tipi come succede per le variabili in C. Cosa punti EBX e' determinato esclusivamente da quale istruzione lo usa. Inoltre, anche il fatto che EBX sia un puntatore e' determinato dalla istruzione. Se EBX non fosse usato

correttamente, spesso non verrebbe segnalato nessun errore dall'assembler. Pero', il programmma non funzionerebbe correttamente. Questa e' una delle molte ragioni per cui la programmazione in assembly e' piu' portata all'errore rispetto alla programmazione nei linguaggi di alto livello.

Tutti i registri generali a 32 bit (EAX, EBX, ECX, EDX) e i registri indice (ESI, EDI) possono essere usati per l'indirizzamento indiretto. In generale i registri a 16 e 8 bit invece non possono essere usati.

4.2 Semplice esempio di sottoprogramma

Un sottoprogramma e' una unita' indipendente di codice che puo' essere usata da parti diverse del programma. In altre parole, un sottoprogramma equivale ad una funzione in C. Il sottoprogramma puo' essere invocato con una istuzione di salto, ma il suo ritorno puo' presentare un problema. Se il sotto programma viene usato da diverse parti del programma, deve necessariamente tornare alla sezione del codice che lo ha invocato. Cosi' il salto indietro dal sottoprogramma non puo' essere collegato ad una etichetta. Il codice sotto mostra come cio' viene fatto utilizzando una forma indiretta della istruzione `JMP`. Questa forma usa il valore di un registro per determinare dove deve saltare (in questo modo il registro si comporta come un *puntatore a funzione* in C.) Ecco il primo programma del capitolo 1 riscritto per usare un sottoprogramma.

sub1.asm

```
1  ; file: sub1.asm
2  ; Programma di esempio
3  %include "asm_io.inc"
4
5  segment .data
6  prompt1 db    "Enter a number: ", 0       ; non dimenticate il terminatore null
7  prompt2 db    "Enter another number: ", 0
8  outmsg1 db    "You entered ", 0
9  outmsg2 db    " and ", 0
10 outmsg3 db    ", the sum of these is ", 0
11
12 segment .bss
13 input1  resd 1
14 input2  resd 1
15
16 segment .text
17         global  _asm_main
18 _asm_main:
19         enter   0,0               ; setup routine
```

```
20          pusha
21
22          mov     eax, prompt1      ; stampa il prompt
23          call    print_string
24
25          mov     ebx, input1       ; memorizza l'indirizzo di input1 in ebx
26          mov     ecx, ret1         ; memorizza l'indirizzo di ritorno in ecx
27          jmp     short get_int     ; legge l'intero
28  ret1:
29          mov     eax, prompt2      ; stampa il prompt
30          call    print_string
31
32          mov     ebx, input2
33          mov     ecx, \$+ 7        ; ecx = questo indirizzo + 7
34          jmp     short get_int
35
36          mov     eax, [input1]     ; eax = dword in input1
37          add     eax, [input2]     ; eax += dword in input2
38          mov     ebx, eax          ; ebx = eax
39
40          mov     eax, outmsg1
41          call    print_string      ; stampa il primo messaggio
42          mov     eax, [input1]
43          call    print_int         ; stampa input1
44          mov     eax, outmsg2
45          call    print_string      ; stampa il secondo messaggio
46          mov     eax, [input2]
47          call    print_int         ; stampa input2
48          mov     eax, outmsg3
49          call    print_string      ; stampa terzo messaggio
50          mov     eax, ebx
51          call    print_int         ; stampa la somma (ebx)
52          call    print_nl          ; stampa una nuova linea (a capo)
53
54          popa
55          mov     eax, 0            ; ritorna al C
56          leave
57          ret
58  ; sottoprogramma get_int
59  ; Parametri:
60  ;   ebx - indirizzo di una dword in cui salvo l'intero
61  ;   ecx - indirizzo dell'istruzione a cui ritornare
```

```
62 ; Notes:
63 ;   valore di eax e' cancellato
64 get_int:
65         call    read_int
66         mov     [ebx], eax          ; memorizza input in memoria
67         jmp     ecx                 ; torna al chiamante
```

sub1.asm

Il sottoprogramma `get_int` usa una semplice convenzione basata sui registri. Questa si aspetta che il registro EBX contenga l'indirizzo della DWORD in cui memorizzare il numero inserito e il registro ECX l'indirizzo del istruzione a cui ritornare. Dalla riga 25 alla 28, l'etichetta `ret1` e' utilizzata per calcolare questo indirizzo di ritorno. Dalla riga 32 alla 34, l'operatore `$` e' utilizzato per calcolare l'indirizzo di ritorno. L'operatore `$` ritorna l'indirizzo corrente della riga in cui appare. L'espressione `$ + 7` calcola l'indirizzo della istruzione `MOV` alla riga 36.

Entrambe queste operazioni di calcolo dell'indirizzo di ritorno sono scomode. Il primo metodo richiede che venga definita una etichetta per ogni chiamata ad un sottoprogramma. Il secondo metodo non richiede una etichetta, ma richiede una particolare attenzione. Se fosse stato utilizzato un salto di tipo near invece di uno di tipo short, il numero da aggiungere a `$` non sarebbe stato 7! Fortunatamente, c'e' un modo molto piu' semplice di chiamare i sottoprogrammi. Questo metodo usa lo *stack*[1].

4.3 Lo Stack

Molte CPU hanno un supporto integrato per lo stack. Uno stack e' una pila di tipo *LIFO*(Last-In First-Out, l'ultimo che e' entra e' il primo che esce). Lo stack e' un'area di memoria organizzata in questo modo. L'istruzione `PUSH` aggiunge i dati allo stack e l'istruzione `POP` li rimuove. I dati rimossi sono sempre gli ultimi dati aggiunti (da qui deriva il tipo LIFO dello stack).

Il registro di segmento SS specifica il segmento che contiene lo stack (generalmente e' lo stesso segmento in cui sono memorizzati i dati). Il registro ESP contiene l'indirizzo del dato che sara' rimosso dallo stack. Questo dato viene considerato in *cima* allo stack. I dati possono essere aggiunti solo in unita' di double word. Non e' possibile, quindi, aggiungere allo stack un singolo byte.

L'istruzione `PUSH` inserisce una double word[2] nello stack, prima sottraendo 4 da ESP e poi memorizzando la double word in `[ESP]`. L'istruzione `POP`

[1] letteralmente *pila*(ndt)

[2] In realta' anche le word possono essere inserite, ma in modalita' protetta a 32 bit e' meglio utilizzare double word per lo stack

legge una double word in `[ESP]` e poi somma a ESP, 4. Il codice sotto mostra come queste istruzioni funzionano e assume che ESP sia inizialmente `1000H`.

```
1       push   dword 1    ; 1 memorizzato in 0FFCh, ESP = 0FFCh
2       push   dword 2    ; 2 memorizzato in 0FF8h, ESP = 0FF8h
3       push   dword 3    ; 3 memorizzato in 0FF4h, ESP = 0FF4h
4       pop    eax        ; EAX = 3, ESP = 0FF8h
5       pop    ebx        ; EBX = 2, ESP = 0FFCh
6       pop    ecx        ; ECX = 1, ESP = 1000h
```

Lo stack puo' essere utilizzato come luogo conveniente dove memorizzare temporaneamente i dati. Viene anche utilizzato per gestire le chiamate a sottoprogrammi, passandovi parametri e variabili locali.

L'80x86 inoltre fornisce l'istruzione `PUSHA` che mette i valori dei registri EAX, EBX, ECX, EDX, ESI, EDI e EBP (non in questo ordine) nello stack. L'istruzione `POPA` viene invece utilizzata per recuperare questi registri dallo stack.

4.4 Le istruzioni CALL e RET

L'80x86 fornisce due istruzioni che usano lo stack per rendere le chiamate a sottoprogrammi facili e veloci. L'istruzione CALL effettua un salto incondizionato ad un sottoprogramma e *mette* l'indirizzo della istruzione successiva nello stack. L'istruzione RET *estrae* un'indirizzo e salta quello. Quando si usano queste istruzioni, e' necesario che lo stack sia usato correttamente, in modo che venga estratto il numero corretto dall'istruzione RET!

Il programma precedente puo' essere riscritto per utilizzare queste nuove istruzioni sostituendo le righe dalla 25 alla 34 cosi':

```
      mov    ebx, input1
      call   get_int

      mov    ebx, input2
      call   get_int
```

e cambiando il sottoprogramma `get_int` in:

```
get_int:
      call   read_int
      mov    [ebx], eax
      ret
```

Ci sono molti vantaggi nell'uso di CALL e RET:

- E' piu' semplice!
- Permette di eseguire chiamate annidate ai sottoprogrammi piu' facilmente. Nota che `get_int` chiama `read_int`. Questa chiamata mette un'altro indirizzo nello stack. Al termine del codice di `read_int` e' una RET che estrae l'indirizzo di ritorno e salta al codice di `get_int`. Quando la RET di `get_int` e' eseguita, questa estrae l'indirizzo di ritorno che porta nuovamente a `asm_main`. Tutto cio' funziona correttamente grazie alla proprieta' LIFO dello stack.

Ricorda che e' *molto* importante estrarre tutti i dati che sono stati inseriti nello stack. Per esempio, considera il codice seguente:

```
1 get_int:
2       call   read_int
3       mov    [ebx], eax
4       push   eax
5       ret                  ; estrae il valore EAX, non l'indirizzo di ritorno!!
```

Questo codice non ritorna correttamente al chiamante!

4.5 Convenzioni di chiamata

Quando viene invocato un sottoprogramma, il codice chiamante ed il sottoprogramma *il chiamato* devono avere accordo su come vengono passati i dati fra loro. I linguaggi di alto livello hanno dei modi standard per passare i dati conosciuti come *convenzioni di chiamata*. Per interfacciare il codice di alto livello con il linguaggio assembly, il codice assembly deve usare la stessa convenzione del linguaggio di alto livello. Le convenzioni di chiamata possono differire da compilatore a compilatore o possono variare dipendentemente dal modo in cui il codice viene compilato (*i.e.* se le ottimizzazioni sono attivate o meno). Una convenzione universale e' che il codice sara' invocato con una istruzione `CALL` e ritonera' indietro con una `RET`.

Tutti i compilatori C per PC supportano una convenzione di chiamata che sara' descritta a fasi successive nel resto di questo capitolo. Queste convenzioni permettono di creare sottoprogrammi che sono *rientranti*[3]. Uno sottoprogramma rientrante puo' essere chiamato tranquillamente in qualunque punto di un programma (anche dentro lo stesso sottoprogramma).

[3]Un sottoprogramma e' *rientrante* quando puo' essere chiamato diverse volte in parallelo, senza interferenze fra le chiamate(ndt).

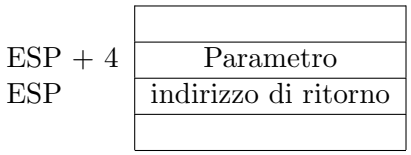


Figura 4.1:

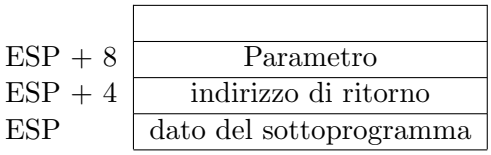


Figura 4.2:

4.5.1 Passaggio di parametri sullo stack

I parametri possono essere passati ad un sottoprogramma attraverso lo stack. Sono messi sullo stack prima della istruzione `CALL`. Come in C, se un parametro viene modificato dal sottoprogramma, deve essere passato *l'indirizzo* del dato, non il suo *valore*. Se la dimensione del parametro e' minore di una double word, deve essere convertito in un double word prima di essere messo nello stack.

I parametri nello stack non sono estratti dal sottoprogramma, ma sono invece acceduti dallo stesso stack. Perche'?

- Dal momento che sono messi nello stack prima dell'istruzione `CALL` l'indirizzo di ritorno dovrebbe essere estratto per primo (e di nuovo messo dentro)

- Spesso i parametri saranno usati in diversi posti nel sottoprogramma. Generalmente, non sono tenuti in un registro per l'intero sottoprogramma e dovrebbero essere salvati in memoria. Lasciarli nello stack permette di avere uno loro copia in memoria che puo' essere acceduta in qualunque punto del sottoprogramma.

Quando si usa l'indirizzamento indiretto, il processore 80x86 accede a diversi segmenti in virtu' di quali registri sono utilizzati nell'espressine di indirizzamento indiretto. ESP (e EBP) usano il segmento dello stack, mentre EAX, EBX, ECX ed EDX usano il segmento dei dati. Comunque, generalemente cio' non e' importante per la maggior parte dei programmi in modalita' protetta, in quanto per loro i due segmenti sono lo stesso.

Considera un sottoprogramma a cui e' passato un solo parametro sullo stack. Quando il sottoprogramma viene invocato, lo stack appare come quello in Figura 4.1. Il parametro puo' essere acceduto utilizzando l'indirizzamento indiretto (`[ESP+4]`[4]).

Se lo stack viene utilizzato all'interno del sottoprogramma per memorizzare i dati, il numero da aggiungere a ESP cambiera'. Per esempio, la

[4]E' lecito aggiungere una costante ad un registro quando si usa l'indirizzamento indiretto. Sono possibili anche espressioni piu' complicate. Questo aspetto sara' discusso nel prossimo capitolo

```
1  subprogram_label:
2        push   ebp           ; salva il valore originale di EBP
3                             ; nello stack
4        mov    ebp, esp      ; nuovo EBP = ESP
5  ; subprogram code
6        pop    ebp           ; ripristina il valore originale di EBP
7        ret
```

Figura 4.3: Forma generale di un sottoprogramma

ESP + 8	EBP + 8	Parametro
ESP + 4	EBP + 4	Indirizzo di ritorno
ESP	EBP	EBP salvato

Figura 4.4:

Figura 4.2 mostra come appare lo stack se una DWORD viene aggiunta allo stack. Ora il parametro si trova a `ESP + 8` non piu' a `ESP + 4`. Cosi', puo' essere molto facile sbagliare nell'utilizzo di ESP nella referenziazione dei parametri. Per risolvere il problema, l'80386 fornisce un'altro registro per l'uso: EBP. Il solo scopo di questo registro e' di referenziare i dati nello stack. La convenzione di chiamata del C stabilisce che un sottoprogramma prima deve salvare il valore di EBP nello stack e poi pone EBP uguale a ESP. Questo permette ad ESP di cambiare in funzione dei dato che viene messo o estratto dallo stack, senza modificare EBP. Alla fine del sottoprogramma , deve essere rispristinato il valore originale di EBP (percio' viene salvato all' inizio del sottoprogramma.) La Figura 4.3 mostra una forma generale di sottoprogramma che segue queste convenzioni.

Le righe 2 e 3 in Figura 4.3 cotruiscono il *prologo* generale del programma. Le righe 5 e 6 creano l'*epilogo*. La Figura 4.4 mostra come lo stack appare immediatamente dopo il prologo. Ora il parametro puo' essere acceduto con `[EBP + 8]` in ogni punto del sottoprogramma senza doversi preoccupare di cosa altro e' stato messo nello stack dal sottoprogramma.

Dopo che il sottoprogramma e' terminato, i parametri che erano stati messi nello stack devono essere rimossi. La convenzione di chiamata del C specifica che il codice chiamante deve fare cio'. Altre convenzioni sono diverse. Per esempio, la convenzione di chiamata del Pascal specifica che e' il sottoprogramma che deve rimuovere questi parametri. (Esiste un' altra forma dell'istruzione RET che rende facile questa operazione.) Alcuni compilatori C supportano entrambe queste convenzioni. La parola chiave

```
1       push   dword 1        ; passa 1 come parametro
2       call   fun
3       add    esp, 4         ; rimuove il parametro dallo stack
```

Figura 4.5: Esempio di chiamata a sottoprogramma

`pascal` e' utilizzata nel prototipo e nella definizione di una funzione per indicare al compilatore di usare questa convenzione. Infatti, la convezione `stdcall` che le funzioni in C delle API di MS Windows utilizzano, funziona in questo modo. Qual'e' il vantaggio? E' leggermente piu' efficente rispetto alla convenzione del C. Perche' allora tutte le funzioni C non usano questa convenzione? In generale, il C permette ad una funzione di avere un numero variabile di argomenti (*Es.* le funzioni `printf` e `scanf`). Per questo tipo di funzioni, l'operazione di rimozione dei parametri dallo stack puo' variare da una chiamata all'altra. La convenzione del C permette alle istruzioni che eseguono queste operazioni di essere modificate facilmente fra una chiamata e l'altra. Le convenzioni pascal e stdcall rendono queste operazioni molto difficile. Percio' la convenzione Pascal (come il linguaggio pascal) non permettono questo tipo di funzioni. MS Windows puo' usare questa convenzione dal momento che nessuna delle funzione delle sue API ha un numero variabile di argomenti.

La Figura 4.5 mostra come un sottoprogramma puo' essere chiamato utilizzando la convenzione del C. La riga 3 rimuove il parametro dallo stack manipolando direttamente il puntatore allo stack. Potrebbe anche essere usata una istruzione `POP` per fare cio', ma richiederebbe di memorizzazione inutilmente in un registro. In realta', per questo particolare caso, molti compilatori utilizzerebbero una istruzione `POP ECX` per rimuovere il parametro. Il compilatore userebbe una `POP` invece di una `ADD` poiche' `ADD` richiede piu' byte per l'istruzione. Comunque, anche `POP` cambia il valore di ECX! Di seguito un'altro programma di esempio con due sottoprogrammi che usano le convenzioni di chiamata del C discusse prima. La riga 54 (ed altre righe) mostra come i segmenti multipli di dati e di testo possano essere dichiaranti in un singolo file sorgente. Questi saranno combinati in segmenti singoli di dati e testo nel processo di collegamento. Dividere i dati ed il codice in segmenti separati permette che i dati utilizzati da un sottoprogramma sia definiti vicino al codice del sottoprogramma stesso.

sub3.asm

```
1  %include "asm_io.inc"
2
3  segment .data
```

```
4  sum     dd   0
5
6  segment .bss
7  input   resd 1
8
9  ;
10 ; algoritmo in pseudo-codice
11 ; i = 1;
12 ; sum = 0;
13 ; while( get_int(i, &input), input != 0 ) {
14 ;   sum += input;
15 ;   i++;
16 ; }
17 ; print_sum(num);
18 segment .text
19         global  _asm_main
20 _asm_main:
21         enter   0,0               ; setup routine
22         pusha
23
24         mov     edx, 1            ; edx e' 'i' nello pseudo-codice
25 while_loop:
26         push    edx               ; salva i nello stack
27         push    dword input       ; mette l'indirizzo di input nello stack
28         call    get_int
29         add     esp, 8            ; rimuove i e &input dallo stack
30
31         mov     eax, [input]
32         cmp     eax, 0
33         je      end_while
34
35         add     [sum], eax        ; sum += input
36
37         inc     edx
38         jmp     short while_loop
39
40 end_while:
41         push    dword [sum]       ; mette il valore della somma nello stack
42         call    print_sum
43         pop     ecx               ; rimuove [sum] dallo stack
44
45         popa
```

```
46          leave
47          ret
48
49  ; sottoprogramma get_int
50  ; Parametri (nell'ordine di inserimento nello stack)
51  ;   numero di input (in [ebp + 12])
52  ;   l'indirizzo della word in cui memorizzare (at [ebp + 8])
53  ; Notes:
54  ;   i valori di eax ed ebx sono distrutti
55  segment .data
56  prompt  db      ") Enter an integer number (0 to quit): ", 0
57
58  segment .text
59  get_int:
60          push    ebp
61          mov     ebp, esp
62
63          mov     eax, [ebp + 12]
64          call    print_int
65
66          mov     eax, prompt
67          call    print_string
68
69          call    read_int
70          mov     ebx, [ebp + 8]
71          mov     [ebx], eax          ; memorizza input in memoria
72
73          pop     ebp
74          ret                         ; ritorna al chiamate
75
76  ; sottoprogramma print_sum
77  ; stampa la somma
78  ; Parametro:
79  ;   somma da stampare (in [ebp+8])
80  ; Note: distrugge il valore di eax
81  ;
82  segment .data
83  result  db      "The sum is ", 0
84
85  segment .text
86  print_sum:
87          push    ebp
```

```
subprogram_label:
      push   ebp            ; salva il valore originale di EBP

      mov    ebp, esp       ; nuovo EBP = ESP
      sub    esp, LOCAL_BYTES   ; = # bytes necessari per le

; subprogram code
      mov    esp, ebp       ; deallocata le variabili
      pop    ebp            ; ripristina il valore originale di EBP
      ret
```

Figura 4.6: Forma generale di sottoprogramma con le variabili locali

```
        mov     ebp, esp

        mov     eax, result
        call    print_string

        mov     eax, [ebp+8]
        call    print_int
        call    print_nl

        pop     ebp
        ret
```

sub3.asm

4.5.2 Variabili locali nello stack

Lo stack puo' essere utilizzato anche come luogo conveniente per le variabili locali. E' esattamente il luogo dove il C memorizza le normali (o *automatic* in C) variabili. Utilizzare lo stack per le variabili e' importante se si vogliono creare sottoprogrammi rientranti. Un sottoprogramma rientrante puo' essere invocato da qualunque luogo, anche dal sottoprogramma stesso. In altre parole, i sottoprogrammi rientranti possono essere chiamati *ricorsivamente*. Utilizzare lo stack per le variabili inoltre risparmia memoria. I dati non memorizzati nello stack utilizzano memoria dall'inizio alla fine del programma (il C chiama queste variabili *globali* o *statiche*). I dati memorizzati nello stack utilizzano memoria solo quando e' attivo il sottoprogramma in cui sono definite.

Le variabili locali sono memorizzate nello stack subito dopo il valore memorizzato di EBP. Sono allocate sottraendo il numero di byte richiesti da ESP nel prologo del sottoprogramma. La Figura 4.6 mostra la nuova

```
void calc_sum( int n, int * sump )
{
  int i, sum = 0;

  for( i=1; i <= n; i++ )
    sum += i;
  *sump = sum;
}
```

Figura 4.7: Versione in C della somma

struttura del sottoprogramma. Il registro EBP e' utilizzato per accedere alle variabili locali. Considera la funzione C in Figura 4.7. La Figura 4.8 mostra come puo' essere scritto l'equivalente sottoprogramma in assembly.

La Figura 4.9 mostra come appare lo stack dopo il prologo del programma in Figura 4.8. Questa sezione dello stack che contiene i parametri, le informazioni di ritorno e le variabili locali e' chiamata *stack frame*. Ogni invocazione di una funzione C crea un nuovo stack frame nello stack.

Il prologo e l'epilogo di un sottoprogramma possono essere semplificati utilizzando due istruzioni speciali che sono state create specificatamente per questo scopo. L'istruzione `ENTER` esegue il codice del prologo e l'istruzione `LEAVE` esegue l'epilogo. L'istruzione `ENTER` accetta due operandi immediati. Per la compatibilita' con la convenzione del C, il secondo operando e' sempre 0. Il primo operando e' il numero di byte necessari dalle variabili locali. L'istruzione `LEAVE` invece non ha operandi. La Figura 4.10 mostra come vengono usate queste istruzioni. Nota che la struttura del programma (Figura 1.7) utilizza anche `ENTER` e `LEAVE`.

Nonostante `ENTER` *e* `LEAVE` *semplifichino il prologo e l'epilogo, non sono usati molto spesso. Perche'? Poiche' sono piu' lenti delle equivalenti istruzioni piu' semplici! Questo e' un'esempio di quando non sia possibile assumere che una sequenza di istruzioni e' piu' veloce di una istruzione multipla.*

4.6 Programmi Multi Modulo

Un *programma multi modulo* e' composto da piu' di un file oggetto. Tutti i programmmi presentati fin qui sono programmi multi modulo. Consistono di un file oggetto guida in C e il file oggetto in assembly (piu' i file oggetto della libreria C). Ricorda che il linker combina i file oggetto in un singolo programma eseguibile. Il linker deve verificare il collegamento fra i riferimenti fatti a ciascuna etichetta in un modulo (*i.e.* file oggetto) e la loro definizione nell'altro modulo. Per far si' che il modulo A possa usare una etichetta definita nel modulo B, deve essere usata la direttiva `extern`. Dopo la direttiva `extern` vengono elencate una serie di etichette delimitate da virgole. La direttiva dice all'assembler di considerare queste etichette come *esterne* al modulo. In altre parole sono etichette che possono essere usate

```
1  cal_sum:
2        push   ebp
3        mov    ebp, esp
4        sub    esp, 4               ; crea spazio per la somma locale
5
6        mov    dword [ebp - 4], 0   ; sum = 0
7        mov    ebx, 1               ; ebx (i) = 1
8  for_loop:
9        cmp    ebx, [ebp+8]         ; e' i <= n?
10       jnle   end_for
11
12       add    [ebp-4], ebx         ; sum += i
13       inc    ebx
14       jmp    short for_loop
15
16 end_for:
17       mov    ebx, [ebp+12]        ; ebx = sump
18       mov    eax, [ebp-4]         ; eax = sum
19       mov    [ebx], eax           ; *sump = sum;
20
21       mov    esp, ebp
22       pop    ebp
23       ret
```

Figura 4.8: Versione in assembly della somma

in questo modulo ma sono definite nell'altro. Il file `asm_io.inc` definisce le routine `read_int`, *etc.* come esterne.

In assembly, le etichette di default non possono essere accedute dall'esterno. Se una etichetta puo' essere acceduta da altri moduli oltre quello in cui e' definita, deve essere dichiarata `global` nel proprio modulo. La direttiva `global` fa proprio questo. La riga 13 del programma scheletro riportato in Figura 1.7 mostra come le etichette di `_asm_main` sono definite globali. Senza questa dichiarazione, ci sarebbe stato un'errore di collegamento. Perche'? Perche' il codice C non sarebbe stato capace di riferire alla etichetta *interna* di `_asm_main`.

Di seguito e' riportato il codice per l'esempio precedente, riscritto utilizzando i moduli. I due sottoprogrammi (`get_int` e `print_sum`) si trovano in un file sorgente diverso da quello di `_asm_main`

main4.asm

ESP + 16	EBP + 12	sump
ESP + 12	EBP + 8	n
ESP + 8	EBP + 4	indirizzo di ritorno
ESP + 4	EBP	EBP salvato
ESP	EBP - 4	sum

Figura 4.9:

```
subprogram_label:
      enter  LOCAL_BYTES, 0  ; = # byte necessari per variabili
                             ; locali
; subprogram code
      leave
      ret
```

Figura 4.10: Forma generale di sottoprogramma con le variabili locali utilizzando `ENTER` e `LEAVE`

```
%include "asm_io.inc"

segment .data
sum     dd   0

segment .bss
input   resd 1

segment .text
        global  _asm_main
        extern  get_int, print_sum
_asm_main:
        enter   0,0               ; setup routine
        pusha

        mov     edx, 1            ; edx e' 'i' in pseudo-code
while_loop:
        push    edx               ; salva i nello stack
        push    dword input       ; mette l'indirizzo di input nello stack
        call    get_int
        add     esp, 8            ; remuone i e &input dallo stack

        mov     eax, [input]
```

```
24          cmp     eax, 0
25          je      end_while
26
27          add     [sum], eax          ; sum += input
28
29          inc     edx
30          jmp     short while_loop
31
32  end_while:
33          push    dword [sum]         ; mette il valore della somma nello stack
34          call    print_sum
35          pop     ecx                 ; remuove [sum] dallo stack
36
37          popa
38          leave
39          ret
```

main4.asm

sub4.asm

```
1   %include "asm_io.inc"
2
3   segment .data
4   prompt  db      ") Enter an integer number (0 to quit): ", 0
5
6   segment .text
7           global  get_int, print_sum
8   get_int:
9           enter   0,0
10
11          mov     eax, [ebp + 12]
12          call    print_int
13
14          mov     eax, prompt
15          call    print_string
16
17          call    read_int
18          mov     ebx, [ebp + 8]
19          mov     [ebx], eax          ; memorizza input in memoria
20
21          leave
22          ret                         ; ritorna al chiamante
23
24  segment .data
25  result  db      "The sum is ", 0
```

```
26
27 segment .text
28 print_sum:
29         enter   0,0
30
31         mov     eax, result
32         call    print_string
33
34         mov     eax, [ebp+8]
35         call    print_int
36         call    print_nl
37
38         leave
39         ret
```
sub4.asm

L'esempio precedente ha solo etichette di codice globlali; Comunque le etichette di dati globali funzionano esattamente nello stesso modo.

4.7 Interfacciare l'Assembly con il C

Oggi, pochissimi programmi sono scritti completamente in assembly. I compilatori hanno raggiunto una grande efficenza nel convertire il codice di alto livello in codice macchina efficente. Dal momento che e' molto piu' facile scrivere codice utilizzando un linguaggio di alto livello, questi sono diventati molto popolari. In aggiunta, il codice di alto livello e' *molto* piu'portabile rispetto a quello in assembly!

Quando viene utilizzato, si tratta di piccole parti di codice. Cio' puo' essere fatto in due modi: attraverso la chiamata a sottoroutine in assembly o con assembly inline. L'assembly inline permette al programmatore posizionare comandi assembly direttamente nel codice C. Questo puo' essere molto conveniente; d'altra parte, ci sono degli svantaggi. Il codice assembly deve essere scritto nel formato che usa il compilatore. Nessun compilatore al momento supporta il formato del NASM. Compilatori differenti richiedono formati diversi. Borland e Microsoft richiedono il formato MASM. DJGPP e il gcc di Linux richiedono il formato GAS[5]. La tecnica di chiamata di una routine assembly e' molto piu' standardizzata sui PC.

Le routine Assembly sono di solito usate con il C per i seguenti motivi:

- E' richiesto l'accesso diretto a caratteristiche hardware del computer a cui e' difficile se non impossibile accedere con il C.

[5]GAS e' l'assembler che viene usato da tutti i compilatori GNU. Utilizza la sintassi AT&T che e' molto diversa da quelle relativamente simili di MASM,TASM e NASM.

```
1  segment .data
2  x            dd     0
3  format       db     "x = %d\n", 0
4
5  segment .text
6  ...
7        push   dword [x]     ; mette il valore di x
8        push   dword format  ; mette l'indirizzo della stringa format
9        call   _printf       ; nota l'underscore!
10       add    esp, 8        ; rimuove i parametri dallo stack
```

Figura 4.11: Chiamata a `printf`

EBP + 12	valore di `x`
EBP + 8	indirizzo della stringa format
EBP + 4	indirizzo di ritorno
EBP	EBP salvato

Figura 4.12: Stack dentro `printf`

- La routine deve essere il piu' veloce possibile ed il programmatore puo' ottimizzare manualmente meglio di cio' che puo' il compilatore.

Il secondo motivo non e' piu' valido come lo era un tempo. La tecnologia dei compilatore e' migliorata nel corso degli anni e oggi i compilatori possono spesso generare codice molto efficente (specialmente se le ottimizzazioni del compilatore sono attivate). Gli svantaggi delle routine in assembly sono: ridotta portabilita' e leggibilita'.

Molte delle convenzioni di chiamata del C sono gia' state analizzate. Ci sono pero' alcune caratteristiche aggiuntive che devono essere descritte.

4.7.1 Il Salvataggio dei Registri

La parola chiave `register` *puo' essere usata nella dichiarazione di una variabile C per specificare al compilatore che per questa variabile venga usato un registro invece di una locazione di memoria. Queste sono conosciute come variabili registro. I Compilatori moderni fanno tutto cio' automaticamente senza la necessita di alcuna specificazione.*

Prima di tutto, il C assume che una sottoroutine mantenga i valori dei seguenti registri: EBX, ESI, EDI, EBP, CS, DS, SS ed ES. Questo non vuol dire che la sottoroutine non possa modificarli internamente. Significa invece che se modifica il loro valore, deve poi ripristinare quello originale prima di ritornare al codice chiamante. I valori di EBX, ESI ed EDI non devono essere modificati perche' il C li utilizza per le *variabili registro*. Generalmente lo stack e' utilizzato per salvare il valore originale di questi registri.

4.7.2 Etichette di Funzioni

Molti compilatori C appongo un singolo carattere underscore(_) all'inizio del nome delle funzioni e delle variabili globali/statiche. Per esempio, ad una funzione di nome `f` sara' assegnata l'etichetta `_f`. Cosi', se si tratta di una sottoroutine in assembly, *deve* essere etichettata `_f`, non `f`. Il compilatore gcc Linux *non* prepende nessun carattere. Negli eseguibili ELF di Linux, si usa semplicemente l'etichetta `f` per la funzione C `f`. Il gcc di DJGPP invece prepende un underscore. Nota che nello scheletro del programma(Figure 1.7), l'etichetta della routine principale e' `_asm_main`.

4.7.3 Passaggio di Parametri

In base alla convenzione di chiamata del C, gli argomenti di una funzione sono messi nello stack nell'ordine *inverso* rispetto a quello con cui appaiono nella chiamata di funzione.

Considera il seguente comando C: `printf("x = %d\n",x);` La Figura 4.11 mostra come verrebbe compilato (nell' equivalente formato NASM). La Figura 4.12 mostra come appare lo stack dopo il prologo dentro la funzione `printf`. La funzione `printf` e' una delle funzioni della libreria C che puo' accettare un numero variabile di argomenti. Le regole delle convenzioni di chiamata del C sono scritte specificatamente per permettere questo tipo di funzioni. Dal momento che l'indirizzo della stringa format e' inserito per ultimo, la sua locazione nello stack sara' *sempre* a `EBP + 8` indipendentemente da quanti parametri vengono passati alla funzione. Il codice di `printf` puo' quindi controllare la stringa format per determinare quanti parametri sono stati passati e cercarli nello stack.

Non e' necessario usare l'assembly per processare un numero arbitrario di argomenti in C. Il file di header `stdarg.h` *definisce delle macro che possono essere usate per processarli in modo portabile. Vedi un buon libro sul C per maggiori dettagli.*

Naturalmente, se c'e' un errore, `printf("x = %d\n")`, il codice di `printf` stampera' ancora la double word che si trova a `[EBP + 12]`. Che pero' non sara' il valore di `x`!.

4.7.4 Calcolo degli indirizzi delle variabili locali

Trovare l'indirizzo di una etichetta definita nei segmenti `data` o `bss` e' semplice. Fondamentalmente lo fa il linker. Calcolare invece l'indirizzo di una variabile locale (o un parametro) nello stack non e' cosi diretto. E' comunque una necessita' molto comune nella chiamata a sottoroutine. Consideriamo il caso del passaggio dell' indirizzo di una variabile (chiamiamolo `x`) ad una funzione (chiamiamola `foo`). Se `x` fosse locato a EPB − 8 sullo stack, non sarebbe possibile usare semplicemente:

```
mov    eax, ebp - 8
```

Perche'? Il valore che `MOV` memorizza in EAX deve essere calcolato dall'assembler (cioe', alla fine deve essere una costante). Invece. esiste una

istruzione che effettua il calcolo desiderato. Si tratta di `LEA` (`LEA` sta per *Load Effective Address*. Il codice seguente calcola l'indirzzo di `x` e lo mette in EAX:

```
lea    eax, [ebp - 8]
```

Ora EAX contiene l'indirizzo di `x` e puo' essere messo nello stack alla chiamata della funzione `foo`. Non ti confondere, sembra che questa istruzione vada a leggere il dato in [EBP−8]; questo, pero', *non* e' vero. L'istruzione `LEA` non legge *mai* la memoria! Sempplicemente calcola l'indirizzo che sarebbe letto da un'altra istruzione e memorizza questo indirizzo nel suo primo operando di tipo registro. Dal momento che non legge nessuna memoria, nessuna definizione della dimensione di memoria (*es.* `dword`) e' richiesta o permessa.

4.7.5 Valori di ritorno

Le funzioni C non-void restituiscono un valore. Le convenzioni di chiamata del C specificano come questo deve essere fatto. I valori di ritorno sono passati attraverso i registri. Tutti i tipi integrali (`char`, `int`, `enum`, *etc.*) sono ritornati nel registro EAX. Se sono piu' piccoli di 32 bit, vengono estesi a 2 bit nel momento in cui vengono messi in EAX. (Come sono estesi dipende se sono con segno o senza segno.) I valori a 64 bit sono ritornati nella coppia di registri EDX:EAX. Anche i valori puntatore sono memorizzati in EAX. I Valori in virgola mobile sono memorizzati nel registro STO del coprocessore matematico. (Questo registro sara' discusso nel capitolo sulla virgola mobile.)

4.7.6 Altre convenzioni di chiamata

Le regole sopra descrivono la convenzione di chiamata del C standard che viene supportata da tutti i compilatori C 80x86. Spesso i compilatori supportano anche altre convenzioni di chiamata. Quando ci si interfaccia con il linguaggio assembly e' *molto* importante conoscere quale convenzione viene utilizzata dal compilatore quando chiama una funzione. Generalmente, la norma e' utilizzare la convenzione di chiamata standard. Purtroppo non e' sempre cosi'.[6]. I compilatori che usano convenzioni multiple spesso hanno opzioni sulla riga di comando che possono essere usate per cambiare la convenzione standard. Forniscono inoltre estensione della sintassi C per assegnare esplicitamente convenzioni di chiamata ad una singola funzione. Queste estensione, pero', non sono standardizzate e possono variare da un compilatore all'altro.

[6]Il compilatore C Watcom e' un'esempio di compilatore che di default *non* usa la convenzione standard. Per maggiori dettagli, vedi il codice sorgente per Watcom

Il compilatore GCC permette diverse convenzioni di chiamata. La convenzione di una funzione puo' essere esplicitamente dichiarata utilizzando l'estenzione `__attribute__`. Per esempio, per dichiarare una funzione void chiamata `f` che usa la convenzione di chiamata standard ed accetta un solo parametro di tipo `int`, si usa la seguente sintassi nel suo prototipo:

```
void f( int ) __attribute__((cdecl));
```

GCC inoltre supporta la convenzione di chiamata *standard call*. La funzione sopra, per utilizzare questa convenzione, dovrebbe essere dichiarata sostituendo la parola `cdecl` con `stdcall`. La differenza tra `stdcall` e `cdecl` e' che `stdcall` richieda che lo sottoroutine rimuova i paramtri dallo stack (come succede per la convenzione di chiamata del Pascal). Cosi', la convenzione `stdcall` puo' essere usata solo per quelle funzioni che hanno un numero prefissato di argomenti (non come ,*Es.* `printf` o `scanf`).

GCC inoltre supporta attributi aggiuntivi chiamati `regparm` che dicono al compilatore di utilizzare i registri per passare fino a 3 paremetri interi ad una funzione anziche' utilizzare lo stack. Questo e' un'esempio comune di ottimizzazione che molti compilatori supportano.

Borland e Microsoft usano una sintassi comune per dichiarare le convenzioni di chiamata. Aggiungono le parole chiavi `__cdecl` e `__stdcall` al C. Queste parole chiavi agiscono come modificatori di funzione e appaiono nel prototipo immediatamente prima del nome della funzione. Per esempio, la funzione `f` vista prima potrebbe essere definita come segue per Borland e Microsoft:

```
void __cdecl f( int );
```

Ci sono sia dei vantaggi che degli svantaggi in ognuna delle convenzioni di chiamata. I vantaggi principali della convenzione `cdecl` e' che e' semplice e molto flessibile. Puo' essere usata per qualunque tipo di funzioni C e compilatori C. L'utilizzo delle altre convenzioni puo' limitare la portabilita' di una sottoroutine. Lo svantaggio principale e' che puo' essere piu' lenta di alcune delle altre e usare piu'memoria (dal momento che ogni chiamata di funzione richiede codice per rimuovere i parametri dallo stack).

I vantaggi della convenzione `stdcall` sono che questa utilizza meno memoria rispetto `cdecl`. Nessun codice di pulizia dello stack e' richiesto dopo l'istruzione `CALL`. Lo svantaggio principale e' che non puo' essere utilizzata con le funzioni che hanno un numero variabile di argomenti.

Il vantaggio di utilizzare una convenzione che usa i registri per passare parametri interi e' la velocita'. Lo svantaggio principale e' che la convenzione e' piu' complessa. Alcuni parametri possono essere nei registri e altri nello stack.

4.7.7 Esempi

Di seguito un esempio che mostra come una routine assembly puo' essere interfacciata ad un programma in C. (Nota che questo programma non usa lo scheletro del programma assembly (Figure 1.7) o il modulo driver.c.)

main5.c

```
#include <stdio.h>
/* prototipo della routine assembly */
void calc_sum( int, int * ) __attribute__((cdecl));

int main( void )
{
  int n, sum;

   printf ("Sum integers up to: ");
  scanf("%d", &n);
  calc_sum(n, &sum);
   printf ("Sum is %d\n", sum);
  return 0;
}
```

main5.c

sub5.asm

```
1  ; subroutine _calc_sum
2  ; trova la somma degli interi da 1 a n
3  ; Parametri:
4  ;   n    - limite della somma (at [ebp + 8])
5  ;   sump - puntatore all'intero in cui memorizzare sum (a [ebp + 12])
6  ; pseudo codice C:
7  ; void calc_sum( int n, int * sump )
8  ; {
9  ;   int i, sum = 0;
10 ;   for( i=1; i <= n; i++ )
11 ;     sum += i;
12 ;   *sump = sum;
13 ; }
14
15 segment .text
16         global  _calc_sum
17 ;
```

```
Somma gli interi fino a: 10
Stack Dump # 1
EBP = BFFFFB70 ESP = BFFFFB68
 +16  BFFFFB80  080499EC
 +12  BFFFFB7C  BFFFFB80
  +8  BFFFFB78  0000000A
  +4  BFFFFB74  08048501
  +0  BFFFFB70  BFFFFB88
  -4  BFFFFB6C  00000000
  -8  BFFFFB68  4010648C
Sum e' 55
```

Figura 4.13: Esecuzione di esempio del programma sub5

```
18 ; variabile locale:
19 ;   sum a [ebp-4]
20 _calc_sum:
21         enter   4,0               ; riserva spazio per sum nello stack
22         push    ebx               ; IMPORTANTE!
23
24         mov     dword [ebp-4],0   ; sum = 0
25         dump_stack 1, 2, 4        ; stampa lo stack da ebp-8 a ebp+16
26         mov     ecx, 1            ; ecx e' i nello pseudo codice
27 for_loop:
28         cmp     ecx, [ebp+8]      ; cmp i e n
29         jnle    end_for           ; se non i <= n, esce
30
31         add     [ebp-4], ecx      ; sum += i
32         inc     ecx
33         jmp     short for_loop
34
35 end_for:
36         mov     ebx, [ebp+12]     ; ebx = sump
37         mov     eax, [ebp-4]      ; eax = sum
38         mov     [ebx], eax
39
40         pop     ebx               ; ripristina ebx
41         leave
42         ret
```

sub5.asm

Perche' la riga 22 di `sub5.asm` e' cosi' importante? Perche' la convenzione di chiamata del C richiede che il valore di EBX non venga modificato dalla chiamata di funzione. Se questo non e' fatto, e' molto probabile che il programma non funzioni correttamente.

La riga 25 dimostra come funziona la macro `dump_stack`. Ricorda che il primo parametro e' solo una etichetta numerica, e che il secondo e terzo parametro determinano quante double word visualizzare rispettivamente sotto e sopra EBP. La Figura 4.13 mostra una esecuzione di esempio del programma. Per questo dump, si puo' vedere che l'indirizzo della dword in cui memorizzare sum e' BFFFFB80 (a EBP + 12); il limite della somma e' 0000000A (a EBP + 8); l'indirizzo di ritorno per la routine e' is 08048501 (a EBP + 4); Il valore salvato di EBP e' BFFFFB88 (at EBP); il valore della variabile locale e' 0 (a EBP - 4); infine, il valore salvato di EBX e' 4010648C (a EBP - 8).

La funzione `calc_sum` dovrebbe essere riscritta per restituire la somma come suo valore di ritorno invece di usare un puntatore al parametro. Dal momento che la somma e' un valore integrale, la somma potrebbe essere depositata nel registro EAX. La riga 11 del file `main5.c` dovrebbe essere cambiata in:

```
sum = calc_sum(n);
```

Inoltra, il prototipo di `calc_sum` dovrebbe essere modificato. Sotto, il codice assembly modificato:

sub6.asm

```
1  ; subroutine _calc_sum
2  ; trova la somma degli interi da 1 a n
3  ; Parametri:
4  ;   n    - limite della somma (a [ebp + 8])
5  ; valore di ritorno:
6  ;   valore della somma
7  ; pseudo codice C:
8  ; int calc_sum( int n )
9  ; {
10 ;   int i, sum = 0;
11 ;   for( i=1; i <= n; i++ )
12 ;     sum += i;
13 ;   return sum;
14 ; }
15 segment .text
16         global  _calc_sum
17 ;
18 ; variabile locale:
```

```
segment .data
format       db "%d", 0

segment .text
...
      lea    eax, [ebp-16]
      push   eax
      push   dword format
      call   _scanf
      add    esp, 8
...
```

Figura 4.14: Chiamata di `scanf` dall'assembly

```
;   sum a [ebp-4]
_calc_sum:
        enter   4,0               ; riserva spazio per sum nello stack

        mov     dword [ebp-4],0   ; sum = 0
        mov     ecx, 1            ; ecx e' i nello pseudocodice
for_loop:
        cmp     ecx, [ebp+8]      ; cmp i e n
        jnle    end_for           ; se non i <= n, esce

        add     [ebp-4], ecx      ; sum += i
        inc     ecx
        jmp     short for_loop

end_for:
        mov     eax, [ebp-4]      ; eax = sum

        leave
        ret
```

sub6.asm

4.7.8 Chiamata di funzioni C dall'assembly

Un grande vantaggio dell'interfacciamento tra C e assembly e' che permette al codice assembly di accedere alle moltissime funzioni della libreria C e quelle scritte da altri. Per esempio, cosa deve fare uno che vuole chiamare la funzione `scanf` per leggere un'intero dalla tastiera? La Figura 4.14 mostra il codice per fare cio'. Un punto molto importante da ricordare e'

che `scanf` segue la convenzione C standard alla lettera. Questo significa che preserva i valori dei registri EBX, ECX ed EDX.I registri EAX, ECX, ed EDX invece possono essere modificati! Infatti, il registro EAX sara' alla fine modificato, visto che conterra' il valore di ritorno della chiamata `scanf`. Per altri esempi sull'utilizzo dell' interfacciamento con il C, guarda al codice in `asm_io.asm` che e' stato utilizzato per creare `asm_io.obj`.

4.8 Sottoprogramma Rientranti e Ricorsivi

Un sottoprogramma rientrante deve soddisfare le seguenti proprieta':

- Non deve modificare nessuna istruzione di codice. Nei linguaggi di alto livello cio' sarebbe molto difficile, ma in assembly non e' cosi' difficile per un programma provare a modificare il proprio codice. Per esempio:

```
mov    word [cs:$+7$], 5      ; copia 5 nella word 7 bytes avanti
add    ax, 2                 ; il comando precedente cambia 2 in 5!
```

  Questo codice funziona in modalita' reale, ma nei sistemi operativi in modalita' protetta, il segmento di codice e' contrassegnato per sola lettura. Quando la prima riga viene eseguita, il programma abortira' su questi sistemi. Questo tipo di programmazione e' pessima per molte ragioni. E' confondente, difficile da mantenere e non permette la condivisione del codice (vedi sotto).

- Non deve modificare i dati globali (come i dati nei segmenti `dati` e `bss`). Tutte le variabili sono memorizzate nello stack.

Ci sono diversi vantaggi nello scrivere codice rientrante.

- Un sottoprogramma rientrante puo' essere chiamato ricorsivamente

- Un programma rientrante puo' essere condiviso da processi multipli. Su molti sistemi operativi multi-tasking, se ci sono istanze multiple di un programma in esecuzione, solo *una* copia del codice e' in memoria. Anche le libreria condivise e le DLL(*Dynamic Link Libraries*) utilizzano questo concetto.

- I sottoprogrammi rientranti funzionano meglio nei programmi *multi-threaded*. [7] Windows 9x/NT e molti sistemi operativi tipo UNIX (Solaris, Linux, *etc.*) supportano programmi multi-threaded.

[7] Un programma multi-threaded ha molteplici tread di esecuzione. Il programma stesso, cioe', e' multi-tasking.

```
1  ; finds n!
2  segment .text
3        global _fact
4  _fact:
5        enter  0,0
6
7        mov    eax, [ebp+8]    ; eax = n
8        cmp    eax, 1
9        jbe    term_cond       ; se n <= 1, termina
10       dec    eax
11       push   eax
12       call   _fact           ; eax = fact(n-1)
13       pop    ecx             ; risposta in eax
14       mul    dword [ebp+8]   ; edx:eax = eax * [ebp+8]
15       jmp    short end_fact
16 term_cond:
17       mov    eax, 1
18 end_fact:
19       leave
20       ret
```

Figura 4.15: Funzione fattoriale ricorsiva

4.8.1 Sottoprogrammi Ricorsivi

Questi tipi di sottoprogrammi chiamano se stessi. La ricorsione puo' essere *diretta* o *indiretta*. La ricorsione diretta si ha quando un sottoprogramma, diciamo `foo`, chiama se stesso dentro il corpo di `foo`. La ricorsione indiretta sia ha quando un sottoprogramma non e' chiamato da se stesso direttamente, ma da un'altro sottoprogramma che questo chiama. Per esempio, il sottoprogramma `foo` potrebbe chiamare `bar` e `bar` a sua volta chiamare `foo`.

I sottoprogrammi ricorsivi devono avere una *condizione di terminazione*. Quando questa condizione e' vera, le chiamate ricorsive cessano. Se una routine ricorsiva non ha una condizione di terminazione o la condizione non diventa mai vera, la ricorsione non terminera' mai (molto simile ad un ciclo infinito).

La Figura 4.15 mostra una funzione che calcola ricorsivamente i fattoriali. Potrebbe essere chiamata dal C con:

```
x = fact(3);          /* trova 3! */
```

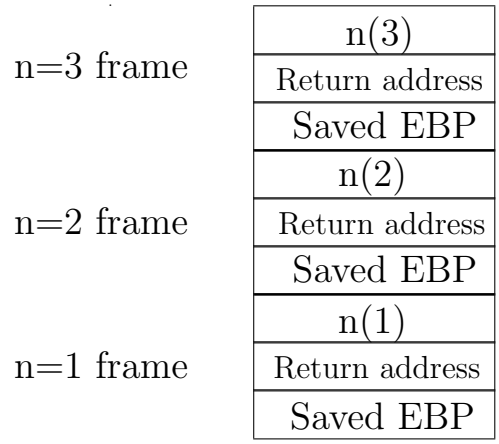


Figura 4.16: Gli Stack frame della funzione factoriale

```
void f( int x )
{
  int i;
  for( i=0; i < x; i++ ) {
     printf ("%d\n", i);
    f(i);
  }
}
```

Figura 4.17: Un'altro esempio (versione C)

La Figura 4.16 mostra come appare lo stack nell'ultimo ciclo di ricorsione per la chiamata a funzione sopra.

Le Figure 4.17 e 4.18 mostrano un'altro esempio, piu' complicato, rispettivamente in C ed in assembly. Qual' e' l'output per `f(3)`? Nota che l'istruzione `ENTER` crea una nuova `i` nello stack per ogni chiamata ricorsiva. Cosi', ogni istanza ricorsiva di `f` ha la propria variabile `i` indipendente. Definire `i` come una double word nel segmento `data` non avrebbe funzionato lo stesso.

4.8.2 Elenco dei tipi di memorizzazione delle variabili in C

Il C fornisce diversi tipi di memorizzazione di variabili.

global Queste variabili sono definete esternamente da ogni funzione e sono memorizzate in una locazione fissa di memoria (nel segmento `data` o nel segmento `bss` ed esistono dall'inizio del programma fino alla

```
1  %define i ebp-4
2  %define x ebp+8          ; useful macros
3  segment .data
4  format       db "%d", 10, 0     ; 10 = '\n'
5  segment .text
6        global _f
7        extern _printf
8  _f:
9        enter  4,0           ; riserva memoria nello stack per i
10
11       mov    dword [i], 0  ; i = 0
12 lp:
13       mov    eax, [i]      ; e' i < x?
14       cmp    eax, [x]
15       jnl    quit
16
17       push   eax           ; chiama printf
18       push   format
19       call   _printf
20       add    esp, 8
21
22       push   dword [i]     ; chiama f
23       call   _f
24       pop    eax
25
26       inc    dword [i]     ; i++
27       jmp    short lp
28 quit:
29       leave
30       ret
```

Figura 4.18: Altro esempio (versione assembly)

fine. Di default, possono essere accedute da ogni funzione all'interno del programma; Se vengono dichiarate come `static`, solo le funzioni dello stesso modulo le possono accedere (*i.e.*In termini di assembly, l'etichetta e' interna, non esterna).

static Ci sono variabili *locali* di una funzione che sono dichiarate `static`. (Sfortunatamente, il C usa la parola chiave `static` per due diversi scopi!) Anche queste variabili sono memorizzate in una locazione fissa di memoria (in `data` o `bss`), ma possono essere accedute direttamente nella funzione che le ha definite.

automatic Questo e' il tipo di default del C di una variabile definita in una funzione. Queste variabili sono allocate nello stack quando la funzione che in cui sono definite viene invocata e sono deallocate al termine della funzione. Percio' non hanno una locazione fissa in memoria.

register Questa parola chiave chiede al compilatore di utilizzare un registro per i dati in questa variabile. Questa e' solo una *richiesta*. Il compilatore *non* e' tenuto a considerarla. Se l'indirizzo della variabile e' usato in qualche parte del programma, il compilatore non prendera' in considerazione la richiesta (dal momento che i registri non hanno indirizzi). Inoltre, solo i tipi integrali semplici possono essere valori dei registri. I tipi strutturati non lo possono essere; Non entrerebbero in un registro! I compilatori C spesso trasformeranno automaticamente le normali variabili automatiche in variabili registri senza nessuna indicazione del programmatore.

volatile Questa parola chiave dice al compilatore che il valore nella variabile puo' cambiare in qualunque momento. Cio' significa che il compilatore non puo' fare nessuna assunzione circa il momento in cui la variabile viene modificata. Spesso un compilatore puo' memorizzare il valore della variabile in un registro temporaneamente ed usare il registro al posto della variabile in una sezione di codice. Questo tipo di ottimizzazione non puo' essere fatta con le variabili `volatile`. Un esempio comune di una variabile volatile potrebbe essere una variabile che viene modificata da due tread in un programma multi-threaded. Considera il codice seguente:

```
x = 10;
y = 20;
z = x;
```

Se `x` fosse modificata da un'altro tread, e' possibile che gli altri tread modifichino `x` fra le righe 1 e 3 cosicche' `z` non sarebbe 10. Invece, se

`x` non fosse dichiarata volatile, il compilatore potrebbe assumere che `x` non e' stato modificato e impostare `z` a 10.

Un'altro uso di `volatile` e' impedire al compilatore di usare un registro per una variabile

Capitolo 5

Array

5.1 Introduzione

Un *array* e' una lista di blocchi continui di dati in memoria. Ogni elemento della lista deve essere dello stesso tipo e deve usare esattamente lo stesso numero di byte di memoria per la sua memorizzazione. In virtu' di queste caratteristiche, gli array consentono un'accesso efficente ai dati attraverso la loro posizione (o indice) nell'array. L'indirizzo di ogni elemento puo' essere calcolato conoscendo tre fattori:

- L'indirizzo del primo elemento dell'array.
- Il numero di byte di ogni elemento
- L'indice dell'elemento

E' conveniente considerare 0 come l'indice del primo elemente di un'array (come in C). E' possibile utilzzare altri valori per il primo indice, ma questo complicherebbe i calcoli.

5.1.1 Definire array

Definizioni di array nei segmenti `data` e `bss`

Per definire un array inizializzato nel segmento `data`, si usano le normali direttive `db`, `dw`, *etc.*. Il NASM inoltre fornisce un direttiva molto utile chiamata `TIMES` che viene usata per ripetere un comando molte volte senza avere la necessita' di duplicare il comando a mano. La Figura 5.1 mostra diversi esempi di utilizzo di queste istruzioni.

Per definire un array non inizializzato nel segmento `bss`, si usano le direttive `resb`, `resw`, *etc.* . Ricorda che queste direttive hanno un'operando

```
1  segment .data
2  ; definisce array di 10 double words inizializzate a 1,2,..,10
3  a1           dd   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
4  ; definisce array di 10 words initializzate a 0
5  a2           dw   0, 0, 0, 0, 0, 0, 0, 0, 0, 0
6  ; come sopra utilizzando TIMES
7  a3           times 10 dw 0
8  ; definisce array di byte con 200 0 e poi 100 1
9  a4           times 200 db 0
10             times 100 db 1
11
12 segment .bss
13 ; definisce array di 10 double word non initializzate
14 a5           resd  10
15 ; definisce array di 100 word non initializzate
16 a6           resw  100
```

Figura 5.1: Defining arrays

che specifica quante unita' di memoria riservare. La Figura 5.1 mostra di nuovo gli esempi per questo tipo di definizioni.

Definizione di array come variabili locali nello stack

Non esiste un metodo diretto per definire una variabile array locale nello stack. Come prima, occorre calcolare il numero totale di byte richiesti da *tutte* le variabili locali, inclusi gli array e sottrarre questo valore da ESP (direttamente o utilizzando l'istruzione `ENTER`). Per esempio, se una funzione ha bisogno di una variabile carattere, due interi double word e di un array di 50 elementi word, sono necessari $1 + 2 \times 4 + 50 \times 2 = 109$ byte. Occorre pero' notare che il numero sottratto da ESP deve essere un multiplo di 4 (112 in questo caso) per mantenere ESP al limite di una double word. E' possibile arrangiare le variabili dentro questi 109 byte in diversi modi. La Figura 5.2 mostra due possibili modi. La parte non utilizzata del primo ordinamento serve per mantenere le double word entro i limiti di double word per velocizzare gli accessi alla memoria.

5.1.2 Accedere ad elementi dell'array

Non esiste in linguaggio assembly l'operatore `[ ]` come in C. Per accedere ad un'elemento dell'array, deve essere calcolato il suo indirizzo. Consideria-

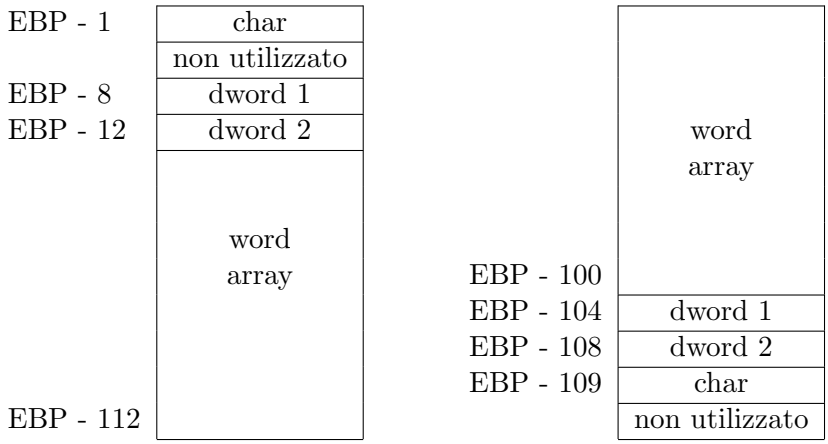


Figura 5.2: Disposizione nello stack

mo la seguente definizione di due array:

```
array1      db     5, 4, 3, 2, 1     ; array di byte
array2      dw     5, 4, 3, 2, 1     ; array di word
```

Ecco alcuni esempi di utilizzo di questi array:

```
1       mov    al, [array1]            ; al = array1[0]
2       mov    al, [array1 + 1]        ; al = array1[1]
3       mov    [array1 + 3], al        ; array1[3] = al
4       mov    ax, [array2]            ; ax = array2[0]
5       mov    ax, [array2 + 2]        ; ax = array2[1] (NON array2[2]!)
6       mov    [array2 + 6], ax        ; array2[3] = ax
7       mov    ax, [array2 + 1]        ; ax = ??
```

Alla riga 5 viene referenziato l'elemento 1 dell'array di word, non l'elemento 2. Perche'? Le word sono unita' a 2 byte, cosi' per spostarsi all'elemento successione di un array di word, occorre muoversi in aventi di 2 byte, non di 1. La riga 7 leggera' un byte dal primo elemento ed un byte dal secondo. In C il compilatore controlla il tipo del puntatore per verificare di quanti byte si deve spostare in una espressione che usa l'aritmetica dei puntatori cosicche' il programmatore non se ne deve occupare. In assembly invece, spetta al programmatore di tener conto della dimensione degli elementi di un array quando si sposta da un'elemento all'altro.

La Figura 5.3 mostra un pezzo di codice che somma tutti gli elementi di `array1` del precedente esempio di codice. Alla riga 7, AX e' sommato a DX. Perche' non AL? Prima di tutto i due operandi della istruzione `ADD` devono avere la stessa dimensione. Secondariamente, sarebbe piu' facile nella somma di byte, raggiungere un valore troppo grande per stare in un

```
1       mov    ebx, array1          ; ebx = indirizzo di array1
2       mov    dx, 0                ; dx manterra' la somma
3       mov    ah, 0                ; ?
4       mov    ecx, 5
5   lp:
6       mov    al, [ebx]            ; al = *ebx
7       add    dx, ax               ; dx += ax (non al!)
8       inc    ebx                  ; bx++
9       loop   lp
```

Figura 5.3: Somma degli elementi di un array (Versione 1)

```
1       mov    ebx, array1          ; ebx = indirizzo di array1
2       mov    dx, 0                ; dx manterra' la somma
3       mov    ecx, 5
4   lp:
5       add    dl, [ebx]            ; dl += *ebx
6       jnc    next                 ; se no carry va a next
7       inc    dh                   ; inc dh
8   next:
9       inc    ebx                  ; bx++
10      loop   lp
```

Figura 5.4: Somma degli elementi di un array (Versione 2)

byte. Utilizzando DX, e' possibile arrivare fino a 65.535. E' comunque da notare che anche AH viene sommato. E percio' AH e' impostato a zero[1] alla riga 3.

Le Figure 5.4 e 5.5 mostrano due modi alternativi per calcolare la somma. Le righe in corsivo sostituiscono le righe 6 e 7 della Figura 5.3.

5.1.3 Indirizzamento Indiretto Avanzato

Non sorprende che l'indirizzamento indiretto venga spesso utilizzato con gli array. La forma piu' comune di riferimento indiretto alla memoria e':

*[base reg + factor*index reg + constant]*

[1] Impostare AH a 0 significa considerare implicitamente AL come un numero unsigned. Se fosse signed, l'operazione corretta sarebbe quella di inserire una istruzione `CBW` fra le righe 6 e 7

```
1       mov     ebx, array1             ; ebx = indirizzo di array1
2       mov     dx, 0                   ; dx manterra' la somma
3       mov     ecx, 5
4   lp:
5       add     dl, [ebx]               ; dl += *ebx
6       adc     dh, 0                   ; dh += carry flag + 0
7       inc     ebx                     ; bx++
8       loop    lp
```

Figura 5.5: Somma degli elementi di un array (Versione 3)

dove:

base reg e' uno dei registri EAX, EBX, ECX, EDX, EBP, ESP, ESI o EDI.

factor puo' essere 1, 2, 4 o 8. (se 1, il fattore e' omesso.)

index reg e' uno dei registri EAX, EBX, ECX, EDX, EBP, ESI, EDI. (Nota che ESP non e' nella lista)

constant e' una costante a 32 bit. La costante puo' essere una etichetta(o una espressione etichetta).

5.1.4 Esempio

Ecco un'esempio che utilizza un array e lo passa ad una funzione. Viene utilizzato il programma `array1c.c` (riproposto sotto) come driver, invece di `driver.c`.

array1.asm

```
1   %define ARRAY_SIZE 100
2   %define NEW_LINE 10
3
4   segment .data
5   FirstMsg        db   "First 10 elements of array", 0
6   Prompt          db   "Enter index of element to display: ", 0
7   SecondMsg       db   "Element %d is %d", NEW_LINE, 0
8   ThirdMsg        db   "Elements 20 through 29 of array", 0
9   InputFormat     db   "%d", 0
10
11  segment .bss
12  array           resd ARRAY_SIZE
13
```

```
14  segment .text
15          extern  _puts, _printf, _scanf, _dump_line
16          global  _asm_main
17  _asm_main:
18          enter   4,0               ; variabile locale dword a EBP - 4
19          push    ebx
20          push    esi
21
22  ; inizializza l'array a 100, 99, 98, 97, ...
23
24          mov     ecx, ARRAY_SIZE
25          mov     ebx, array
26  init_loop:
27          mov     [ebx], ecx
28          add     ebx, 4
29          loop    init_loop
30
31          push    dword FirstMsg         ; stampa FirstMsg
32          call    _puts
33          pop     ecx
34
35          push    dword 10
36          push    dword array
37          call    _print_array  ; stampa i primi 10 elementi dell'array
38          add     esp, 8
39
40  ; richiede all'utente l'indice degli elementi
41  Prompt_loop:
42          push    dword Prompt
43          call    _printf
44          pop     ecx
45
46          lea     eax, [ebp-4]      ; eax = indirizzo della dword locale
47          push    eax
48          push    dword InputFormat
49          call    _scanf
50          add     esp, 8
51          cmp     eax, 1            ; eax = valore di ritorno di scanf
52          je      InputOK
53
54          call    _dump_line  ; dump del resto della riga e passa oltre
55          jmp     Prompt_loop          ; se l'input non e' valido
```

```
56
57  InputOK:
58          mov     esi, [ebp-4]
59          push    dword [array + 4*esi]
60          push    esi
61          push    dword SecondMsg      ; stampa il valore dell'elemento
62          call    _printf
63          add     esp, 12
64
65          push    dword ThirdMsg       ; stampa gli elemeni 20-29
66          call    _puts
67          pop     ecx
68
69          push    dword 10
70          push    dword array + 20*4     ; indirizzo di array[20]
71          call    _print_array
72          add     esp, 8
73
74          pop     esi
75          pop     ebx
76          mov     eax, 0            ; ritorna al C
77          leave
78          ret
79
80  ;
81  ; routine _print_array
82  ; routine richiamabile da C che stampa gli elementi di un array double word come
83  ; interi con segno.
84  ; prototipo C:
85  ; void print_array( const int * a, int n);
86  ; Parametri:
87  ;   a - puntatore all'array da stampare (in ebp+8 sullo stack)
88  ;   n - numero di interi da stampare  (in ebp+12 sullo stack)
89
90  segment .data
91  OutputFormat    db   "%-5d %5d", NEW_LINE, 0
92
93  segment .text
94          global  _print_array
95  _print_array:
96          enter   0,0
97          push    esi
```

```
98          push    ebx
99
100         xor     esi, esi                ; esi = 0
101         mov     ecx, [ebp+12]           ; ecx = n
102         mov     ebx, [ebp+8]            ; ebx = indirizzo dell'array
103 print_loop:
104         push    ecx                     ; printf potrebbe modificare ecx!
105
106         push    dword [ebx + 4*esi]     ; mette array[esi]
107         push    esi
108         push    dword OutputFormat
109         call    _printf
110         add     esp, 12                 ; remuove i parametri (lascia ecx!)
111
112         inc     esi
113         pop     ecx
114         loop    print_loop
115
116         pop     ebx
117         pop     esi
118         leave
119         ret
```

array1.asm

array1c.c

```
#include <stdio.h>

int asm_main( void );
void dump_line( void );

int main()
{
  int ret_status ;
   ret_status = asm_main();
  return ret_status ;
}

/*
 * function dump_line
 * dump di tutti i char lasciati nella riga corrente dal buffer di input
 */
void dump_line()
```

```
{
  int ch;

  while( (ch = getchar()) != EOF && ch != '\n')
    /* null body*/ ;
}
```

array1c.c

L'istruzione `LEA` rivisitata

L'istruzione `LEA` puo' essere usata anche per altri scopi oltre quello di calcolare indirizzi. Un uso abbastanza comune e' per i calcoli veloci. Consideriamo l'istruzione seguente:

```
lea    ebx, [4*eax + eax]
```

Questa memorizza effettivamente il valore di $5 \times$ `EAX` in EBX. Utilizzare `LEA` per questo risulta piu' semplice e veloce rispetto all'utilizzo di `MUL`. E' da notare pero' che l'espressione dentro le parentesi quadre *deve* essere un'indirizzo indiretto valido. Questa istruzione quindi non puo' essere usata per moltiplicare per 6 velocemente.

5.1.5 Array Multidimensionali

Gli array multidimensionali non sono in realta' molto diversi rispetto agli array ad una dimensione discussi prima. Infatti, vengono rappresentati in memoria nello stesso modo, come un'array piano ad una dimensione.

Array a Due Dimensioni

Non sorprende che il piu' semplice array multidimensionale sia bidimensionale. Un array a due dimensioni e' spesso rappresentato con una griglia di elementi. Ogni elemento e' identificato da una coppia di indici. Per convenzione, il primo indice rappresenta la riga dell'elemento ed il secondo indice la colonna.

Consideriamo un array con tre righe e 2 colonne definito come:

```
int a [3][2];
```

Il compilatore C riservera spazio per un array di 6 ($= 2 \times 3$) interi e mappera' gli elementi i questo modo:

```
1   mov    eax, [ebp - 44]          ; ebp - 44 e' l'\emph{i}esima locazione
2   sal    eax, 1                   ; multiplica i per 2
3   add    eax, [ebp - 48]          ; somma j
4   mov    eax, [ebp + 4*eax - 40] ; ebp - 40 e' l'indirizzo di a[0][0]
5   mov    [ebp - 52], eax          ; memorizza il risultato in x (in ebp - 52)
```

Figura 5.6: Codice assembly per `x = a[i][j]`

Indice	0	1	2	3	4	5
Elemento	a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

La Tabella vuole dimostrare che l'elemento referenziato con `a[0][0]` e' memorizzato all'inizio dell'array a una dimensione a 6 elementi. L'elemento `a[0][1]` e' memorizzato nella posizione successiva (indice 1) e cosi' via. Ogni righa dell'array a due dimensioni e' memorizzata sequenzialmente. L'ultimo elemento di una riga e' seguito dal primo elemento della riga successiva. Questa rappresentazione e' conosciuta come rappresentazione a livello di riga ed e' il modo in cui i compilatori C/C++ rappresentano l'array.

In che modo il compilatore determina dove si trova l'elemento `a[i][j]` nella rappresentazione a livello di riga? Una formula calcolera' l'indice partendo da `i` e `j`. La formula in questo caso e' $2i + j$. Non e' difficile capire da dove deriva questa formula. Ogni riga ha due elementi; Il primo elemento della riga i si trova in posizione $2i$. La posizione della colonna j e' quindi data dalla somma di j a $2i$. Questa analisi ci mostra inoltre come e' possibile generalizzare questa formula ad un array con `N` colonne: $N \times i + j$. Nota che la formula *non* dipende dal numero di righe.

Come esempio, vediamo come *gcc* compila il codice seguente (utilizzando l'array `a` definito prima):

```
x = a[i][j];
```

La Figura 5.6 mostra l'assembly generato dalla traduzione. In pratica, il compilatore converte il codice in:

```
x = *(&a[0][0] + 2*i + j);
```

ed infatti il programmatore potrebbe scrivere in questo modo con lo stesso risultato.

Non c'e' nulla di particolare nella scelta della rappresentazione a livello di riga. Una rappresentazione a livello di colonna avrebbe dato gli stessi risultati:

Indice	0	1	2	3	4	5
Elemento	a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]

Nella rappresentazione a livello di colonna, ogni colonna e' memorizzata contiguamente. L'elemento `[i][j]` e' memorizzato nella posizione $i + 3j$. Altri linguaggi (per esempio, il FORTRAN) utilizzano la rappresentazione a livello di colonna. Questo e' importante quando si interfaccia codice con linguaggi diversi.

Dimensioni Oltre Due

La stessa idea viene applicata agli array con oltre due dimensioni. Consideriamo un array a tre dimensioni:

```
int b [4][3][2];
```

Questo array sarebbe memorizzato come se fossero 4 array consecutivi a due dimensioni ciascuno di dimensione `[3][2]`. La Tabella sotto mostra come inizia:

Indice	0	1	2	3	4	5
Elemento	b[0][0][0]	b[0][0][1]	b[0][1][0]	b[0][1][1]	b[0][2][0]	b[0][2][1]
Indice	6	7	8	9	10	11
Elemento	b[1][0][0]	b[1][0][1]	b[1][1][0]	b[1][1][1]	b[1][2][0]	b[1][2][1]

la formula per calcolare la posizione di `b[i][j][k]` e' $6i + 2j + k$. Il 6 e' determinato dalla dimensione degli array `[3][2]`. In generale, per ogni array dimensionato come `a[L][M][N]`, la posizione dell'elemento `a[i][j][k]` sara' $M \times N \times i + N \times j + k$. E' importante notare di nuovo che la prima dimensione (`L`) non appare nella formula.

Per dimensioni maggiori, viene generalizato lo stesso processo. Per un array a n dimensioni, di dimensione da D_1 a D_n, la posizione dell'elemento indicato dagli indici i_1 a i_n e' dato dalla formula:

$$D_2 \times D_3 \cdots \times D_n \times i_1 + D_3 \times D_4 \cdots \times D_n \times i_2 + \cdots + D_n \times i_{n-1} + i_n$$

oppure per i patiti della matematica, puo' essere scritta piu' succintamente come:

$$\sum_{j=1}^{n} \left(\prod_{k=j+1}^{n} D_k \right) i_j$$

La prima dimensione, D_1, non appare nella formula.

Da questo puoi capire che l'autore e' laureato in fisica.(il riferimento al FORTRAN era una rivelazione)

Per la rappresentazione a livello di colonne, la formula generale sarebbe:

$$i_1 + D_1 \times i_2 + \cdots + D_1 \times D_2 \times \cdots \times D_{n-2} \times i_{n-1} + D_1 \times D_2 \times \cdots \times D_{n-1} \times i_n$$

o sempre per i patiti della matematica:

$$\sum_{j=1}^{n} \left(\prod_{k=1}^{j-1} D_k \right) i_j$$

In questo caso, e' l'ultima dimensione, D_n, che non appare nella formula.

Passaggio di Array Multidimensionali come Parametri in C

La rappresentazione a livello di riga degli array multidimensionali ha un effetto diretto nella programmazione in C. Per gli array ad una dimesione, ls dimensione dell'array non e' necessaria per il calcolo della posizione di uno specifico elemento in memoria. Questo non e' vero per gli array multidimensionali. Per accedere agli elementi di questi array, il compilatore deve conoscere tutte le dimensioni ad eccezione della prima. Questo diventa apparente quando consideriamo il prototipo di una funzione che accetta un array multidimensionale come parametro. Il seguente prototipo non verrebbe compilato:

```
void f( int a[ ][ ] );  /* nessuna informazione sulla dimensione */
```

Invece questo verrebbe compilato:

```
void f( int a[ ][2] );
```

Ogni array bidimensionale con due colonne puo' essere passato a questa funzione. La prima dimensione non e' necessaria[2].

Da non confondere con una funzione con questo prototipo:

```
void f( int * a[ ] );
```

Questo definisce un puntatore ad un array di interi ad una dimensione (che accidentalmente puo' essere usato per creare un array di array che funziona come un array a due dimensioni).

Per gli array con piu' dimensioni, tutte le dimensioni, ad eccezione della prima devono essere specificate nel parametro. Per esempio, un parametro array a 4 dimensioni dovrebbe essere passato in questo modo:

```
void f( int a[ ][4][3][2] );
```

5.2 Istruzioni Array/String

La famiglia dei processori 80x86 fornisce diverse istruzioni che sono create espressamente per lavorare con gli array. Queste istruzioni sono chiamate *istruzioni stringa*. Utilizzano i registri indice(ESI e EDI) per eseguire una operazione con incremento o decremento automatico di uno dei due registri indice. Il *flag di direzione*(DF) nel registro FLAGS indica se i registri indice sono incrementati o decrementati. Ci sono due istruzioni che modificano il flag di direzione:

CLD pulisce il flag di direzione. In questo stato, i registri indice sono incrementati.

[2]Una dimesione puo' essere specificata in questo caso, ma sarebbe ignorata dal compilatore.

LODSB AL = [DS:ESI] ESI = ESI ± 1	STOSB [ES:EDI] = AL EDI = EDI ± 1
LODSW AX = [DS:ESI] ESI = ESI ± 2	STOSW [ES:EDI] = AX EDI = EDI ± 2
LODSD EAX = [DS:ESI] ESI = ESI ± 4	STOSD [ES:EDI] = EAX EDI = EDI ± 4

Figura 5.7: Istruzioni string di Lettura e Scrittura

```
1  segment .data
2  array1  dd  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
3
4  segment .bss
5  array2  resd 10
6
7  segment .text
8        cld                    ; non dimenticare questo!
9        mov    esi, array1
10       mov    edi, array2
11       mov    ecx, 10
12 lp:
13       lodsd
14       stosd
15       loop  lp
```

Figura 5.8: Esempio di caricamento e memorizzazione

STD imposta il flag di direzione. In questo stato, i registri indice sono decrementati.

Un errore *molto* comune nella programmazione 80x86 e' quello di dimenticare di impostare esplicitamente il flag di direzione nello stato corretto. Questo porta a codice che funzione per la maggior parte delle volte (quando il flag di direzione sembra essere nello stato desiderato), ma non funziona *tutte* le volte.

5.2.1 Lettura e scrittura della memoria

Le istruzioni stringa di base possono leggere la memoria, scrivere in memoria oppure fare entrambe le operazioni. Possono leggere o scrivere un byte, una word o una double word alla volta. La Figura 5.7 mostra queste istruzioni con una piccola decrizione in pseudo codice di cosa fanno. Ci sono

MOVSB	byte [ES:EDI] = byte [DS:ESI] ESI = ESI ± 1 EDI = EDI ± 1
MOVSW	word [ES:EDI] = word [DS:ESI] ESI = ESI ± 2 EDI = EDI ± 2
MOVSD	dword [ES:EDI] = dword [DS:ESI] ESI = ESI ± 4 EDI = EDI ± 4

Figura 5.9: Istruzioni stringa di spostamento memoria

```
1  segment .bss
2  array  resd 10
3
4  segment .text
5        cld                    ; non dimenticarlo!
6        mov    edi, array
7        mov    ecx, 10
8        xor    eax, eax
9        rep stosd
```

Figura 5.10: Esempio con array di zero

alcuni punti su cui porre l'attenzione. Prima di tutto, ESI e' usato per la lettura e EDI per la scrittura. E' facile da ricordare se si tiena a mente che SI sta per *Source Index*(indice sorgente) e DI sta per *Destination Index*(indice destinazione). Inoltre, e' da notare che il registro che mantiene i dati e' fisso (AL,AX o EAX). Infine, le istruzioni di memorizzazione utilizzano ES per determinare il segmento in cui scrivere e non DS. Nella programmazione in modalita' protetta questo non costituisce un problema dal momento che c'e' solo un segmento dati ed ES viene automaticamente inizializato per riferenziare questo (come accade per DS). Nella programmazione in modalita' reale, invece, e' *molto* importante per il programmatore inizializzare ES al corretto valore di selezione del segmento[3]. La Figura 5.8 mostra un esempio di utilizzo di queste istruzioni per la copia di un array in un'altro.

La combinazione delle istruzioni `LODSx` e `STOSx` (alle righe 13 e 14 del-

[3] Un'altra complicazione sta nel fatto che non e' possibile copiare direttamente il valore del registro DS nel registro ES con una istruzione `MOV`. Il valore di DC invece, deve essere copiato in un registro generale (come AX) e poi copiato da li' nel registro ES utilizzando due istruzioni `MOV`.

CMPSB	compara il byte [DS:ESI] e il byte [ES:EDI] ESI = ESI ± 1 EDI = EDI ± 1
CMPSW	compara la word [DS:ESI] e la word [ES:EDI] ESI = ESI ± 2 EDI = EDI ± 2
CMPSD	compara la dword [DS:ESI] e la dword [ES:EDI] ESI = ESI ± 4 EDI = EDI ± 4
SCASB	compara AL e [ES:EDI] EDI ± 1
SCASW	compara AX e [ES:EDI] EDI ± 2
SCASD	compara EAX e [ES:EDI] EDI ± 4

Figura 5.11: Istruzioni stringa di comparazione

la Figura 5.8) e' molto comune. Infatti, questa combinazione puo' essere eseguita con una solo istruzione stringa `MOVSx`. La Figura 5.9 descrive le operazioni che vengono eseguite da queste istruzioni. Le righe 13 e 14 della Figura 5.8 potrebbero essere sostituite con una sola istruzione `MOVSD` ottenendo lo stesso risultato. L'unica differenza sta nel fatto che il registro EAX non verrebbe usato per nulla nel ciclo.

5.2.2 Il prefisso di istruzione `REP`

La famiglia 80x86 fornisce uno speciale prefisso di istruzione[4] chiamato `REP` che puo' essere usato con le istruzioni stringa di cui sopra. Questo prefisso dice alla CPU di ripetere la successiva istruzione stringa un numero di volte specificato. Il registro ECX e' usato per contare le iterazioni (come per l'istruzione `LOOP`). Utilizzando il prefisso `REP`, il ciclo in Figura 5.8 (righe dalla 12 alla 15) puo' essere sostituito con una singola riga:

```
rep movsd
```

La Figura 5.10 mostra un'altro esempio che azzera il contenuto di un array.

[4] Un prefisso di istruzione non e' una istruzione, ma uno speciale byte che anteposto alla istruzione stringa ne modifica il comportamento. Altri prefissi sono anche usati per sovrappore i difetti dei segmenti degli accessi alla memoria

```
1  segment .bss
2  array        resd 100
3
4  segment .text
5        cld
6        mov    edi, array    ; puntatore all'inizio dell'array
7        mov    ecx, 100      ; numero di elementi
8        mov    eax, 12       ; numero da scansionare
9  lp:
10       scasd
11       je     found
12       loop   lp
13   ; codice da eseguire se non trovato
14       jmp    onward
15 found:
16       sub    edi, 4      ; edi ora punta a 12 nell'array
17   ; codice da eseguire se trovato
18 onward:
```

Figura 5.12: Esempio di ricerca

`REPE`, `REPZ`	ripete l'istruzione finche' lo Z flag e' settato o al massimo ECX volte
`REPNE`, `REPNZ`	ripete l'istruzione finche' lo Z flag e' pulito o al massimo ECX volte

Figura 5.13: Prefissi di istruzione `REPx`

5.2.3 Istruzioni stringa di comparazione

La Figura 5.11 mostra diverse nuove istruzioni stringa che possono essere usate per comparare memoria con altra memoria o registro. Questo sono utili per la ricerca e la comparazione di array. Impostano il registro FLAGS come l'istruzione `CMP`. Le istruzioni `CMPSx` comparano le corrispondenti locazioni di memoria e le istruzioni `SCASx` scansionano le locazioni di memoria per un valore specifico.

La Figura 5.12 mostra un piccolo pezzo di codice che ricerca il numero 12 in un array di double word. L'istruzione `SCASD` alla riga 10 aggiunge sempre 4 a EDI, anche se il valore cercato viene trovato. Cosi' se si vuole conoscere l'indirizzo del 12 trovato nell'array, e' necessario sottrarre 4 da EDI (come fa la riga 16).

```
1  segment .text
2        cld
3        mov    esi, block1        ; indirizzo del primo blocco
4        mov    edi, block2        ; indirizzo del secondo blocc
5        mov    ecx, size          ; dimensione dei blocchi in bytes
6        repe   cmpsb              ; ripete finche' Z flag e' settato
7        je     equal              ; se Z e' settato, blocchi uguali
8     ; codice da eseguire se i blocchi non sono uguali
9        jmp    onward
10 equal:
11    ; codice da eseguire se sono uguali
12 onward:
```

Figura 5.14: Comparazione di blocchi di memoria

5.2.4 Prefissi di istruzione `REPx`

Ci sono diversi altri prefissi di istruzione tipo `REP` che possono essere usati con le istruzioni stringa di comparazione. La Figura 5.13 mostra i due nuovi prefissi e descrive le loro operazioni. `REPE` e `REPZ` sono sinonimi dello stesso prefisso (come lo sono `REPNE` e `REPNZ`). Se l'istruzione stringa di comparazione ripetuta si interrompe a causa del risultato della comparazione, il registro o i registri indice sono ancora incrementati ed ECX decrementato; Il registro FLAGS mantiene comunque lo stato che ha terminato al ripetizione. E' cosi' possibile usare lo Z flag per determinare se la ripetizione della comparazione e' terminata a causa della comparazione stessa o per ECX che e' diventato 0.

Perche' non controllare se ECX e' zero dopo la comparazione ripetuta?

La Figura 5.14 mostra un pezzo di codice di esempio che stabilisce se due blocchi di memoria sono uguali. Il `JE` alla riga 7 dell'esempio controlla per vedere il risultato della istruzione precedente. Se la comparazione ripetuta e' terminata perche' sono stati trovati due byte non uguali, lo Z flag sara' ancora pulito e non ci sara' nessun salto; Se le comparazioni terminano perche' ECX e' diventato zero, lo Z flag sara' ancora settato ed il codice saltera' all'etichetta `equal`.

5.2.5 Esempio

Questa sezione contiene un file sorgente in assembly con diverse funzioni che implementano le operazioni sugli array utilizzando le istruzioni stringa. Molte di queste funzioni duplicano le note funzioni della libreria C.

`memory.asm`

```
1  global _asm_copy, _asm_find, _asm_strlen, _asm_strcpy
2
3  segment .text
4  ; funzione _asm_copy
5  ; copia blocchi di memoria
6  ; prototipo C
7  ; void asm_copy( void * dest, const void * src, unsigned sz);
8  ; parametri:
9  ;   dest - puntatore al buffer in cui copiare
10 ;   src  - puntatore al buffer da cui copiare
11 ;   sz   - numero di byte da copiare
12
13 ; di seguito sono definiti alcuni simbli utili
14
15 %define dest [ebp+8]
16 %define src  [ebp+12]
17 %define sz   [ebp+16]
18 _asm_copy:
19         enter   0, 0
20         push    esi
21         push    edi
22
23         mov     esi, src        ; esi = indirizzo del buffer da cui copiare
24         mov     edi, dest       ; edi = indirizzo del in cui copiare
25         mov     ecx, sz         ; ecx = numero di byte da copiare
26
27         cld                     ; pulisce il flag di direzione
28         rep     movsb           ; esegue movsb ECX volte
29
30         pop     edi
31         pop     esi
32         leave
33         ret
34
35
36 ; funzione _asm_find
37 ; cerca in memoria un certo byte
38 ; void * asm_find( const void * src, char target, unsigned sz);
39 ; parametri:
40 ;   src    - puntatore al buffer di ricerca
41 ;   target - valore del byte da cercare
42 ;   sz     - numero di byte nel buffer
```

```
43 ; valore di ritorno:
44 ;   Se il valore e' trovato, e' ritornato il puntatore alla prima occorrenza del valore
45 ;   nel buffer
46 ;   altrimenti
47 ;     e' ritonato NULL
48 ; NOTE: target e' un valore byte,ma e' messo nello stack come un valore dword.
49 ;       Il valore byte e' memorizzato negli 8 bit bassi.
50 ;
51 %define src    [ebp+8]
52 %define target [ebp+12]
53 %define sz     [ebp+16]
54
55 _asm_find:
56         enter   0,0
57         push    edi
58
59         mov     eax, target     ; al ha il valore da cercare
60         mov     edi, src
61         mov     ecx, sz
62         cld
63
64         repne   scasb           ; cerca finche' ECX == 0 o [ES:EDI] == AL
65
66         je      found_it        ; se zero flag e' settato, il valore e' trovato
67         mov     eax, 0          ; se non trovato, ritorna puntatore NULL
68         jmp     short quit
69 found_it:
70         mov     eax, edi
71         dec     eax              ; se trovato ritorna (DI - 1)
72 quit:
73         pop     edi
74         leave
75         ret
76
77
78 ; funzione _asm_strlen
79 ; ritorno la dimensione della stringa
80 ; unsigned asm_strlen( const char * );
81 ; parametri:
82 ;   src - puntatore alla stringa
83 ; valore di riton:
84 ;   numero di caratteri della stringa (non contano lo 0 finale) (in EAX)
```

```
85
86  %define src [ebp + 8]
87  _asm_strlen:
88          enter   0,0
89          push    edi
90
91          mov     edi, src        ; puntatore alla stringa
92          mov     ecx, 0FFFFFFFFh ; usa il piu' grande ECX possibile
93          xor     al,al           ; al = 0
94          cld
95
96          repnz   scasb           ; cerca il terminatore 0
97
98  ;
99  ; repnz fara' un passo in piu', cosi' la lunghezza e' FFFFFFFE - ECX,
100 ; non FFFFFFFF - ECX
101 ;
102         mov     eax,0FFFFFFFEh
103         sub     eax, ecx        ; lunghezza = 0FFFFFFFEh - ecx
104
105         pop     edi
106         leave
107         ret
108
109 ; funzione _asm_strcpy
110 ; copia una stringa
111 ; void asm_strcpy( char * dest, const char * src);
112 ; parametri:
113 ;   dest - puntatore alla stringa in cui copiare
114 ;   src  - puntatore alla stringa da copiare
115 ;
116 %define dest [ebp + 8]
117 %define src  [ebp + 12]
118 _asm_strcpy:
119         enter   0,0
120         push    esi
121         push    edi
122
123         mov     edi, dest
124         mov     esi, src
125         cld
126 cpy_loop:
```

```
127         lodsb                   ; carica AL & inc si
128         stosb                   ; memorizza AL & inc di
129         or      al, al          ; imposta i flag di condizione
130         jnz     cpy_loop        ; se non e' passato il terminatore 0, continua
131
132         pop     edi
133         pop     esi
134         leave
135         ret
```

memory.asm

memex.c

```
#include <stdio.h>

#define STR_SIZE 30
/* prototipi */

void asm_copy( void *, const void *, unsigned ) __attribute__((cdecl));
void * asm_find( const void *,
                 char target, unsigned ) __attribute__((cdecl));
unsigned asm_strlen( const char * ) __attribute__((cdecl));
void asm_strcpy( char *, const char * ) __attribute__((cdecl));

int main()
{
  char st1[STR_SIZE] = "test string";
  char st2[STR_SIZE];
  char * st;
  char   ch;

  asm_copy(st2, st1, STR_SIZE);  /* copia tutti i 30 caratteri della stringa */
   printf ("%s\n", st2);

   printf ("Enter a char: ");  /* cerca il byte nella stringa */
  scanf("%c%*[^\n]", &ch);
  st = asm_find(st2, ch, STR_SIZE);
  if ( st )
    printf ("Found it: %s\n", st);
  else
    printf ("Not found\n");

  st1[0] = 0;
```

```
   printf ("Enter  string :" );
  scanf("%s", st1);
   printf ("len  = %u\n", asm_strlen(st1));

  asm_strcpy( st2, st1);      /* copia i dati significativi nella stringa */
   printf ("%s\n", st2 );

  return 0;
}
```

memex.c

Capitolo 6

Virgola Mobile

6.1 Rappresentazione in Virgola Mobile

6.1.1 Numeri Binari non Interi

Quando abbiamo discusso dei sistemi numerici nel primo capitolo, abbiamo parlato solo di numeri interi. Ovviamente, deve essere possibile rappresentare i numeri non interi in altre base oltre che in decimale. In decimale, le cifre alla destra della virgola decimale hanno associata una potenza negativa del 10:

$$0,123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

Non sorprende che i numeri binari seguano regole simili:

$$0,101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0,625$$

Questo principio puo' essere combinato con il metodo per gli interi del capitolo 1 per convertire un numero generico:

$$110,011_2 = 4 + 2 + 0,25 + 0,125 = 6,375$$

Convertire da decimale a binario non e' molto difficile. In generale si divide il numero decimale in due parti: intero e frazione. Si converte la parte intera in binario usando i metodi del capitolo 1. La parte frazionaria viene convertita utilizzando il metodo descritto sotto.

Consideriamo una frazione binaria con i bit chiamati $a, b, c, \ldots$ Il numero in binaria apparirebbe cosi':

$$0, abcdef \ldots$$

Si moltiplica il numero per due. La rappresentazione binaria del nuovo numero sara':

$$a.bcdef \ldots$$

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
0,5625 \times 2 & = & 1,125 & \text{primo bit} & = & 1 \\
0,125 \times 2 & = & 0,25 & \text{secondo bit} & = & 0 \\
0,25 \times 2 & = & 0,5 & \text{terzo bit} & = & 0 \\
0,5 \times 2 & = & 1,0 & \text{quarto bit} & = & 1
\end{array}
$$

Figura 6.1: Conversione di 0,5625 in binario

$$
\begin{array}{rcl}
0,85 \times 2 & = & 1,7 \\
0,7 \times 2 & = & 1,4 \\
0,4 \times 2 & = & 0,8 \\
0,8 \times 2 & = & 1,6 \\
0,6 \times 2 & = & 1,2 \\
0,2 \times 2 & = & 0,4 \\
0,4 \times 2 & = & 0,8 \\
0,8 \times 2 & = & 1,6
\end{array}
$$

Figura 6.2: Convertire 0,85 in binario

Nota che il primo bit e' in posizione 1. Si sostituisce a con 0 ed otteniamo:

$$0.bcdef \ldots$$

moltiplichiamo di nuovo per due:

$$b.cdef \ldots$$

Ora il secondo bit (b) e' in posizione 1. Questa procedura puo' essere ripetuta fino a trovare i bit necessari. La Figura 6.1 mostra un'esempio reale che converte 0,5625 in binario. Il metodo si ferma quando la viene raggiunta una frazione di zero.

Come altro esempio, consideriamo la conversione di 23,85 in binario. La parte intera e' facile da convertire ($23 = 10111_2$), e la parte frazionaria $(0,85)$? La Figura 6.2 mostra l'inizio di questo calcolo. Guardando i numeri

attentamente, ci accorgiamo di essere in un ciclo infinito! Questo significa che 0,85 e' un binario periodico (diverso da un periodico decimale in base 10)[1]. Esiste una parte periodica per questi numeri nel calcolo. Guardando alla parte periodica, si puo' vedere che $0,85 = 0,11\overline{0110}_2$. Cosi', $23,85 = 10111,11\overline{0110}_2$.

Una conseguenza importante del calcolo precedente e' che 23,85 non puo' essere rappresentato *esattamente* in binario utilizzando un numero finito di bit. (cosi' come $\frac{1}{3}$ non puo' essere rappresentato in decimale con un numero finito di cifre.) Come mostrera' questo capitolo, le variabili C `float` e `double` vengono memorizzate in binario. Cosi' valori come 23,85 non possono essere memorizzati esattamente in queste variabili, ma solo una loro approssimazione.

Per semplificare l'hardware, i numeri in virgola mobile sono memorizzati in un formato consistente. Questo formato usa la notazione scientifica (ma in binario utilizzando le potenze del 2, non del 10). Per esempio, 23,85 o $10111,11011001100110\ldots_2$ sarebbe memorizzato come:

$$1,011111011001100110\ldots \times 2^{100}$$

(dove l'esponente (100) e' in binario). Un numero in virgola mobile *normalizzato* ha il seguente formato:

$$1,sssssssssssssss \times 2^{eeeeeee}$$

dove $1,sssssssssssss$ e il *significante* e $eeeeeeee$ e l' *esponente*.

6.1.2 Rappresentazione in virgola mobile IEEE

L'IEEE (Institute of Electrical and Electronic Engineers - Istituto di ingegneria elettrica ed elettronica) e' una organizzazione internazionale che ha creato degli specifici formati binari per la memorizzazione dei numeri in virgola mobile. Questo formato e' il piu' utilizzato (ma non l'unico) dai computer prodotti oggigiorno. Spesso e' supportato direttamente dall'hardware del computer stesso. Per esempio, il processore numerico (o matematico) della Intel (che viene montato su tutte le CPU a partire dal Pentium) utilizza questo formato. L'IEEE definisce due diversi formati con differenti precisioni: precisione singola e precisione doppia. In C, la precisione singola e' utilizzata dalla variabili `float` e la precisione doppia da quelle `double`.

Il coprocessore matematico della Intel utilizza anche un terza precisione, piu' elevata, chiamata *precisione estesa*. Infatti, tutti i dati che si trovano nel coprocessore stesso hanno questa precisione. Quando vengono trasferiti dal coprocessore alla memoria, vengono convertiti automaticamente in

[1]Non dovrebbe sorprendere che un numero puo' essere periodico in una base, ma non in'ultra. Pensa a $\frac{1}{3}$, e' periodico in decimale, ma in base 3 (ternario) sarebbe 0.1_3.

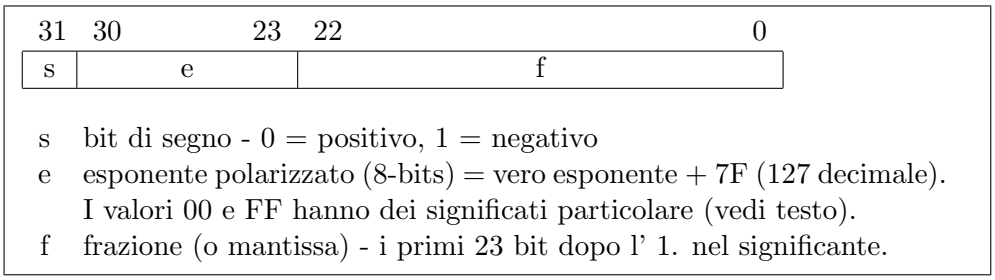


Figura 6.3: precisione singola + IEEE

precisione singola o doppia.[2] La precisione estesa usa formati leggermente diversi rispetto ai formati IEEE per float e double e non verranno discussi in questo testo.

Precisione singola IEEE

La virgola mobile a singola precisione usa 32 bit per codificare il numero. E' di solito accurato a 7 cifre decimali significative. I numeri in virgola mobile sono memorizzati in modo molto piu' complicato rispetto agli interi. La Figura 6.3 mostra il formato base dei numeri in singola precisione IEEE. Ci sono alcune stravaganze con questo formato. I numeri in virgola mobile non usano la rappresentazione in complemento a 2 per i numeri negativi. Utilizzano la rappresentazione "signed magnitude". Come mostrato, il bit 31 determina il segno del numero.

L'esponente binario non e' memorizzato direttamente. Viene invece memorizzata la somma dell'esponente con 7F nei bit dal 23 al 30. Questo *esponente polarizzato* e' sempre non-negativo.

La parte frazionaria (mantissa) presume un significante normalizzato (nella forma $1,sssssssss$). Dal momento che il primo bit e' sempre uno, l'uno di testa *non viene memorizzato!* Questo permette la memorizzazione di un ulteriore bit alla fine e questo incrementa leggermente la precisione. Questa idea e' conosciuta come *rappresentazione dell'uno nascosto*.

Occorre ricordarsi che i byte 41 BE CC CD possono essere interpretati in diversi modi a seconda di cosa il programma fa con loro! Come numero a precisione singola, rappresentano 23,0000381, ma come interi double word, rappresentano 1.103.023.309! La CPU non sa qual'e' la corretta interpretazione!

Come verrebbe memorizzato 23,85? Prima di tutto , e' positivo, quindi il bit di segno e' 0. Poi il vero esponente e' 4, percui l'esponente polarizzato e' $7F + 4 = 83_{16}$. Infine la frazione (mantissa) e' 01111101100110011001100 (ricorda che l'uno di testa e' nascosto). Mettendo tutto insieme (Per chiarezza, il bit di segno e la mantissa sono state sottolineate ed i bit sono stati raggruppati in nibble da 4 bit):

$$\underline{0}\,100\ 0001\ 1\underline{011\ 1110\ 1100\ 1100\ 1100\ 1100}_2 = 41\text{BECCCC}_{16}$$

[2] I tipi `long double` di alcuni compilatori (come il Borland) utilizzano questa precisione estesa. Gli altri compilatori invece usano la doppia precisione sia per i `double` che per i `long double`.(Questo e' permesso dall'ANSI C.)

$e = 0$ e $f = 0$	indica il numero zero (cbe non puo' essere normalizzato) Nota che ci puo' essere +0 e -0.
$e = 0$ e $f \neq 0$	indica un *numero denormalizzato*. Saranno discussi nella sezione successiva.
$e =$ FF e $f = 0$	indica infinito (∞). Esistono infiniti negativi e positivi.
$e =$ FF e $f \neq 0$	indica un risultato indefinito, conosciuto come *NaN* (Not a Number).

Tabella 6.1: Valori speciali di f e e

Questo numero non e' esattamente 23,85 (da momento che e' un periodico binario). Se si esegue la conversione inversa in decimale, si ottiene approssimativamente 23,849998474. Questo numero e' molto vicino a 23,85 ma non e' lo stesso. In realta', in C, 23,85 non verrebbe rappresentato esattamente come sopra. Dal momento che il bit piu' a sinistra che era troncato dalla esatta rappresentazione, e' 1, l'ultimo bit e' arrotondato ad 1. Cosi' 23,85 sarebbe rappresentato come 41 BE CC CD in esadecimale utilizzando la precisione singola. La sua conversione in decimale da 23,850000381 che rappresenta un approssimazione leggermente migliore di 23,85.

Come sarebbe rappresentato -23,85? Basta cambiare il bit del segno: C1 BE CC CD. *Non* si esegue il complemento a 2!

Alcune combinazioni di e e f hanno particolari significati per le virgole mobili IEEE. La Tabella 6.1 descrive questi valori speciali. Un infinito e' prodotto da un overflow o da una divisione per zero. Un risultato indefinito e' prodotto da un operazione non valida come la radice quadrata di un numero negativo, la somma di infiniti, *etc.*

I numeri normalizzati a precisione singola possono andare in magnitudine da 1.0×2^{-126} ($\approx 1.1755 \times 10^{-35}$) a $1.11111\ldots \times 2^{127}$ ($\approx 3.4028 \times 10^{35}$).

Numeri denormalizzati

I numeri denormalizzati possono essere usati per rappresentare numeri con magnitudine troppo piccola per essere normalizzata (*i.e.* sotto 1.0×2^{-126}). Per esempio, consideriamo il numero $1.001_2 \times 2^{-129}$ ($\approx 1.6530 \times 10^{-39}$). Nella forma normalizzata risultante, l'esponente e' troppo piccolo. Puo' essere pero' rappresentato in forma denormalizzata: $0.01001_2 \times 2^{-127}$. Per memorizzare questo numero, l'esponente polarizzato e' impostato a 0 (vedi la Tabella 6.1) e la frazione diventa il significante completo del numero scritto come il prodotto con 2^{-127}(*i.e.* tutti i bit sono memorizzati incluso quello a sinistra della virgola decimale). La rappresentazione di 1.001×2^{-129}

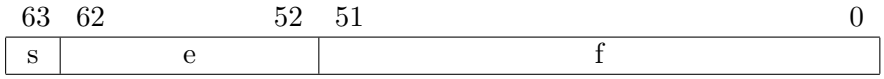


Figura 6.4: doppia precisione IEEE

e' quindi:

$$\underline{0}\,000\ 0000\ 0\,\underline{001\ 0010\ 0000\ 0000\ 0000\ 0000}$$

Doppia precisione IEEE

La doppia precisione IEEE usa 64 bit per rappresentare i numeri e solitamente e' accurato a circa 15 cifre decimali significative. Come mostra la Figura 6.4, il formato base e' molto simile a quelli della precisione singola. Vengono usati piu' bit per l'esponente polarizzato (11) e per la frazione (52) rispetto alla precisione singola.

Il piu' largo spettro per l'esponente polarizzato ha due conseguenze. La prima e' che viene calcolato dalla somma del vero esponente con 3FF (1023)(non 7F come per la precisione singola). Secondariamente e' disponibile uno spettro di veri esponenti piu' ampio (e di conseguenza uno spettro piu' ampio di magnitudine. La magnitudine in doppia precisione puo' andare approssimativamente da 10^{-308} a 10^{308}.

E' il campo della frazione (mantissa) piu' ampio che consente l'incremento del numero di cifre significativo nei valori in doppia precisione.

Come esempio, consideriamo ancora 23,85. L'esponente polarizzato sara' $4 + 3FF = 403$ in esadecimale. La rappresentazione in doppia precisione sara':

$$\underline{0}\,100\ 0000\ 0011\ \underline{0111\ 1101\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1010}$$

o 40 37 D9 99 99 99 99 9A in esadicimale. Se si esegue la conversione inversa in decimale, si ottiene 23,8500000000000014 (ci sono 12 zeri!) che rappresenta una approssimazione sicuramente migliore di 23,85.

La doppia precisione ha gli stessi valori speciali della precisione singola[3]. I numeri denormalizzati sono molto simili. La differenza principale e' che i numeri denormalizzati in doppia precisione usano 2^{-1023} anziche' 2^{-127}.

6.2 Aritmetica in Virgola Mobile

L'aritmetica in virgola mobile su di un computer e' diversa dalla matematica tradizionale. In matematica, tutti i numeri possono essere considerati

[3]La sola differenza e' che per i valori di infinito e risultato indefinito, l'esponente polarizzato e' 7FF e non FF.

esatti. Come mostrato nella sezione precedente, su di un computer molti numeri non possono essere rappresentati esattamente con un numero finito di bit. Tutti i calcoli sono eseguiti con una precisione limitata. Negli esempi di questa sezione, saranno usati per semplicita' i numeri con un significante di 8 bit.

6.2.1 Addizione

Per sommare due numeri in virgola mobile, l'esponente deve essere uguale. Se non sono uguali, devono essere resi uguali spostando il significante del numero con l'esponente piu' piccolo. Per esempio, consideriamo $10.375 + 6.34375 = 16.71875$ o in binario:

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 1.1001011 \times 2^2 \\ \hline \end{array}$$

Questi due numeri non hanno lo stesso esponente cosi' occorre spostare (shift) il significante per rendere gli esponenti uguali e poi fare la somma:

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 0.1100110 \times 2^3 \\ \hline & 10.0001100 \times 2^3 \end{array}$$

Nota che lo shift di $1,1001011 \times 2^2$ toglie l'uno che segue e dopo arrotonda il risultato in $0,1100110 \times 2^3$. Il risultato della somma $10,0001100 \times 2^3$ (o $1,00001100 \times 2^4$) e' uguale a $10000,110_2$ o 16,75. Questo *non* e' uguale alla risposta esatta (16.71875)! E' solo una approssimazione dovuta all'errore di arrotondamento del processo di addizione.

E' importante capire che l'aritmetica in virgola mobile su di un computer (o calcolatore) e' sempre una approssimazione. Le regole matematiche non sempre valgono con i numeri in virgola mobile su di un computer. Le regole matematiche assumono una precisione infinita che nessun computer e' in grado di realizzare. Per esempio, la matematica insegna che $(a + b) - b = a$; Questo invece non e' mai esattamente vero su di un computer!

6.2.2 Sottrazione

Le Sottrazioni lavorano in maniera simile e hanno gli stessi problemi dell' addizione. Come esempio, consideriamo $16.75 - 15.9375 = 0.8125$:

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.1111111 \times 2^3 \\ \hline \end{array}$$

Lo shift di 1.1111111×2^3 da' (approssimato) 1.0000000×2^4

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.0000000 \times 2^4 \\ \hline & 0.0000110 \times 2^4 \end{array}$$

$0.0000110 \times 2^4 = 0.11_2 = 0.75$ che non e' esattamente corretto.

6.2.3 Moltiplicazione e divisione

Per le moltiplicazioni, i significanti sono moltiplicati e gli esponenti sono sommati. Consideriamo $10.375 \times 2.5 = 25.9375$:

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ \times & 1.0100000 \times 2^1 \\ \hline & 10100110 \\ + & 10100110 \\ \hline & 1.10011111000000 \times 2^4 \end{array}$$

Naturalmente, il risultato reale sarebbe arrotondato a 8 bit per ottenere:

$$1.1010000 \times 2^4 = 11010.000_2 = 26$$

Le divisioni sono piu' complicate, ma hanno problemi simili con gli errori di approssimazione.

6.2.4 Ramificazioni per la programmazione

Il punto principale di questa sezione e' che i calcoli in virgola mobile non sono esatti. Il programmatore ha bisogno di tenerne conto. Un errore comune che i programmatori fanno con i numeri in virgola mobile e' comparrarli assumendo che la computazione sia esatta. Per esempio, consideriamo una funzione chiamata f(x) che esegue un calcolo completo e un programma che prova a trovare la radice della funzione[4]. Si potrebbe provare ad usare questo comando per vedere se x e' la radice:

```
if ( f(x) == 0.0 )
```

Ma cosa succede se f(x) ritorna 1×10^{-30}? Cio' indica che x e' una *ottima* approssimazione della vera radice; Pero' l'uguaglianza restituira' false. Non dovrebbe esserci nessun valore in virgola mobile IEEE di x che ritorna esattamente zero, a causa degli errori di arrotondamento di f(x).

Un metodo migliore consiste nell'usare:

```
if ( fabs(f(x)) < EPS )
```

dove EPS e' una macro definita come valore positivo molto piccolo (come 1×10^{-10}). Questa uguaglianza e' vera finche' f(x) e' molto vicino a zero. In generale, per comparare un valore in virgola mobile (diciamo f(x)) ad un'altro (y) si usa:

```
if ( fabs(x - y)/fabs(y) < EPS )
```

[4]La radice di una funzione e' il valore x tale che $f(x) = 0$

6.3 Il Coprocessore Numerico

6.3.1 Hardware

I primi processori Intel non avevano supporto hardware per le operazini in virgola mobile. Cio' significa che dovevano eseguire delle procedure composte da molte istruzioni non in virgola mobile. Per questi primi sistemi, l'Intel forni un chip addizionale chiamato *coprocessore matematico*. Un coprocessore matematico ha delle istruzioni macchina che eseguono molte operazioni in virgola mobile molto piu' velocemente rispetto alle procedure software (sui primi processori, almeno 10 volte piu' velocemente!). Il coprocessore per l'8086/8088 era chiamato 8087. Per l'80286 c'era l'80287 e per l'80386, l'80387. Il processore 80486DX integrava il coprocessore matematica nello stesso 80486.[5] Da Pentium in poi, tutte le generazioni di processori 80x86 hanno un coprocessore matematico integrato. Viene comunque programmato come se fosse una unita' separata. Anche sui primi sistemi senza un coprocessore si puo' installare software che emula un coprocessore matematico. Questi pacchetti di emulazione sono attivati automaticamente quando un programma esegue una istruzione del coprocessore ed eseguono una procedura software che produce lo stesso risultato che si avrebbe col il coprocessore (sebbene molto piu' lentamente, naturalmente).

Il coprocessore numerico ha 8 registri in virgola mobile. Ogni registro tiene 80 bit di dati. In questi registri, i numeri in virgola mobile sono *sempre* memorizzati come numeri a precisione estesa di 80 bit. I registri sono chiamati `ST0`, `ST1`, `ST2`, ... `ST7`. I registri in virgola mobile sono usati diversamente rispetto ai registri per interi della CPU principale. I registri in virgola mobile sono organizzati come uno *stack*. Ricorda che uno stack e' una lista di tipo *LIFO* (Last-In First-Out). `ST0` si riferisce sempre al valore in cima allo stack. Tutti i nuovi numeri sono aggiunti in cima allo stack. I numeri esistenti sono spinti giu' nello stack per fare spazio ai nuovi numeri.

Nel coprocessore numerico esiste anche un registro di stato. Ha diversi flags. Verranno discussi i 4 flag usati nelle comparazione: C_0, C_1, C_2 e C_3. Il loro uso sara' discusso piu' tardi.

6.3.2 Istruzioni

Per distinguere facilmente le normali istruzioni della CPU da quelle del coprocessore, tutti gli mnemonici del coprocessore iniziano con una `F`.

[5] L'80486SX *non* aveva un coprocessore integrato. C'era un chip 80487SX separato per queste macchine.

Caricamento e Memorizzazione

Esistono diverse istruzioni che caricano i dati in cima al registro stack del coprocessore:

`FLD` *sorgente* — carica un numero in virgola mobile dalla memoria in cima allo stack. *sorgente* puo' essere un numero a precisione singola, doppia o estesa oppure un registro del coprocessore.

`FILD` *sorgente* — legge un'*intero* dalla memoria, lo converte in virgola mobile e memorizza il risultato in cima allo stack. La *sorgente* puo' essere una word, una double word o una quad word.

`FLD1` — memorizza un 1 in cima allo stack.

`FLDZ` — memorizza uno 0 in cima alla stack.

Ci sono inoltre diverse istruzioni che memorizzano i dati dallo stack alla memoria. Alcune di queste istruzioni *estraggono* (*i.e.* remuovono) il numero dallo stack come se questo vi fosse memorizzato.

`FST` *dest* — memorizza la cima dello stack (`ST0`) in memoria. La *destinazione* puo' essere un numero a precisione singola o doppia oppure un registro del coprocessore.

`FSTP` *dest* — memorizza la cima dello stack in memoria come `FST`; Inoltre, dopo che il numero e' stato memorizzato, il suo valore viene estratto dallo stack. La *destinazione* puo' essere un numero a precisione singola, doppia o estesa, oppure un registro del coprocessore.

`FIST` *dest* — memorizza il valore in cima allo stack convertito in un intero in memoria. La *destinazione* puo' essere una word o una double word. Lo stack non viene modificato. Il tipo di conversione dipende da alcuni bit nella *parola di controllo* del coprocessore. Questa e' un registro speciale di dimensione word (non in virgola mobile) che controlla come lavora il coprocessore. Come impostazione predefinita, la parola di controllo e' inizializzata in modo da arrotondare all'intero piu' vicino, nella conversione di un'intero. Le istruzioni `FSTCW` (Store Control Word) e `FLDCW` (Load Control Word) possono essere usate per modificare questo comportamento.

`FISTP` *dest* — stesso utilizzo di `FIST` ad eccezione di due cose. La cima dello stack viene estratta e la *destinazione* puo' essere anche una quad word.

Esistono altre due istruzioni che possono muovere o rimuovere i dati dallo stack.

`FXCH ST`*n* — scambia il valore di `ST0` e `ST`*n* nello stack (dove *n* e' il numero di registro da 1 a 7).

`FFREE ST`*n* — libera un registro nello stack marcandolo come inutilizzato o vuoto.

```
1  segment .bss
2  array        resq SIZE
3  sum          resq 1
4
5  segment .text
6        mov    ecx, SIZE
7        mov    esi, array
8        fldz                  ; ST0 = 0
9  lp:
10       fadd   qword [esi]    ; ST0 += *(esi)
11       add    esi, 8         ; muove al double successivo
12       loop   lp
13       fstp   qword sum      ; memorizza il risultato in sum
```

Figura 6.5: Esempio di somma di array

Addizione e sottrazione

Ognuna della istruzioni di addizione calcola la somma di `ST0` e di un'altro operando. Il risultato e' sempre memorizzato in un registro del coprocessore.

`FADD` *src*	`ST0 +=` *src*. La *src* puo' essere qualunque registro del coprocessore oppure un numero in memoria a precisione singola o doppia.
`FADD` *dest*`, ST0`	*dest* `+= ST0`. La *dest* puo' essere qualunque registro del coprocessore.
`FADDP` *dest* o `FADDP` *dest*`, ST0`	*dest* `+= ST0` poi estrae lo stack. La *dest* puo' essere qualunque registro del coprocessore.
`FIADD` *src*	`ST0 += (float)` *src*. somma un'intero a`ST0`. La *src* deve essere una word o una double word in memoria.

Ci sono il doppio delle istruzioni di sottrazione rispetto a quelle di addizione poiche' l'ordine degli operandi e' importante per le sottrazioni (*i.e.* $a + b = b + a$, ma $a - b \neq b - a$!). Per ogni istruzione ne esiste una alternativa che sottrae nell'ordine inverso. Queste istruzioni invertite finiscono tutte per `R` o `RP`. La Figura 6.5 mostra un piccolo pezzo di codice che somma due elementi di un array di double. Alle righe 10 e 13, occorre specificare la dimensione dell'operando di memoria. Altrimenti l'assembler non potrebbe sapere se l'operando di memoria e' un float (dword) o un double (qword).

`FSUB` *src*	`ST0 -=` *src*. La *src* puo' essere un registro del coprocessore o un numero in memoria a precisione singola o doppia.
`FSUBR` *src*	`ST0 =` *src* `- ST0`. La *src* puo' essere un registro del coprocessore o un numero in memoria a precisione singola o doppia.
`FSUB` *dest*`, ST0`	*dest* `-= ST0`. La *dest* puo' essere un registro del coprocessore.
`FSUBR` *dest*`, ST0`	*dest* `= ST0 -` *dest*. La *src* puo' essere un registro del coprocessore.
`FSUBP` *dest* o `FSUBP` *dest*`, ST0`	*dest* `-= ST0` poi estae dallo stack. La *dest* puo' essere un registro del coprocessore.
`FSUBRP` *dest* or `FSUBRP` *dest*`, ST0`	*dest* `= ST0 -` *dest* poi estrae dallo stack. La *dest* puo' essere un registro del coprocessore.
`FISUB` *src*	`ST0 -= (float)` *src*. Sottrae un intero da `ST0`. La *src* deve essere una word o una dword in memoria.
`FISUBR` *src*	`ST0 = (float)` *src* `- ST0`. Sottrae`ST0` da un intero. La *src* deve essere una word o una dword in memoria.

Moltiplicazione e divisione

Le istruzioni di moltiplicazione sono completamente analoghe alle istruzioni di addizione.

`FMUL` *src*	`ST0 *=` *src*. La *src* puo' essere un registro del coprocessore o un numero in memoria a precisione singola o doppia.
`FMUL` *dest*`, ST0`	*dest* `*= ST0`. La *dest* puo' essere un registro del coprocessore.
`FMULP` *dest* o `FMULP` *dest*`, ST0`	*dest* `*= ST0` poi estrae dallo stack. La *dest* puo' essere un registro del coprocessore.
`FIMUL` *src*	`ST0 *= (float)` *src*. Moltiplica un intero per `ST0`. La *src* deve essere una word o una double word in memoria.

Non sorprende che le istruzioni di divizione siano anologhe a quelle di sottrazione. La divisione per Zero restituisce un'infinito.

`FDIV` *src*	`ST0 /=` *src*. La *src* puo' essere un qualunque registro del coprocessore o un numero in memoria a precisione singola o doppia.
`FDIVR` *src*	`ST0 =` *src* `/ ST0`. La *src* puo' essere un qualunque registro del coprocessore o un numero in memoria a precisione singola o doppia.
`FDIV` *dest*`, ST0`	*dest* `/= ST0`. La *dest* puo' essere un qualunque registro del coprocessore.
`FDIVR` *dest*`, ST0`	*dest* `= ST0 /` *dest*. La *dest* puo' essere un qualunque registro del coprocessore.
`FDIVP` *dest* or `FDIVP` *dest*`, ST0`	*dest* `/= ST0` poi estrae lo stack. La *src* puo' essere un qualunque registro del coprocessore.
`FDIVRP` *dest* o `FDIVRP` *dest*`, ST0`	*dest* `= ST0 /` *dest* poi estrae lo stack. La *dest* puo' essere un qualunque registro del coprocessore.
`FIDIV` *src*	`ST0 /= (float)` *src*. Divide `ST0` per un'intero. La *src* deve essere una word o double word in memoria.
`FIDIVR` *src*	`ST0 = (float)` *src* `/ ST0`. Divide un intero per `ST0`. La *src* deve essere una word o double word in memoria.

Comparazioni

Il coprocessore e' anch in grado di eseguire comparazioni tra numeri in virgola mobile. La famiglia delle istruzioni `FCOM` si occuppa di queste operazioni.

`FCOM` *src*	compara `ST0` e *src*. La *src* puo' essere un registro del coprocessore o un float o un double in memoria.
`FCOMP` *src*	compara `ST0` e *src*, poi estrae dallo stack. La *src* puo' essere un registro del coprocessore o un float o un double in memoria.
`FCOMPP`	compara `ST0` e `ST1`, poi estrae dallo stack due volte.
`FICOM` *src*	compara `ST0` e `(float)` *src*. La *src* puo' essere un intero word o dword in memoria.
`FICOMP` *src*	compara `ST0` e `(float)`*src*, poi estrae dallo stack. La *src* puo' essere un intero word o dword in memoria.
`FTST`	compara `ST0` e 0.

Queste istruzioni cambiano i bit C_0, C_1, C_2 e C_3 del registro di stato del coprocessore. Sfortunatamente, non e' possibile per la CPU accedere direttamente a questi bit. Le istruzioni di salto condizionato utilizzano il registro FLAGS, non il registro di stato del coprocessore. Comunque e' relativamente facile spostare i bit della word di stato nei corrispondenti bit del registro FLAGS utilizzando alcune nuove istruzioni:

```
1  ;     if ( x > y )
2  ;
3        fld    qword [x]       ; ST0 = x
4        fcomp  qword [y]       ; compara ST0 e y
5        fstsw  ax              ; sposta C bit in FLAGS
6        sahf
7        jna    else_part       ; se x non e' superiore a y, vai else_part
8  then_part:
9        ; code for then part
10       jmp    end_if
11 else_part:
12       ; code for else part
13 end_if:
```

Figura 6.6: Comparison example

`FSTSW` *dest*	Memorizza la word di stato del coprocessore o in un word in memoria o nel registro AX.
`SAHF`	Memorizza il registro AH nel registro FLAGS.
`LAHF`	Carica il registro AH con i bit del registro FLAGS.

La Figura 6.6 mostra un piccolo esempio di codice. Le righe 5 e 6 trasferiscono i bit C_0, C_1, C_2 e C_3 della word di stato del coprocessore nel registro FLAGS. I bit sono trasferiti in modo da essere analoghi al risultato della comparazione di due interi *unsigned*. Ecco perche' la riga 7 usa una istruzione `JNA`

Il Pentium Pro (e gli altri piu' recenti (Pentium II e III)) supportano due nuovi operatori di comparazione che modificano direttamente il registro FLAGS della CPU.

`FCOMI` *src*	compara `ST0` e *src*. La *src* deve essere un registro del coprocessore.
`FCOMIP` *src*	compara `ST0` e *src*, poi estrae dallo stack. La *src* deve essere un registro del coprocessore.

La Figura 6.7 mostra una sottoroutine di esempio che trova il massimo tra due double utilizzando l'istruzione `FCOMIP`. Non confondere queste istruzioni con le funzioni di comparazione di interi (`FICOM` e `FICOMP`).

Istruzioni Miste

Questa sezione mostra alcune altre istruzioni varie che il coprocessore fornisce.

`FCHS`	`ST0 = - ST0` Cambia il segno di `ST0`
`FABS`	`ST0` $= \lvert$`ST0`$\rvert$ Prende il valore assoluto di `ST0`
`FSQRT`	`ST0` $= \sqrt{\texttt{ST0}}$ Prende la radice quadrata di `ST0`
`FSCALE`	`ST0` $= \texttt{ST0} \times 2^{\lfloor \texttt{ST1} \rfloor}$ moltiplica `ST0` per una potenza del 2 velocemente. `ST1` non e' rimosso dallo stack del coprocessore. La Figura 6.8 mostra un esempio di utilizzo di queste istruzioni.

6.3.3 Esempi

6.3.4 Formula quadratica

Il primo esempio mostra come la formula quadratica puo' essere codificata in assembly. Ricordo che la formula quadratica calcola le soluzioni di una equazione quadratica:

$$ax^2 + bx + c = 0$$

La formula stessa da' due soluzioni per x: x_1 e x_2.

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

L'espressione dentro la radice quadrata ($b^2 - 4ac$) e' chiamata il *discriminante*. Il suo valore e' utile per determinare quali delle tre seguenti possibilita' sono vere per le soluzioni.

1. Esiste una sola soluzione reale degenerata. $b^2 - 4ac = 0$
2. Esistono due soluzioni reali. $b^2 - 4ac > 0$
3. Esistono due soluzioni complesse. $b^2 - 4ac < 0$

Ecco un piccolo programma in C che usa la sottoroutine assembly:

quadt.c

```
#include <stdio.h>

int quadratic( double, double, double, double *, double *);

int main()
{
  double a,b,c, root1, root2;

   printf ("Enter a, b, c: ");
  scanf("%lf %lf %lf", &a, &b, &c);
```

```
  if (quadratic( a, b, c, &root1, &root2) )
    printf ("roots: %.10g %.10g\n", root1, root2);
  else
    printf ("No real roots\n");
  return 0;
}
```

quadt.c

Ecco la routine assembly:

quad.asm

```
; funzione quadratica
; trova le soluzioni dell'equazione quadratica:
;       a*x^2 + b*x + c = 0
; prototipo C:
;   int quadratic( double a, double b, double c,
;                  double * root1, double *root2 )
; Parametri:
;   a, b, c - coefficenti delle potenze dell'equazione quadratica (vedi sopra)
;   root1   - puntatore a double in cui memorizzare la prima radice
;   root2   - puntatore a double in cui memorizzare la seconda radice
; Valore di Ritorno:
;   ritorna 1 se viene trovata una radice reale, altrimenti 0

%define a               qword [ebp+8]
%define b               qword [ebp+16]
%define c               qword [ebp+24]
%define root1           dword [ebp+32]
%define root2           dword [ebp+36]
%define disc            qword [ebp-8]
%define one_over_2a     qword [ebp-16]

segment .data
MinusFour       dw      -4

segment .text
        global  _quadratic
_quadratic:
        push    ebp
        mov     ebp, esp
        sub     esp, 16         ; alloca 2 double (disc & one_over_2a)
        push    ebx             ; devo salvare il valore originale ebx
```

```
32
33          fild    word [MinusFour]; stack -4
34          fld     a               ; stack: a, -4
35          fld     c               ; stack: c, a, -4
36          fmulp   st1             ; stack: a*c, -4
37          fmulp   st1             ; stack: -4*a*c
38          fld     b
39          fld     b               ; stack: b, b, -4*a*c
40          fmulp   st1             ; stack: b*b, -4*a*c
41          faddp   st1             ; stack: b*b - 4*a*c
42          ftst                    ; test with 0
43          fstsw   ax
44          sahf
45          jb      no_real_solutions ; se disc < 0, nessuna soluzione reale
46          fsqrt                   ; stack: sqrt(b*b - 4*a*c)
47          fstp    disc            ; memorizza ed estrae dallo stack
48          fld1                    ; stack: 1.0
49          fld     a               ; stack: a, 1.0
50          fscale                  ; stack: a * 2^(1.0) = 2*a, 1
51          fdivp   st1             ; stack: 1/(2*a)
52          fst     one_over_2a     ; stack: 1/(2*a)
53          fld     b               ; stack: b, 1/(2*a)
54          fld     disc            ; stack: disc, b, 1/(2*a)
55          fsubrp  st1             ; stack: disc - b, 1/(2*a)
56          fmulp   st1             ; stack: (-b + disc)/(2*a)
57          mov     ebx, root1
58          fstp    qword [ebx]     ; store in *root1
59          fld     b               ; stack: b
60          fld     disc            ; stack: disc, b
61          fchs                    ; stack: -disc, b
62          fsubrp  st1             ; stack: -disc - b
63          fmul    one_over_2a     ; stack: (-b - disc)/(2*a)
64          mov     ebx, root2
65          fstp    qword [ebx]     ; store in *root2
66          mov     eax, 1          ; ritorna 1
67          jmp     short quit
68
69  no_real_solutions:
70          mov     eax, 0          ; ritorna 0
71
72  quit:
73          pop     ebx
```

```
74          mov     esp, ebp
75          pop     ebp
76          ret
```
quad.asm

6.3.5 Leggere array da file

In questo esempio, una routine assembly legge le double da un file. Ecco un piccolo programma di test in C:

readt.c

```c
/*
 * Questo programma testa la procedura assembly a 32 bit read_doubles().
 * Legge i double dalla stdin. (Usa la redirezione per leggere dal file.)
 */
#include <stdio.h>
extern int read_doubles( FILE *, double *, int );
#define MAX 100

int main()
{
  int i,n;
  double a[MAX];

  n = read_doubles(stdin, a, MAX);

  for( i=0; i < n; i++ )
    printf("%3d %g\n", i, a[i]);
  return 0;
}
```

readt.c

Ecco la routine assembly

read.asm

```
1  segment .data
2  format  db      "%lf", 0        ; format for fscanf()
3
4  segment .text
5          global  _read_doubles
6          extern  _fscanf
7
```

```
8  %define SIZEOF_DOUBLE   8
9  %define FP              dword [ebp + 8]
10 %define ARRAYP          dword [ebp + 12]
11 %define ARRAY_SIZE      dword [ebp + 16]
12 %define TEMP_DOUBLE     [ebp - 8]
13
14 ;
15 ; funzione _read_doubles
16 ; prototipo C:
17 ;   int read_doubles( FILE * fp, double * arrayp, int array_size );
18 ; Questa funzione legge i double da un file di testo in un array
19 ; fino a EOF o finche' l'array non e' pieno.
20 ; Parametri:
21 ;   fp         - puntatore FILE per leggere da (deve essere aperto per l'input)
22 ;   arrayp     - puntatore all'array di double in cui copiare
23 ;   array_size - numero di elementi dell'array
24 ; Valore di ritorno:
25 ;   numero di double memorizzati nell'array (in EAX)
26
27 _read_doubles:
28         push    ebp
29         mov     ebp,esp
30         sub     esp, SIZEOF_DOUBLE      ; definisce un double nello stack
31
32         push    esi                     ; salva esi
33         mov     esi, ARRAYP             ; esi = ARRAYP
34         xor     edx, edx                ; edx = indice dell'array (inizialmente 0)
35
36 while_loop:
37         cmp     edx, ARRAY_SIZE         ; Se edx < ARRAY_SIZE?
38         jnl     short quit              ; se no, esce dal ciclo
39 ;
40 ; chiama fscanf() per leggere un double in TEMP_DOUBLE
41 ; fscanf() puo' cambiare edx, cosi' occorre salvarlo
42 ;
43         push    edx                     ; salva edx
44         lea     eax, TEMP_DOUBLE
45         push    eax                     ; mette &TEMP_DOUBLE
46         push    dword format            ; mette &format
47         push    FP                      ; mette il puntatore al file
48         call    _fscanf
49         add     esp, 12
```

```
50          pop     edx                     ; ripristina edx
51          cmp     eax, 1                  ; fscanf ha ritornato 1?
52          jne     short quit              ; Se no, esci dal ciclo.
53
54  ;
55  ; copia TEMP_DOUBLE in ARRAYP[edx]
56  ; (Gli 8 bit del double sono copiati in du copie da 4 bit)
57  ;
58          mov     eax, [ebp - 8]
59          mov     [esi + 8*edx], eax      ; prima copia i 4 byte piu' bassi
60          mov     eax, [ebp - 4]
61          mov     [esi + 8*edx + 4], eax  ; poi copia i 4 byte piu' alti
62
63          inc     edx
64          jmp     while_loop
65
66  quit:
67
68          pop     esi                     ; ripristina esi
69
70          mov     eax, edx                ; memorizza il valore di ritorno in eax
71
72          mov     esp, ebp
73          pop     ebp
74          ret
```

read.asm

6.3.6 Ricerca di numeri primi

Questo esempio finale si occupa nuovamente di trovare i numeri primi. Questa implementazione e' pero' piu' efficente rispetto a quella precedente. Questa memorizza i numeri primi in un'array e divide solo per i precendenti numeri primi che ha trovato invece di ogni numero dispari per trovare nuovi numeri primi.

Un'altra differenza e' che calcola la radice quadrata del valore possibile per il successivo numero primo, per determinare a che punto puo' terminare la ricerca dei fattori. Altera la parola di controlo del coprocessore in maniera che quando memorizza la radice quadrata come un intero, tronca invece di arrotondare. Questo processo viene controllato dai bit 10 e 11 della parola di controllo. Questi bit sono chiamati bit RC (Rounding control). Se sono entrambi a 0 (di default) , il coprocessore tronca le conversioni degli interi. Nota che la routine sta attenta a salvare il valore originale della parola di controllo e lo ripristina prima di ritorna il controllo al chiamante.

Ecco il programma driver in C:

fprime.c

```
#include <stdio.h>
#include <stdlib.h>
/*
 * funzione find_primes
 * trova un numero stabilito di numeri primi
 * Parametri:
 *   a - array che contiene i numeri primi
 *   n - quanti numeri primi trovare
 */
extern void find_primes( int * a, unsigned n );

int main()
{
  int status;
  unsigned i;
  unsigned max;
  int * a;

  printf("How many primes do you wish to find? ");
  scanf("%u", &max);

  a = calloc( sizeof(int), max);

  if ( a ) {

    find_primes(a,max);

    /* stampa gli ultimi 20 numeri primi trovati */
    for(i= ( max > 20 ) ? max - 20 : 0; i < max; i++ )
      printf("%3d %d\n", i+1, a[i]);

    free(a);
    status = 0;
  }
  else {
    fprintf( stderr, "Can not create array of %u ints\n", max);
    status = 1;
  }
```

```
  return status;
}
```

fprime.c

Ecco la routine assembly:

prime2.asm

```
segment .text
        global  _find_primes
;
; funzione find_primes
; trova un numero stabilito di numeri primi
; Parametri:
;   array - array che contiene i numeri primi
;   n_find - quanti numeri primi trovare
; Prototipo C:
;extern void find_primes( int * array, unsigned n_find )
;
%define array         ebp + 8
%define n_find        ebp + 12
%define n             ebp - 4           ; number of primes found so far
%define isqrt         ebp - 8           ; floor of sqrt of guess
%define orig_cntl_wd  ebp - 10          ; original control word
%define new_cntl_wd   ebp - 12          ; new control word

_find_primes:
        enter   12,0                    ; fa spazio per le variabili locali

        push    ebx                     ; salva le possibile variabili registro
        push    esi

        fstcw   word [orig_cntl_wd]     ; ottiene la parola di controllo attuale
        mov     ax, [orig_cntl_wd]
        or      ax, 0C00h               ; imposta i bit di arrotondamento a 11 (tronca
        mov     [new_cntl_wd], ax
        fldcw   word [new_cntl_wd]

        mov     esi, [array]            ; esi punta all' array
        mov     dword [esi], 2          ; array[0] = 2
        mov     dword [esi + 4], 3      ; array[1] = 3
        mov     ebx, 5                  ; ebx = guess = 5
        mov     dword [n], 2            ; n = 2
```

```
36  ;
37  ; Questo ciclo esterno trova un numero primo ad ogni iterazione, che somma
38  ; alla fine dell'array. Diversamente dalla prima versione del problema, questa
39  ; funzione non determina la primitivita' dividendo per tutti i numeri dispari
40  ; Semplicemnte divide per i numeri primi che sono gia' stati trovati (E' per
41  ; questo che vengono memorizzati in un array.)
42  ;
43  while_limit:
44          mov     eax, [n]
45          cmp     eax, [n_find]           ; while ( n < n_find )
46          jnb     short quit_limit
47
48          mov     ecx, 1                  ; ecx e' utilizzato come indice dell'array
49          push    ebx                     ; memorizza il possibile primo nello stack
50          fild    dword [esp]             ; carica il possibile primo nello stack
51                                          ; del coprocessore
52          pop     ebx                     ; toglie il possibile valore dallo stack
53          fsqrt                           ; trova la radice quadrata di guess
54          fistp   dword [isqrt]           ; isqrt = floor(sqrt(quess))
55  ;
56  ; Questo ciclo interno divide il possibile primo (ebx) per i numeri
57  ; primi trovati finora finche' non trova  un suo fattore primo ( che
58  ; significa che il possibile primo non e' primo) o finche' il numero
59  ; primo da dividere e' maggiore di floor(sqrt(guess))
60  ;
61  while_factor:
62          mov     eax, dword [esi + 4*ecx]        ; eax = array[ecx]
63          cmp     eax, [isqrt]                    ; while ( isqrt < array[ecx]
64          jnbe    short quit_factor_prime
65          mov     eax, ebx
66          xor     edx, edx
67          div     dword [esi + 4*ecx]
68          or      edx, edx                        ; && guess % array[ecx] != 0 )
69          jz      short quit_factor_not_prime
70          inc     ecx                             ; prova il numero primo successivo
71          jmp     short while_factor
72
73  ;
74  ; trovato un nuovo numero primo !
75  ;
76  quit_factor_prime:
77          mov     eax, [n]
```

```
78          mov     dword [esi + 4*eax], ebx        ; aggiunge il possibile primo in fondo
79
80          inc     eax
81          mov     [n], eax                        ; inc n
82
83  quit_factor_not_prime:
84          add     ebx, 2                          ; prova il successivo numero dispari
85          jmp     short while_limit
86
87  quit_limit:
88
89          fldcw   word [orig_cntl_wd]             ; ripristina la parola di controllo
90          pop     ebx
91
92          leave
93          ret
```

prime2.asm

```
1  global _dmax
2
3  segment .text
4  ; funzione _dmax
5  ; ritorna il piu' grande  fra i suoi due argomenti double
6  ; prototipo C
7  ; double dmax( double d1, double d2 )
8  ; Parametri:
9  ;   d1   - primo double
10 ;   d2   - secondo double
11 ; Valore di ritorno:
12 ;   Il piu' grande fra d1 e d2 (in ST0)
13 %define d1   ebp+8
14 %define d2   ebp+16
15 _dmax:
16         enter   0, 0
17
18         fld     qword [d2]
19         fld     qword [d1]          ; ST0 = d1, ST1 = d2
20         fcomip  st1                 ; ST0 = d2
21         jna     short d2_bigger
22         fcomp   st0                 ; estrae d2 dallo stack
23         fld     qword [d1]          ; ST0 = d1
24         jmp     short exit
25 d2_bigger:                          ; se d2 e' il massimo, niente
26                                     ; da fare
27 exit:
28         leave
29         ret
```

Figura 6.7: Esempio di `FCOMIP`

```
1  segment .data
2  x            dq  2.75          ; converte in formato double
3  five         dw  5
4
5  segment .text
6        fild   dword [five]      ; ST0 = 5
7        fld    qword [x]         ; ST0 = 2.75, ST1 = 5
8        fscale                   ; ST0 = 2.75 * 32, ST1 = 5
```

Figura 6.8: Esempio di `FSCALE`

Capitolo 7

Strutture e C++

7.1 Strutture

7.1.1 Introduzione

Le strutture sono usate in C per raggruppare in una variabile composita dati correlati. Questa tecnica ha diversi vantaggi:

1. Rende piu' pulito il codice mostrando che i dati che sono raggruppati in una struttura sono intimamente correlati.

2. Semplifica il passaggio di dati alle funzioni. Invece di passare variabili multiple separatamente, possono essere passate come una singola unita'.

3. Incrementa la *localita'*[1] del codice.

Dal punto di vista dell'assembly, una struttura puo' essere considerata come un array di elementi con *varie* dimensioni. Gli elementi di un array reale sono sempre della stessa dimensione e dello stesso tipo. Questa proprieta' permette di calcolare l'indirizzo di ogni elemento conoscendo l'indirizzo di parteza dell'array, la dimensione dell'elemento e l'indice dell'elemento desiderato.

Gli elementi di una struttura non devono essere della stessa dimensione (e solitamente non lo sono). Percio', ogni elemento della struttura deve essere esplicitamente specificato e gli viene dato un *tag* (o nome) invece di un indice numerico.

In assembly, l'elemento di una struttura verra' acceduto in modo simile agli elementi di un array. Per accedere ad un elemento occorre conoscere l'indirizzo di partenza della struttura e l'*offset relativo* di quell'elemento

[1] Vedi la sezione sul controllo della memoria virtuale di un manuale di qualunque sistema operativo per la discussione di questo termine.

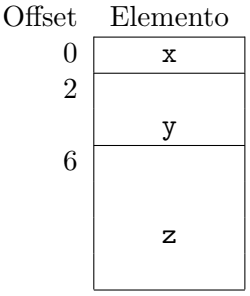


Figura 7.1: Struttura S

dall'inizio della struttura. Mentre negli array questo offset poteva essere calcolato dall'indice dell'elemento, in una struttura e' il compilatore che assegna un offset all'elemento.

Per esempio, consideriamo la seguente struttura:

```
struct S {
  short int x;    /* intero 2-byte */
  int       y;    /* intero 4-byte */
  double    z;    /* float  8-byte */
};
```

La Figura 7.1 mostra come una variabile di tipo `S` apparirebbe nella memoria del computer. Lo standard ANSI C definisce che gli elementi di una struttura siano disposti in memoria con lo stesso ordine in cui sono definiti nella definizione della `struct`. Definisce anche che il primo elemento si trova all'inizio della struttura (*i.e.* a offset zero). Definisce inoltre una utile macro nel file di header `stddef.h`, chiamata `offsetof()`. Questa macro calcola e restituisce l'offset di qualunque elemento di una struttura. Questa macro accetta due parametri, il primo e' il nome del *tipo* della struttura, il secondo e' il nome dell'elemento di cui si vuole trovare l'offset. Il risultato di `offsetof(S, y)` in base alla Figura 7.1 sarebbe 2.

7.1.2 Allineamento di Memoria

Ricordo che un indirizzo e' al limite della double word , se e' divisibile per 4

Se utilizziamo la macro `offsetof` per trovare l'offset di `y` utilizzando il compilatore *gcc*, vediamo che questa restituisce 4 anziche' 2. Perche? Perche' *gcc* (e molti altri compilatori) allineano di default le variabili a sezioni di double word. In modalita' protetta a 32 bit la CPU riesce a leggere piu' velocemente i dati se questi iniziano al limite della double word. La Figura 7.2 mostra come la struttura `S` appare realmente utilizzando *gcc*. Il compilatore inserisce due byte non usati nella struttura per allineare `y` (e `z`) al limite della double word. Questo dimostra come sia una buona idea usare

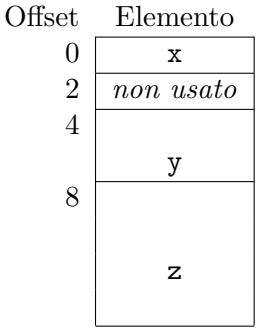


Figura 7.2: Struttura S

`offsetof` per calcolare gli offset invece di calcolarli a mano quando si usano strutture definite in C.

Naturalmente, se la struttura e' usata sono in assembly, il programmatore puo' determinare da solo gli offset. Se si interfacciano il C e l' assembly, e' molto importante che sia il codice C che quello assembly concordino su gli offset degli elementi della struttura! Una complicazione e' data dal fatto che i diversi compilatori possono dare diversi offset agli elementi. Per esempio, come abbiamo visto, il compilatore *gcc* crea una struttura `S` come quella in Figura 7.2; Il compilatore Borland invece creera' una struttura come quella in Figura 7.1. I compilatori C forniscono dei metodi per specificare l'allineamento usato per i dati. Pero' l'ANSI C non specifica come questo debba essere fatto e cosi' ogni compilatore usa una propria strada.

Il compilatore *gcc* ha un sistema flessibile ma complicato per specificare l'allineamento. Il compilatore permette di specificare qualsiasi tipo di allineamento utilizzando una sintassi speciale. Per esempio, la riga seguente:

```
typedef short int unaligned_int __attribute__ (( aligned (1)));
```

definisce un nuovo tipo chiamato `unaligned_int` che e' allieneato al limite del byte. (Si', tutte le parentesi dopo `__attribute__` sono necessarie!) L'1 nel parametro `aligned` puo' essere sostituito con qualsiasi altra potenza del due per specificare altri allineamenti. (2 per allineamento a word, 4 a double word, *etc.*) Se il tipo dell'elemento `y` della struttura fosse cambiato in `unaligned_int`, *gcc* porrebbe `y` a offset 2. `z` avrebbe invece lo stesso offset (8) dal momento che i double sono anche di default allineati in doppia word. La definizione del tipo di `z` dovrebbe essere modificata per poterlo mettere a offset 6.

Il compilatore *gcc* permette anche di *impacchettare* una struttura. Cio' significa che il compilatore utilizzera' il minimo spazio possibile per la struttura. La Figura 7.3 mostra come la struttura `S` possa essere riscritta in questo modo. Questa forma di `S` usera' meno byte possibili, 14.

```
struct S {
  short int x;     /* intero 2-byte */
  int       y;     /* intero 4-byte */
  double    z;     /* float 8-byte */
} __attribute__((packed));
```

Figura 7.3: Struttura impacchettata usando *gcc*

```
#pragma pack(push) /* salva lo stato di allinemento */
#pragma pack(1)    /* imposta l'allineamento ad un byte */

struct S {
  short int x;     /* 2-byte integer */
  int       y;     /* 4-byte integer */
  double    z;     /* 8-byte float   */
};

#pragma pack(pop)  /* ripristina l'allineamento originale */
```

Figura 7.4: Strutture pacchettate con Microsoft e Borland

I compilatori Microsoft e Borland supportano entrambi lo stesso metodo per specificare l'allineamento utilizzando la direttiva `#pragma`.

```
#pragma pack(1)
```

La direttiva qua sopra dice al compilatore di impacchettare gli elementi della struttura al limite di un byte (*i.e.*, senza nessuna aggiunta). L'1 puo' essere sostituito da 2,4,8 o 16 per specificare l'allianeamento rispettivamente al limite di una word, di una double word, di una quad word o di un paragrafo. La direttiva e' valida finche' non e' sostituita da un'altra direttiva. Questo puo' causare problemi dal momento che queste direttive sono spesso usate nei file di header. Se il file di header e' incluso prima di altri file header che contengano delle strutture, queste strutture potrebbero essere dislocate diversamente dalla loro impostazione predefinita. Tutto cio' puo' portare ad errori difficili da scoprire. I diversi moduli di un programma potrebbero disporre gli elementi delle strutture in posti *diversi*!

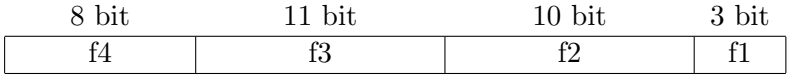
Esiste un modo per evitare questo problema. Microsoft e Borland supportano un metodo per salvare lo stato dell'allineamento corrente e ripristinarlo in seguito. La Figura 7.4 mostra come cio' viene realizzato.

```
struct S {
  unsigned f1 : 3;    /* campo a 3-bit */
  unsigned f2 : 10;   /* campo a 10-bit */
  unsigned f3 : 11;   /* campo a 11-bit */
  unsigned f4 : 8;    /* campo a 8-bit */
};
```

Figura 7.5: Esempio di Campi di bit

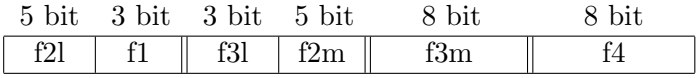
7.1.3 Campi di Bit

I campi di bit permettono di specifica il numero di bit che potranno essere utilizzati dai membri di una struttura. La dimensione in bit non deve essere un multiplo di 8. Un campo di bit membro e' definito come un membro **unsigned int** o un membro **int** con due punti e il numero di bit dopo di questi. La Figura 7.5 mostra un'esempio. Questo definisce una variabile a 32 bit che viene suddivisa in questo modo:



Il primo campo di bit e' assegnato ai bit meno significativi della sua double word.[2]

Il formato, pero', non e' cosi' semplice se guardiamo a come sono diposti realmente i bit in memoria. La difficolta' sorge quando i campi di bit sono a cavallo di limiti di byte. La ragione e' che i byte nei processori little endian sono inverti in memoria. Per esempio, i campi di bit della struttura `S` saranno disposti in memoria in questo modo:



L'etichetta *f2l* si riferisce agli ultimi 5 bit (*i.e.*, i 5 bit meno significativi) del campo di bit *f2*. L'etichetta *f2m* si riferisce invece ai 5 bit piu' significativi di *f2*. Le doppie linee verticali indicano i limiti di un byte. Se vengono invertiti tutti i byte, le sezioni dei campi *f2* e *f3* verranno riunite nelle posizioni corrette.

La disposizione della memoria fisica generalmente non e' importante a meno che i dati non vengano trasferiti dal o al programma (cio' e' abbastanza comune con i campi di bit). E' comune che le interfaccie dei dispositivi Hardware usino numeri dispari di bit che e' possibile rappresentare con i campi di bit.

[2] In realta', lo standard ANSI/ISO C da' ai compilatori una certa flessibilita' su come esattamente devono essere disposti i bit. I compilatori C piu' comuni (*gcc*, *Microsoft* e *Borland*) comunque utilizzano questo metodo.

Byte \ Bit	7	6	5	4	3	2	1	0
0	Codice Operativo (08h)							
1	Unita' Logica #			msb di LBA				
2	mezzo del Logical Block Address							
3	lsb del Logicial Block Address							
4	Lunghezza di Trasferimento							
5	Controllo							

Figura 7.6: Formato del Comando di Lettura SCSI

Un esempio e' SCSI[3]. Un comando diretto di lettura per una periferica SCSI viene specificato mandando un messaggio a 6 byte alla periferica nel formato specificato in Figura 7.6. La difficolta della sua rappresentazione utilizzando i campi di bit sta nel *logical block address* che occupa tre byte del comando. La Figura 7.6 mostra che i dati sono memorizzati nel formato big endian. La Figura 7.7 invece mostra una definizione possibilmente portabile per tutti i compilatori. Le prime due righe definiscono una macro che e' vera se il codice e' compilato con un compilatore Microsoft o Borland. La parte potenzialmente piu' spiazzante sono le righe dalla 11 alla 14. Innanzitutto ci si potrebbe chiedere perche' i campi lba_mid e lba_lsb sono definiti separatamente e non come un singolo campo a 16 bit? La ragione e' che i dati sono in ordine big endian. Un campo a 16 bit verrebbe memorizzato in ordine little endian dal compilatore. Inoltre, i campi lba_msb e logical_unit sembrano essere invertiti. In realta' non e' cosi'. Devono essere messi in questo ordine. La Figura 7.8 mostra come i campi di bit sono mappati in un entita' a 48 bit. (Il limite di byte e' ancora individuato dalle doppie linee.) Quando vengono allocati in memoria in ordine little endian, i bit sono disposti nel formato desiderato (Figura 7.6).

A complicare ulteriormente le cose, la definizione di SCSI_read_cmd non funziona correttamente nel C Microsoft. Se viene valutata l'espressione **sizeof**(SCSI_read_cmd), il C di Microsoft restituira' 8 e non 6! Questo perche' il compilatore Microsoft usa il tipo dei campi di bit per determinare il mappaggio dei bit. Dal momento che tutti i campi di bit sono definiti come tipi **unsigned** il compilatore aggiunge due byte alla fine della struttura per ottenere un numero intero di double word. Cio' puo' essere rimediato assegnando a tutti i campi il tipo **unsigned short** . Cosi' il compilatore Microsoft non ha bisogno di aggiungere nessun byte dal momento che 6 byte e' un numero intero di word a 2 byte.[4] Anche gli altri compilatori funzionano correttamente con questa modifica. La Figura 7.9 mostra appunto un'al-

[3]Small Computer Systems Interface, uno standard industriale per gli Hard disk

[4]Mescolare diversi tipi di campi di bit porta ad un comportamento molto distorto! Il lettore e' invitato a sperimentarlo.

```
#define MS_OR_BORLAND (defined(__BORLANDC__) \
                       || defined(_MSC_VER))

#if MS_OR_BORLAND
# pragma pack(push)
# pragma pack(1)
#endif

struct SCSI_read_cmd {
  unsigned opcode : 8;
  unsigned lba_msb : 5;
  unsigned logical_unit : 3;
  unsigned lba_mid : 8;     /* middle bits */
  unsigned lba_lsb : 8;
  unsigned transfer_length : 8;
  unsigned control : 8;
}
#if defined(__GNUC__)
   __attribute__((packed))
#endif
;

#if MS_OR_BORLAND
# pragma pack(pop)
#endif
```

Figura 7.7: Struttura del Formato del Comando di Lettura SCSI

tra definizione che funziona correttamente con tutti e tre i compilatori. I questa definizione non vengono usati campi di bit di tipo **unsigned char** ad eccezione di due.

Il lettore non dovrebbe farsi scoraggiare se ha trovato la discussione precedente molto confusa. E' confusa! L'autore ritiene che spesso sia meno complesso evitare i campi di bit ed usare le operazioni sui bit per esaminare e modificare i bit manualmente.

7.1.4 Utilizzare le strutture in assembly

Come discusso prima, accedere ad una struttura in assembly e' molto simile all'accesso ad un'array. Come semplice esempio, poniamo il caso di voler scrivere una routine che azzeri l'elemento `y` della struttura `S`. Assumiamo che il prototipo della routine sia:

8 bit	8 bit	8 bit	8 bit	3 bit	5 bit	8 bit
controllo	lunghezza di transf	lba_lsb	lba_mid	unita' logica	lba_msb	opcode

Figura 7.8: Mappaggio dei campi del `SCSI_read_cmd`

```
struct SCSI_read_cmd {
  unsigned char opcode;
  unsigned char lba_msb : 5;
  unsigned char  logical_unit  : 3;
  unsigned char lba_mid;     /* middle bits */
  unsigned char lba_lsb;
  unsigned char  transfer_length ;
  unsigned char control;
}
#if defined(__GNUC__)
    __attribute__((packed))
#endif
;
```

Figura 7.9: Struttura alternativa del Formato del Comando di Lettura SCSI

void zero_y(S * s_p);

La routine assembly sarebbe:

```
1  %define      y_offset  4
2  _zero_y:
3        enter  0,0
4        mov    eax, [ebp + 8]      ; ottine s_p (puntatore alla struct) dallo stack
5        mov    dword [eax + y_offset], 0
6        leave
7        ret
```

Il C permette di passare una struttura per valore ad una funzione; E' comunque quasi sempre una cattiva idea. Nel passaggio per valore, tutti i dati della struttura devono essere copiati nello stack e poi recuperati dalla routine. E' molto piu' efficente passare invece un puntatore alla struttura.

Il C inoltre permette di usare un tipo struttura come valore di ritorno di una funzione. Ovviamente la struttura non puo' essere ritornata nel registro `EAX`. Ogni compilatore gestisce la situazione diversamente. Una soluzione comune che utilizzano i compilatori e' quella di riscrivere internamente la funzione come se accettasse come parametro un puntatore alla struttura.

```
#include <stdio.h>

void f( int x )
{
   printf ("%d\n", x);
}

void f( double x )
{
   printf ("%g\n", x);
}
```

Figura 7.10: Due funzioni `f()`

Il puntatore e' utilizzato per mettere il valore di ritorno in una struttura definita fuori dalla routine chiamata.

La maggior parte dei compilatori (incluso NASM) forniscono un supporto integrato per definire le strutture nel codice asssembly. Consulta la tua documentazione per maggiori dettagli.

7.2 Assembly e C++

Il linguaggio di programmazione C++ e' una estensione del linguaggio C. Molte delle regole di base per l'interfacciamento tra assembly e C si applicano anche al C++. Altre regole invece devono essere modificate. Inoltre alcune delle estensioni del C++ sono piu' facili da capire con la conoscenza del linguaggio assembly. Questa sezione richiede una conoscenza di base del C++.

7.2.1 Overloading e Name Mangling

Il C++ permette di definire diverse funzioni (o funzioni membro di classe) con lo stesso nome. Quando piu' di una funzione condivide lo stesso nome, si dice che le funzioni sono *sovraccaricate*. In C, se due funzioni sono definite con lo stesso nome, il linker produrra' un'errore poiche' trovera'due definizioni per lo stesso simbolo nel file oggetto che sta collegando. Per esempio, consideriamo il codice in Figura 7.10. L'equivalente codice assembly dovrebbe definire due etichette chiamate `_f`, il che ovviamente produrrebbe un'errore.

Il C++ usa lo stesso processo di collegamento del C, ma evita questo

errore applicando il *name mangling*[5] o modificando il simbolo usato per etichettare la funzione. In un certo modo anche il C usa il name mangling. Aggiunge un' underscore (_) al nome della funzione C quando crea un etichetta per questa. Il C pero' , "storpiera"' il nome di entrambe le funzioni in Figura 7.10 nello stesso modo e produrra' un'errore. Il C++ usa invece un processo di "storpiatura" piu' sofisticato che produce due diverse etichette per le funzioni. Per esempio, il DJGPP assegnerebbe alla prima funzione in Figura 7.10 l'etichetta `_f__Fi` e alla seconda funzione l'etichetta `_f__Fd`. Questo evita errori di collegamento.

Sfortunatamente non esiste uno standard di come maneggiare i nomi in C++ e i diversi compilatori modificano i nomi differentemente. Per esempio, il Borland C++ utilizzerebbe le etichette `@f$qi` e `@f$qd` per le due funzioni in Figura 7.10. Le regole non sono comunque completamente arbitrarie. Il nome modificato codifica la *firma* di una funzione. La firma di una funzione e' data dall'ordine e dal tipo dei suoi parametri. Nota che la funzione che accetta un solo argomento di tipo `int` ha una *i* al termine del suo nome modificato (sia per DJGPP e Borland) e che quella che accetta un argomento di tipo `double` ha invece una *f* alla fine del nome modificato. Se ci fosse stata una funzione chiamata `f` con questo prototipo:

```
void f( int x, int y, double z);
```

DJGPP avrebbe modificato il suo nome in `_f__Fiid` e Borland in `@f$qiid`.

Il valore di ritorno della funzione *non* e' parte della firma della funzione e non e' codificato nel suo nome modificato. Questo spiega una regola dell'overloadig in C++. Solo le funzioni, la cui firma e' unica possono essere sovraccaricate. Come si puo' vedere, se vengono definite in C++ due funzioni con lo stesso nome e con la stessa firma, queste produrrano lo stesso nome modificato e genereranno un'errore nel collegamento. Di default, tutte le funzioni in C++ hanno il nome modificato, anche quelle che non sono sovraccaricate. Quando si compila un file, il compilatore non ha modo di sapere se una particolare funzione e' sovraccaricata o meno, cosi' modifica tutti i nomi. Infatti modifica anche i nomi delle variabili globali codificanto il tipo della variabile in modo simile alla firma delle funzioni. Cosi' se viene creata una variabile globale di un certo tipo in un file, e si prova ad utilizzarla in un'altro file col tipo sbagliato, verra' generato un errore di collegamento. Questa caratteristica del C++ e' conosciuta come *collegamento sicuro di tipo*. Inoltre questo genera anche un'altro tipo di errori, i prototipi inconsistenti. Questi errori nascono quando la definizione di una funzione in un modulo non concorda con il prototipo usato in un'altro modulo. In C, questi possono essere errori molto difficili da trovare. Il C non controlla

[5]Letteralmente " storpiatura del nome", tecnica conosciuta anche come *name-decoration*(ndt).

questi errori. Il programma viene compilato e collegato senza problemi, ma avra' un conportamento imprevedibile quando il codice chiamante mettera' nello stack tipi di parametro diversi da quelli che la funzione si aspetta. In C++, verra' generato un'errore di collegamento.

Quando il compilatore C++ analizza una chiamata di funzione, cerca la funzione chiamata in base al tipo di argomenti passati alla funzione stessa[6]. Se trova una corrispondenza, allora crea una `CALL` alla funzione corretta utilizzando le regole di name mangling del compilatore.

Dal momento che diversi compilatori usano diverse regole di name mangling, il codice C++ compilato da diversi compilatori potrebbe non essere collegabile insieme. Questo fatto e' importante quando si ha in mente di utilizzare librerie C++ precompilate! Se si vuole scrivere una funzione in assembly che venga usata da codice C++, occorre conoscere le regole di name mangling per il compilatore che verra' usato (o si usera' la tecnica spiegata sotto).

Lo studente scaltro potrebbe chiedere se il codice in Figura 7.10 funzionera' come ci si aspetta. Da momento che il C++ modifichera' i nomi di tutte le funzioni, la funzione `printf` sara' storpiata ed il compilatore non produrra una `CALL` alla etichetta `_printf`. Questo e' un valido riferimento! Se il prototipo di `_printf` fosse semplicemente posto in cima al file, succederebbe questo. Il prototipo e':

```
int printf( const char *, ...);
```

DJGPP lo modificherebbe in `_printf__FPCce`. (La `F` sta per *funzione*, `P` per *puntatore*, `C` per *const*, `c` per *char* e `e` per ellissi.) Non sarebbe una chiamata alla funzione `printf` delle normale libreria C! Naturalmente, ci deve essere un modo per il codice C++ di chiamare il codice C. Questo e' molto importante perche' esiste *moltissimo* vecchio ma utile codice C in giro. In aggiunta alla possibilita' di chiamare correttamente il codice C, il C++ permette anche di chiamare il codice assembly utilizzando le normali convenzioni del name mangling del C.

Il C++ estende la parola chiave `extern` per permettergli di specificare che la funzione o la variabile globale a cui viene applicata, utilizzi la normale convenzione del C. Nella terminologia del C++, si dice che la funzione o la variabile globale usano *un collegamento C*. Per esempio, per dichiarare `printf` in modo che abbia un collegamento C, si usa il prototipo:

```
extern "C" int printf( const char *, ... );
```

Questo istruisce il compilatore di non usare le regole di name mangling su questa funzione, ma di usare invece le regole del C. Comunque, cosi' facendo,

[6]La corrispondenza non e' una corrispondenza esatta, il compilatore considerera' la corrispondenza dopo il casting degli argomenti. Le regole di questo processo vanno comunque oltre gli scopi di questo libro. Consulta un libro sul C++ per maggiori dettagli.

```
void f( int & x )      // la & indica un parametro per riferimento
{ x++; }

int main()
{
  int y = 5;
  f(y);                // viene passato il riferimento a y, nessuna & qui!
  printf ("%d\n", y);  // stampa 6!
  return 0;
}
```

Figura 7.11: Esempio di riferimento

la funzione `printf` non potrebbe essere sovraccaricata. Questo fornisce il modo piu' veloce per interfacciare il C++ con l'assembly, cioe' definire la funzione in modo che utilizi il collegamento C e che usi la convenzione di chiamata del C.

Per convenienze, il C++ inoltre fornisce il collegamento ad un blocco di funzioni o variabili globali che devono essere definite. Il blocco e' indicato dalle solite parentesi graffe.

```
extern "C" {
  /* variabili globali e prototipi di funzione
     a collegamento C                          */
}
```

Se esaminiamo i file di header dell'ANSI C che sono contenuti nei compilatori C/C++ troveremo le seguenti righe in cima ad ognuno di essi:

```
#ifdef __cplusplus
extern "C" {
#endif
```

e una costruzione simile in fondo contenente la parentesi graffa di chiusura. Il compilatore C++ definisce la macro `__cplusplus` (con *due* underscore in testa). Il dettaglio sopra racchiude l'intero file di header dentro un blocco `extern C` se il file di header e' compilato come C++, altrimenti non fa nulla se e' compilato come C (dal momento che il compilatore C genererebbe un errore di sintassi per `extern C`). Questa stessa tecnica puo' essere usata da qualunque programmatore per creare un file di header per le routine assembly che possa essere usato sia in C che in C++.

7.2.2 Riferimenti

I *riferimenti* sono un'altra nuova caratteristica del C++. Questi permettono il passaggio di parametri ad una funzione senza usare esplicitamente i puntatori. Per esempio, consideriamo il codice in Figura 7.11. In realta', i parametri riferimento sono molto semplici, sono giusto dei puntatori. Il compilatore semplicemente nasconde cio' al programmatore (come un compilatore Pascal implementa i parametri `var` come puntatori). Quando il compilatore genera l'assembly per la chiamata di funzione alla riga 7, passa l'*indirizzo* di `y`. Se uno avesse dovuto scrivere la funzione `f` in assembly, avrebbe dovuto farlo considerando che il prototipo sarebbe stato[7]:

```
void f( int * xp);
```

I riferimenti sono solo una convenienza che puo' essere molto utile in particolare per il sovraccaricamento degli operatori. Questa e' un'altra caratteristica del C++ che permette di ridefinire il significato dei piu' comuni operatori, per i tipi struttura e classe. Per esempio, un uso comune e' quello di ridefinire l'operatore piu' (`+`) per concatenare oggetti stringa. Cosi', se `a` e `b` sono stringhe, `a + b` dovrebbe ritornare la concatenazione delle stringhe `a` e `b`. Il C++ in realta' chiamera' una funzione per fare cio' (infatti, queste espressioni possono essere riscritte in notazione di funzione come `operator +(a,b)`). Per efficenza, occorre passare l'indirizzo dell'oggetto stringa invece del loro valore. Senza riferimenti, questo potrebbe essere fatto come `operator +(&a,&b)`, ma questo richiederebbe di scriverlo con la sintassi dell'operatore, cioe' `&a + &b`. Cio' potrebbe essere molto problematico. Invece, con l'utilizzo dei riferimenti, si puo' scriverlo come `a + b`, che appare molto naturale.

7.2.3 Funzioni In Linea

Le *funzioni in linea* sono un'altra caratteristica del C++ [8]. Le funzioni in linea sono state pensate per sostituire le macro basate sul preprocessore che accettano parametri. Richiamandoci al C, in cui una macro che eleva al quadrato un numero potrebbe essere scritta come:

```
#define SQR(x) ((x)*(x))
```

Poiche' il preprocessore non conosce il C e fa una semplice sostituzione, il calcolo corretto della risposta, in molti casi, richiede la presenza delle parentesi. Purtroppo anche questa versione non da' il risultato corretto per `SQR(x++)`.

[7] Naturalmente, potrebbero voler dichiarare la funzione con il collegamento C per evitare il name mangling come discusso nella sezione 7.2.1

[8] I compilatori C spesso supportano questa caratteristica come una estensione dell'ANSI C.

```
inline int inline_f ( int x )
{ return x*x; }

int f( int x )
{ return x*x; }

int main()
{
  int y, x = 5;
  y = f(x);
  y = inline_f (x);
  return 0;
}
```

Figura 7.12: Esempio di "in linea"

Le Macro sono usate perche' eliminano il sovraccarico dell'esecuzione di una chiamata di funzione, per le funzioni piu' semplici. Come ha dimostrato il capitolo sui sottoprogrammi, l'esecuzione di una chiamata di funzione richiede diversi passaggi. Per una funzione molto semplice, il tempo richiesto per eseguire la chiamata di funzione potrebbe essere superiore al tempo richiesto per eseguire le operazioni nella funzione stessa! Le funzioni in linea sono un modo piu' semplice di scrivere codice che sembra una normale funzione, ma che *non* esegue una `CALL` ad un comune blocco di codice. Invece, le chiamate alle funzioni in linea sono sostituite dal codice che esegue la funzione. Il C++ permette ad una funzione di essere in linea piazzando la parola chiave `inline` di fronte alla definizione della funzione. Per esempio, consideriamo la funzione dichiarata in Figura 7.12. La chiamata alla funzione `f` alla riga 10 esegue una normale chiamata a funzione (in assembly, si assume che `x` si trovi all' indirizzo `ebp-8` e `y` a `ebp-4`):

```
1       push   dword [ebp-8]
2       call   _f
3       pop    ecx
4       mov    [ebp-4], eax
```

Invece, la chiamata alla funzione `inline_f` alla riga 11 sarebbe:

```
1       mov    eax, [ebp-8]
2       imul   eax, eax
3       mov    [ebp-4], eax
```

In questi casi ci sono due vantaggi nell"'illineamento". Prima di tutto, le funzioni in linea sono piu' veloci. i parametri non sono passati nello stack, non viene creato nessuno stack frame e poi distrutto, nessun salto viene eseguito. Secondariamente, le chiamate delle funzioni in linea richiedono meno codice! Quest'ultimo punto e' vero per questo esempio, ma non e' sempre vero in generale.

Lo svantaggio principale dell"'illineamento" e' che il codice in linea non e' collegato percui il codice di una funzione in linea deve essere disponibile per *tutti* i file che lo utilizzano. Il precedente esempio di codice mostra questo. La chiamata ad una funzione non in linea richiedere solo di conoscere i parametri, il tipo del valore di ritorno, la convenzione di chiamata e il nome della etichetta per la funzione. Tutte queste informazioni sono disponibili dal prototipo della funzione. Utilizzando una funzione in linea, e' necessario conoscere il codice di tutta la funzione. Questo significa che se *una* parte della funzione e' modificata, *tutti* i file sorgenti che usano quella funzione devono essere ricompilati. Ricordati che per le funzione non in linea, se il prototipo non cambia, spesso i file che usano la funzione non necessitano di essere ricompilati. Per tutte queste ragioni, il codice per le funzioni in linea sono solitamente posti nei file di header. Questa pratica contraddice la normale regola del C secondo cui il codice eseguibile non e' *mai* posto nei file di header.

7.2.4 Classi

Una classe C++ descrive un tipo di *oggetto*. Un'oggetto contiene sia i membri dati che i membri funzioni[9]. In altre parole, una classe e' una `struct` con i dati e le funzioni associate. Consideriamo la semplice classe definita in Figura 7.13. Una variabile di tipo `Simple` apparira' come una normale `struct` in C con un solo membro `int`. Le funzioni *non* sono allocate nella memoria assegnata alla struttura. Le funzioni membro sono diverse rispetto alle altre funzioni. A loro viene passato un parametro *nascosto*. Questo parametro e' un puntatore all'oggetto su cui le funzioni membro agiscono.

In realta', il C++ usa la parola chiave `this` *per accedere al puntatore all' oggetto utilizzato dall'interno della funzione membro.*

Per esempio, consideriamo il metodo `set_data` della classe `Simple` in Figura 7.13. Se fosse stata scritta in C, sarebbe sembrata una funzione a cui e' stato passato esplicitamente un puntatore all'oggetto su cui essa avrebbe agito, come mostra il codice in Figura 7.14. L'opzione `-S` nel compilatore *DJGPP* (e anche nei compilatori *gcc* e Borland) dice al compilatore di produrre un file assembly contenente l'equivalente linguaggio assembly per il codice prodotto. Per *DJGPP* e *gcc* il file assembly termina con l'estensione `.s` e sfortunatamente usa la sintassi assembly AT&T, leggermente diversa

[9] spesso chiamate *funzioni membro* in C++ o piu' generalmente *metodi*.

```
class Simple {
public:
  Simple();                  // costruttore di default
  ~Simple();                 // distruttore
  int get_data() const;      // funzioni membro
  void set_data( int );
private:
  int data;                  // dati membro
};

Simple::Simple()
{ data = 0; }

Simple::~Simple()
{ /* null body */ }

int Simple::get_data() const
{ return data; }

void Simple::set_data( int x )
{ data = x; }
```

Figura 7.13: Una semplice classe C++

da quelle del NASM e MASM[10]. (I compilatori Borland e MS generano un file con l'estensione `.asm` utilizzando la sintassi del MASM.) La Figura 7.15 mostra il prodotto di *DJGPP* convertito nella sintassi NASM, con i commenti aggiunti per chiarire le ragione dei comandi. Alla prima riga, nota che al metodo `set_data` e' assegnata una etichetta "storpiata" che codifica il nome del metodo, il nome della classe e i parametri. Il nome della classe e' codificata perche' altre classi potrebbero avere un metodo chiamato `set_data` e i due metodi *devono* avere etichette diverse. I parametri sono codificati in maniera che la classe possa sovraccaricare il metodo `set_data` per accettare altri parametri come per le normali funzioni C++. Comunque, i diversi compilatori, come prima, codificheranno queste informazioni in modo diverso nella etichetta modificata.

Alle righe 2 e 3 appare il familiare prologo della funzione. Alla riga 5 il

[10]Il sistema del compilatore *gcc* include il proprio assembler chiamato *gas*. L'assembler *gas* usa la sintassi AT&T , cosi' il compilatore produce codice nel formato per il *gas*. Ci sono diverse pagine sul web che discutono le differenze fra i formati INTEL e AT&T. Esiste anche un programma gratuito chiamato `a2i`(http://www.multimania.com/placr/a2i.html) che converte il formato AT&T in formato NASM.

```
void set_data( Simple * object, int x )
{
  object->data = x;
}
```

Figura 7.14: Versione C di Simple::set_data()

```
1  _set_data__6Simplei:           ; nome modificto
2        push   ebp
3        mov    ebp, esp
4
5        mov    eax, [ebp + 8]   ; eax = puntatore all'oggetto (this)
6        mov    edx, [ebp + 12]  ; edx = parametro intero
7        mov    [eax], edx       ; data e' all'offset 0
8
9        leave
10       ret
```

Figura 7.15: Risultato del Compilatore per Simple::set_data(int)

primo parametro sullo stack e' memorizzato in `EAX`. Questo *non* e' il parametro `x`! Rappresenta invece il parametro nascosto[11] che punta all'oggetto sui cui interverra'. La riga 6 memorizza il parametro `x` in `EDX` e la riga 7 memorizza `EDX` nella double word a cui punta `EAX`. Questo rappresenta il membro `data` dell'oggetto `Simple` su cui si opera, che rappresenta l'unico dato nella classe, che e' allocato a offset 0 nella struttura `Simple`.

Esempio

Questa sezione usa i principi del capitolo per creare una classe C++ che rappresenta un intero senza segno di dimensione arbitraria. Dal momento che l'intero puo' essere di qualunque dimensione, sara' memorizzato in un array di interi senza segno (double word). Puo' essere reso di qualunque dimensione utilizzando l'allocamento dinamico. Le double word sono memorizzate in ordine inverso[12] (*i.e.* la double word meno significativa e' a indice 0). La Figura 7.16 mostra la definizione della classe `Big_int`[13]. La dimen-

[11] Come al solito, *nulla* e' nascosto nel codice assembly!

[12] Perche'? Poiche' le operazioni di addizione inizieranno il processo dal principio dell'array e si muoverano in avanti.

[13] vedi i codice sorgenti di esempio per il codice completo di questo esempio. Il testo si riferira' solo ad una parte di esso

```
class Big_int {
public:
  /*
   * Parametri:
   *   size          - dimensione dell'intero espresso come numero
   *                   di unsigned int
   *   initial_value - valore iniziale di Big_int come un normale
   *                   unsigned int
   */
  explicit Big_int( size_t     size,
                    unsigned initial_value = 0);
  /*
   * Parametri:
   *   size          - dimensione dell'intero espresso come numero
   *                   di unsigned int
   *   initial_value - valore iniziale di Big_int come stringa che
   *                   contiene la rappresentazione esadecimale
   *                   del valore
   */
  Big_int( size_t          size,
           const char *  initial_value );

  Big_int( const Big_int & big_int_to_copy );
  ~Big_int();

  // ritorna la dimensione di Big_int (in termini di unsigned int)
  size_t size() const;

  const Big_int & operator = ( const Big_int & big_int_to_copy );
  friend Big_int operator + ( const Big_int & op1,
                              const Big_int & op2 );
  friend Big_int operator - ( const Big_int & op1,
                              const Big_int & op2);
  friend bool operator == ( const Big_int & op1,
                            const Big_int & op2 );
  friend bool operator < ( const Big_int & op1,
                           const Big_int & op2);
  friend ostream & operator << ( ostream &      os,
                                 const Big_int & op );
private:
  size_t        size_ ;     // dimensione dell'array di unsigned
  unsigned * number_; // puntatore all'array che contiene i valori
};
```

Figura 7.16: Definizione della classe Big_int

```
// prototipi per le routine assembly
extern "C" {
  int add_big_ints ( Big_int &       res,
                     const Big_int & op1,
                     const Big_int & op2);
  int sub_big_ints ( Big_int &       res,
                     const Big_int & op1,
                     const Big_int & op2);
}

inline Big_int operator + ( const Big_int & op1, const Big_int & op2)
{
  Big_int result (op1.size ());
  int res = add_big_ints( result , op1, op2);
  if (res == 1)
    throw Big_int::Overflow();
  if (res == 2)
    throw Big_int::Size_mismatch();
  return result ;
}

inline Big_int operator - ( const Big_int & op1, const Big_int & op2)
{
  Big_int result (op1.size ());
  int res = sub_big_ints( result , op1, op2);
  if (res == 1)
    throw Big_int::Overflow();
  if (res == 2)
    throw Big_int::Size_mismatch();
  return result ;
}
```

Figura 7.17: Codice Aritmetico della Classe Big_int

sione di una `Big_int` e' misurata dalla dimensione dell'array di `unsigned` che viene utilizzato per memorizzare i dati. Al membro dati `size_` della classe e' assegnato offset zero e al membro `number_` e' assegnato offset 4.

Per semplificare questo esempio, solo le istanze di oggetto con la stessa dimensione di array possono essere sommate o sottratte l'un l'altra.

La classe ha tre costruttori: il primo (riga 9) inizializza l'istanza della classe utilizzando un normale intero senza segno; il secondo (riga 18) inizializza l'istanza utilizzando una stringa che contiene un valore esadecimale; il terzo costruttore (riga 21) e' il *costruttore di copia*.

Questa discussione si focalizza su come funzionano gli operatori di addizione e sottrazione dal momento che e' qui che viene usato il linguggio assembly. La Figura 7.17 mostra la parte di interesse dei file di header per questi operatori. Questi mostrano come gli operatori sono impostati per richiamare le routine in assembly. Dal momento che i diversi compilatori usano regole diverse per la modifica dei nomi delle funzioni operatore, le funzioni operatore in linea vengono usate per impostare routine assembly con collegamento C. Cio' rende relativamente facile la portabilita' a diversi compilatori e rende l'operazione piu' veloce rispetto alla chiamata diretta. Questa tecnica inoltre elimina la necessita' di sollevare una eccezione dall'assembly!

Perche' l'assembly e' usato solo qui? Ricorda che per eseguire l'aritmetica in precisione multipla, il riporto deve essere spostato di una dword per essere sommato alla dword significativa successiva. Il C++ (e il C) non permettono al programmatore di accedere al carry flag della CPU. L'esecuzione dell'addizione puo' essere fatta solo ricalcolando il carry flag indipendentemente dal C++ e sommandolo alla successiva dword. E' molto piu' efficente scrivere codice in assembly dove il carry flag puo' essere acceduto e utilizzando l' istruzione `ADC` che somma automaticamente il carry flag.

Per brevita', verra' discussa qui solo la routine assembly `add_big_ints` . Sotto e' riportato il codice per questa routine (da `big_math.asm`):

big_math.asm

```
1  segment .text
2          global  add_big_ints, sub_big_ints
3  %define size_offset 0
4  %define number_offset 4
5
6  %define EXIT_OK 0
7  %define EXIT_OVERFLOW 1
8  %define EXIT_SIZE_MISMATCH 2
9
10 ; Parametri per entrambe le routine di somma e sottrazione
11 %define res ebp+8
```

```
12 %define op1 ebp+12
13 %define op2 ebp+16
14
15 add_big_ints:
16         push    ebp
17         mov     ebp, esp
18         push    ebx
19         push    esi
20         push    edi
21         ;
22         ; prima si imposta  esi per puntare a op1
23         ;              edi per puntare a op2
24         ;              ebx per puntare a res
25         mov     esi, [op1]
26         mov     edi, [op2]
27         mov     ebx, [res]
28         ;
29         ; assicurati che tutte e 3 le Big_int abbiano la stessa dimensione
30         ;
31         mov     eax, [esi + size_offset]
32         cmp     eax, [edi + size_offset]
33         jne     sizes_not_equal                 ; op1.size_ != op2.size_
34         cmp     eax, [ebx + size_offset]
35         jne     sizes_not_equal                 ; op1.size_ != res.size_
36
37         mov     ecx, eax                        ; ecx = size of Big_int's
38         ;
39         ; ora, setta i registri per puntare ai rispettivi array
40         ;      esi = op1.number_
41         ;      edi = op2.number_
42         ;      ebx = res.number_
43         ;
44         mov     ebx, [ebx + number_offset]
45         mov     esi, [esi + number_offset]
46         mov     edi, [edi + number_offset]
47
48         clc                                ; pulisce carry flag
49         xor     edx, edx                   ; edx = 0
50         ;
51         ; addition loop
52 add_loop:
53         mov     eax, [edi+4*edx]
```

```
54          adc     eax, [esi+4*edx]
55          mov     [ebx + 4*edx], eax
56          inc     edx                         ; non altera carry flag
57          loop    add_loop
58
59          jc      overflow
60  ok_done:
61          xor     eax, eax                    ; valore di ritorno = EXIT_OK
62          jmp     done
63  overflow:
64          mov     eax, EXIT_OVERFLOW
65          jmp     done
66  sizes_not_equal:
67          mov     eax, EXIT_SIZE_MISMATCH
68  done:
69          pop     edi
70          pop     esi
71          pop     ebx
72          leave
73          ret
```

big_math.asm

Mi auspico che la maggior parte del codice sia chiaro al lettore in questo momento. Dalla riga 25 alla 27 vengono memorizzati i puntatori agli oggetti `Big_int` passati alla funzione attraverso i registri. Ricorda che i riferimenti solo in realta' solo dei puntatori. Le righe 31 fino alla 35 controllano per essere sicuri che le dimensioni degli array dei tre oggetti siano uguali. (Nota che l'offset di `size_` e' sommato al puntatore per accedere ai dati membro.) Le righe dalla 44 alla 46 aggiustano i registri per puntare all'array utilizzati nei rispettivi oggetti invece che agli oggetti stessi. (Di nuovo, l'offset del membro `number_` e' sommato al puntatore dell'oggetto.)

Le righe dalla 52 alla 57 sommano insieme gli interi memorizzati negli array sommando prima le dword meno significative, poi quelle meno significative successive, *etc.* La somma deve essere fatta in questa sequenza per l'aritmetica in precisione estesa (vedi la Sezione 2.1.5). La riga 59 controlla gli overflow, in caso di overflow il carry flag viene settato dall'ultima somma della dword piu' significativa. Dal momento che le dword sono memorizzate con ordine little endian, il ciclo inizia dal primo elemento dell'array e si sposta fino alla fine.

La Figura 7.18 mostra un piccolo esempio di utilizzo della classe `Big_int`. Nota che la costante `Big_int` deve essere dichiarata esplicitamente come alla riga 16. Questo e' necessario per due motivi. Il primo e' che non esiste nessun costruttore di conversione che converta un unsigned int in un `Big_int`. Secondo, solo gli oggetti `Big_int` della stessa dimensione possono essere

```
#include "big_int.hpp"
#include <iostream>
using namespace std;

int main()
{
  try {
    Big_int b(5,"8000000000000a00b");
    Big_int a(5,"80000000000010230");
    Big_int c = a + b;
    cout << a << " + " << b << " = " << c << endl;
    for( int i=0; i < 2; i++ ) {
      c = c + a;
      cout << "c = " << c << endl;
    }
    cout << "c-1 = " << c - Big_int(5,1) << endl;
    Big_int d(5, "12345678");
    cout << "d = " << d << endl;
    cout << "c == d " << (c == d) << endl;
    cout << "c > d " << (c > d) << endl;
  }
  catch( const char * str ) {
    cerr << "Caught: " << str << endl;
  }
  catch( Big_int :: Overflow ) {
    cerr << "Overflow" << endl;
  }
  catch( Big_int :: Size_mismatch ) {
    cerr << "Size mismatch" << endl;
  }
  return 0;
}
```

Figura 7.18: Semplice uso di `Big_int`

sommati. Cio' rende la conversione problematica dal momento che sarebbe molto difficile conoscere la dimensione in cui convertire. Una implementazione piu' sofisticata della classe avrebbe permesso la somma di qualunque dimensione con qualunque altra. L'autore non intendeva complicare oltre questo esempio implementando qui questa versione. (Il lettore, pero', e' incoraggiato a farlo).

7.2.5 Ereditarieta' e Polimorfismo

L'*Ereditarieta'* permette ad una classe di ereditare i dati ed i metodi di un'altra classe. Per esempio, consideriamo il codice in Figura 7.19. Mostra due classi, `A` e `B`, dove la classe `B` eredita dalla classe `A`. L'output del programma e':

```
Size of a: 4 Offset of ad: 0
Size of b: 8 Offset of ad: 0 Offset of bd: 4
A::m()
A::m()
```

Nota che i membri dati `ad` di entrambe le classi (`B` sono ereditati da `A`) e si trovano allo stesso offset. Questo e' importante dal momento che alla funzione `f` potrebbe essere passato un puntatore all'oggetto `A` o a qualsiasi oggetto di tipo derivato (*i.e.* derivato da) `A`. La Figura 7.20 mostra il codice asm (editato) per la funzione (generato da *gcc*).

Dall'output si nota che il metodo `m` di `A` viene chiamato da entrambi gli oggetti `a` e `b`. Dall'assembly si puo' vedere che la chiamata a `A::m()` e' codificata nella funzione. Per la vera programmazione orientata agli oggetti, il metodo chiamato deve dipendere dal tipo di oggetto passato alla funzione. Questo concetto e' conosciuto come *polimorfismo*. Il C++ disattiva questa caratteristica di default. Si utilizza la parola chiave *virtual* per abilitarla. La Figura 7.21 mostra come le due classi dovrebbero essere modificate. Niente dell' altro codice deve essere modificato. Il polimorfismo puo' essere implementato in diversi modi. Sfortunatamente, l'implementazione del *gcc* e' in transizione al momento della scrittura e sta diventando significativamente piu' complicata rispetto alla implementazione iniziale. Nell'ottica di semplificare questa discussione, l'autore coprira' l'implementazione del polimorfismo utilizzata dai compilatori Microsoft e Borland basati su Windows. Questa implementazione non ha subito variazioni per molti anni ed e' probabile che non ne subira' nel prossimo futuro.

Con queste modifiche, l'output del programma diventa:

```
Size of a: 8 Offset of ad: 4
Size of b: 12 Offset of ad: 4 Offset of bd: 8
A::m()
```

```
#include <cstddef>
#include <iostream>
using namespace std;

class A {
public:
  void __cdecl m() { cout << "A::m()" << endl; }
  int ad;
};

class B : public A {
public:
  void __cdecl m() { cout << "B::m()" << endl; }
  int bd;
};

void f( A * p )
{
  p->ad = 5;
  p->m();
}

int main()
{
  A a;
  B b;
  cout << "Size of a: " << sizeof(a)
       << " Offset of ad: " << offsetof(A,ad) << endl;
  cout << "Size of b: " << sizeof(b)
       << " Offset of ad: " << offsetof(B,ad)
       << " Offset of bd: " << offsetof(B,bd) << endl;
  f(&a);
  f(&b);
  return 0;
}
```

Figura 7.19: Ereditarieta' semplice

```
1  _f__FP1A:                        ; nome della funzione ‘‘modificato’’
2        push   ebp
3        mov    ebp, esp
4        mov    eax, [ebp+8]        ; eax punta all’oggetto
5        mov    dword [eax], 5      ; usa l’offset 0 per ad
6        mov    eax, [ebp+8]        ; passagio di indirizzo dell’oggetto a A::m()
7        push   eax
8        call   _m__1A              ; nome del metodo ‘‘modificato’’ per A::m()
9        add    esp, 4
10       leave
11       ret
```

Figura 7.20: Codice Assembly per l’ereditarieta’ semplice

```
class A {
public:
  virtual void __cdecl m() { cout << "A::m()" << endl; }
  int ad;
};

class B : public A {
public:
  virtual void __cdecl m() { cout << "B::m()" << endl; }
  int bd;
};
```

Figura 7.21: Ereditarieta’ Polimorfica

```
1  ?f@@YAXPAVA@@@Z:
2        push   ebp
3        mov    ebp, esp
4
5        mov    eax, [ebp+8]
6        mov    dword [eax+4], 5  ; p->ad = 5;
7
8        mov    ecx, [ebp + 8]    ; ecx = p
9        mov    edx, [ecx]        ; edx = puntatore alla vtable
10       mov    eax, [ebp + 8]    ; eax = p
11       push   eax               ; mette il puntatore "this"
12       call   dword [edx]       ; chiama la prima funzione in vtable
13       add    esp, 4            ; pulisce lo stack
14
15       pop    ebp
16       ret
```

Figura 7.22: Codice Assembly per la Funzione `f()`

`B::m()`

Ora la seconda chiamata a `f` chiama il metodo `B::m()` poiche' gli viene passato l'oggetto `B`. Questo non e' comunque l'unico cambiamento. La dimensione di una `A` e' ora 8 (e `B` e' 12). Inoltre l'offset di`ad` e' 4, non 0. Cosa c'e' a offset 0? La risposta a questa domanda e' collegata a come viene implementato il polimorfismo.

Una classe C++ che ha tutti i metodi virtuali, viene dotata di campo aggiuntivo, nascosto, che rappresenta un puntatore ad un'array di puntatori ai metodi[14]. Questa tabella e' spesso chiamata *vtable*. Per le classi `A` e `B` questo puntatore e' memorizzato ad offset 0. I compilatori Windows mettono sempre questo puntatore all'inizio della classe all'apice dell'albero delle ereditarieta'. Guardando al codice assembly (Figura 7.22) generato per la funzione `f` (dalla Figura 7.19) per la versione con metodo virtuale del programma , si puo' vedere che la chiamata al metodo `m` non e' ad una etichetta. La riga 9 trova l'indirizzo dell'oggetto nella vtable. L'indirizzo dell'oggetto e' messo nello stack alla riga 11. La riga 12 chiama il metodo virtuale saltando al primo indirizzo della vtable[15]. Questa chiamata non usa

[14]Per le classi senza metodi virtuali e' il compilatore C++ che si occupa di renderle compatibili con una normale struttura in C con gli stessi dati membro.

[15]Naturalmente, questo valore e' gia' nel registro `ECX`. C'era stato messo alla riga 8 e la riga 10 potrebbe essere rimossa mentre la riga successiva modificata per mettere `ECX` nello stack. Il codice non e' molto efficente poiche' e' stato generato senza l'attivazione delle

```
class A {
public:
  virtual void __cdecl m1() { cout << "A::m1()" << endl; }
  virtual void __cdecl m2() { cout << "A::m2()" << endl; }
  int ad;
};

class B : public A {     // B eredita m2() di A
public:
  virtual void __cdecl m1() { cout << "B::m1()" << endl; }
  int bd;
};
/* stampa la vtable di una dato oggetto */
void print_vtable ( A * pa )
{
  // p vede pa come un array di dword
  unsigned * p = reinterpret_cast<unsigned *>(pa);
  // vt vede vtable come un'array di puntatori
  void ** vt = reinterpret_cast<void **>(p[0]);
  cout << hex << "vtable address = " << vt << endl;
  for( int i=0; i < 2; i++ )
    cout << "dword " << i << ": " << vt[i] << endl;

  // chiama una funzione virtuale in un modo ESTREMAMENTE non
  // portabile !
  void (*m1func_pointer)(A *);    // variabile puntatore a funzione
  m1func_pointer = reinterpret_cast<void (*)(A*)>(vt[0]);
  m1func_pointer(pa);             // chiama il metodo m1 attraverso
                                  // il puntatore a funzione

  void (*m2func_pointer)(A *);    // variabile puntatore a funzione
  m2func_pointer = reinterpret_cast<void (*)(A*)>(vt[1]);
  m2func_pointer(pa);             // chiama il metodo m2 attraverso
                                  // il puntatore a funzione
}

int main()
{
  A a;    B b1;   B b2;
  cout << "a: " << endl;   print_vtable (&a);
  cout << "b1: " << endl;  print_vtable (&b);
  cout << "b2: " << endl;  print_vtable (&b2);
  return 0;
}
```

Figura 7.23: Esempio piu' complicato

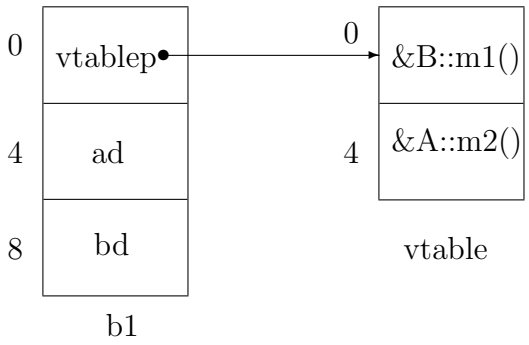


Figura 7.24: Rappresentazione interna di `b1`

una etichetta, salta all'indirizzo di codice puntato da `EDX`. Questo tipo di chiamata e' un esempio di *late binding*. Il late binding ritarda la decisione di quale metodo chiamare alla fase di esecuzione del codice. Cio' permette al codice di chiamare il metodo appropriato dell'oggetto. Il caso normale (Figure 7.20) codifica la chiamata ad un certo metodo nell'istruzione ed e' chiamato *early binding* (Dal momento che in questo caso il metodo e' collegato all'inizio, in fase di compilazione).

Il lettore attento si chiedera' perche' i metodi della classe in Figura 7.21 sono esplicitamente dichiarati per usare la convenzione di chiamata del C , con la parola chiave `__cdecl`. Come impostazione predefinita, Microsoft usa una diversa convenzione di chiamata per i metodi di una classe C++ invece della convenzione standard del C. Esso passa un puntatore all'oggetto sui cui il metodo agisce, nel registro `ECX` invece di usare lo stack. Lo stack e' ancora usato per gli altri parametri espliciti del metodo. Il modificatore `__cdecl` indica di usare la convenzione di chiamata standard del C. Il Borland C++ usa di defualt la convenzione standard

Guardiamo ora ad un'esempio leggermente piu' complicato (Figura 7.23). In questo, le classi `A` e `B` hanno entrambe due metodi: `m1` e `m2`. Ricorda che, dal momento che la classe `B` non definisce il proprio metodo `m2`, lo eredita dal metodo della classe `A`. La Figura 7.24 mostra come l'oggetto `b` appare in memoria. La Figura 7.25 mostra l'output del programma. Prima di tutto guardiamo gli indirizzi della vtable per ogni oggetto. Gli indirizzi dei due oggetti `B` sono gli stessi e cosi' essi condividono la stessa vtable. Una vtable rappresenta una proprieta' della classe e non un'oggetto (come un membro *statico*). Controlliamo poi gli indirizzi nelle vtable. Guardando all'output dell'assembly, si puo' stabilire che il puntatore al metodo `m1` si trova a offset 0 (o dword 0) e quello di `m2` a offset 4 (dword 1). I puntatori al metodo `m2` sono gli stessi per le vtable delle classi `A` e `B` poiche' la classe `B` eredita' il

ottimizzazioni del compilatore.

```
a:
indirizzo della vtable = 004120E8
dword 0:  00401320
dword 1:  00401350
A::m1()
A::m2()
b1:
indirizzo della vtable = 004120F0
dword 0:  004013A0
dword 1:  00401350
B::m1()
A::m2()
b2:
indirizzo della vtable = 004120F0
dword 0:  004013A0
dword 1:  00401350
B::m1()
A::m2()
```

Figura 7.25: Output del programma in Figura 7.23

motodo `m2` dalla classe `A`.

Le righe dalla 25 alla 32 mostra come si possa chiamare una funzione virtuale recuperando il suo indirizzo della vtable dell'oggetto[16]. L'indirizzo del metodo e' memorizzato in un puntatore a funzione di tipo C con un'esplicito puntatore *this*. Dall'output in Figura 7.25, si puo' vedere come funziona il tutto. Cerca pero' di *non* scrivere codice come questo! Questo codice e' utilizzato solo per illustrare come i metodi virtuali usano la vtable.

Ci sono alcune lezioni pratiche da imparare da questo esempio. Un fatto importante e' che occorre stare molto attenti quando si leggono e scrivono variabile classe da un file binario. Non e' possibile utilizzare una lettura o scrittura binaria dell'intero oggetto dal momento che queste operazioni leggerebbero o scriverebbero il puntatore alla vtable dal file! Questo puntatore indica dove risiede la vtable nella memoria del programma e puo' variare da programma a programma. Lo stesso problema nasce in C con le struct, ma in C le struct hanno dei puntatori solo se il programmatore le mette esplicitamente nelle struct. Non ci sono puntatori ovvii definiti nella classe `A` ne' nella classe `B`.

Di nuovo, e' importante capire che i diversi compilatori implementano i

[16] Ricorda che il questo codice funziona solo con i compilatori MS e Borland, non con *gcc*.

metodi virtuali diversamente. In Windows, gli oggetti classe COM (Component Object Model) usano le vtable per implementare le interfacce COM[17]. Solo i compilatori che implementano le vtables dei metodi virtuali come fa Microsoft possono creare classi COM. Questa e' la ragione per cui Borland usa la stessa implementazione di Microsoft ed una delle ragioni per cui *gcc* non puo' essere usato per creare classi COM.

Il codice per i metodi virtuali sembra esattamente come quello per i metodi non virtuali. Solo il codice che chiama e' diverso. Se il compilatore vuole essere assolutamente sicuro di quale metodo virtuale verra' chiamato, deve ignorare la vtable e chiamare il metodo direttamente (*e.g.*, utilizzando l'early binding).

7.2.6 Altre caratteristiche del C++

I lavori sulle altre caratteristiche del C++ (*e.g.* RunTime Type Information,gestione delle eccezioni ed ereditarieta' multipla) vanno oltre gli scopi di questo testo. Se il lettore vuole approfondire, un buon punto di partenza e' il *The Annotated C++ Reference Manual* di Ellis e Stroustrup e *The Design and Evolution of C++* di Stroustrup.

[17] Le classi COM inoltre usano la convenzione di chiamata `__stdcall` non quella standard C.

Appendice A

Istruzioni 80x86

A.1 Istruzioni non in Virgola Mobile

Questa sezione elenca e descrive le azioni e i formati delle istruzioni non in virgola mobile della famiglia delle CPU 80x86 Intel.

I formati usano le seguenti abbreviazioni:

R	registro generale
R8	registro 8-bit
R16	registro 16-bit
R32	registro 32-bit
SR	registro di segmento
M	memoria
M8	byte
M16	word
M32	double word
I	valore immediato

Queste possono essere combinate per istruzioni a operando multiplo. Per esempio, il formato *R, R* indica che l'istruzione accetta due operandi di tipo registro. Molte delle istruzioni a due operandi consentono lo stesso operando. L'abbreviativo *O2* e' usato per rappresentare questi operandi: *R,R R,M R,I M,R M,I*. Se e' possibile usare come operando un registro a 8 bit o la memoria, viene utilizzato l'abbreviativo *R/M8*.

La tabella inoltre mostra come i vari bit del registro flag sono influenzati da ogni istruzione. Se la colonna e' vuota, il bit corrispondente non viene influenzato. Se il bit viene sempre impostato ad un particolare valore, l'1 o lo 0 sono inidicati nella colonna. Se il bit e' impostato con un valore che dipende dall'operando dell'istruzione, nella colonna sara' posta una *C*. Infine, se il bit e' impostato in modo indefinibile, la colonna conterra' un

?. Poiche' le uniche istruzioni che modificano il flag di direzione sono `CLD` e `STD`, questo non e' compreso nella colonna di FLAGS.

			Flags					
Nome	**Descrizione**	**Formati**	**O**	**S**	**Z**	**A**	**P**	**C**
`ADC`	somma con Carry	O2	C	C	C	C	C	C
`ADD`	somma di interi	O2	C	C	C	C	C	C
`AND`	AND a livello di bit	O2	0	C	C	?	C	0
`BSWAP`	Byte Swap	R32						
`CALL`	Chiamata di Routine	R M I						
`CBW`	Converte Byte in Word							
`CDQ`	Converte Dword in Qword							
`CLC`	Pulisce Carry							0
`CLD`	Pulisce Direction Flag							
`CMC`	Complement del Carry							C
`CMP`	Compara Integi	O2	C	C	C	C	C	C
`CMPSB`	Compara Byte		C	C	C	C	C	C
`CMPSW`	Compara Word		C	C	C	C	C	C
`CMPSD`	Compara Dword		C	C	C	C	C	C
`CWD`	Converte Word in Dword in DX:AX							
`CWDE`	Converte Word in Dword in EAX							
`DEC`	Decrementa un Intero	R M	C	C	C	C	C	
`DIV`	Divisione senza segno	R M	?	?	?	?	?	?
`ENTER`	Crea uno stack frame	I,0						
`IDIV`	Divisione con segno	R M	?	?	?	?	?	?
`IMUL`	Moltiplicazione con segno	R M R16,R/M16 R32,R/M32 R16,I R32,I R16,R/M16,I R32,R/M32,I	C	?	?	?	?	C
`INC`	Incrementa un Intero	R M	C	C	C	C	C	
`INT`	Genera Interrupt	I						
`JA`	Salta se sopra	I						
`JAE`	Salta se sopra o uguale	I						
`JB`	Salta se sotto	I						
`JBE`	Salta se sotto o uguale	I						

Nome	Descrizione	Formati	Flags					
			O	S	Z	A	P	C
JC	Salta se Carry e' impostato	I						
JCXZ	Salta se CX = 0	I						
JE	Salta se uguale	I						
JG	Salta se piu' grande	I						
JGE	Salta se piu' grande o uguale	I						
JL	Salta se piu' piccolo	I						
JLE	Salta se piu' piccolo o uguale	I						
JMP	Salto incondizionato	R M I						
JNA	Salta se non sopra	I						
JNAE	Salta se non sopra o uguale	I						
JNB	Salta se non sotto	I						
JNBE	Salta se non sotto o uguale	I						
JNC	Salta se il carry non e' impostato	I						
JNE	Salta se non uguale	I						
JNG	Salta se non piu' grande	I						
JNGE	Salta se non piu' grande o uguale	I						
JNL	Salta se non piu' piccolo	I						
JNLE	Salta se non piu' piccolo o uguale	I						
JNO	Salta se non Overflow	I						
JNS	Salta se non Sign	I						
JNZ	Salta se non Zero	I						
JO	Salta se Overflow	I						
JPE	Salta se Parita' pari	I						
JPO	Salta se Parita' dispari	I						
JS	Salta se Sign	I						
JZ	Salta se Zero	I						
LAHF	Carica FLAGS in AH							
LEA	Carica l'indirizzo effettivo	R32,M						
LEAVE	Lascia lo stack frame							

Nome	Descrizione	Formati	Flags					
			O	S	Z	A	P	C
LODSB	carica Byte							
LODSW	carica Word							
LODSD	carica Dword							
LOOP	ciclo	I						
LOOPE/LOOPZ	ciclo se uguale	I						
LOOPNE/LOOPNZ	ciclo se non uguale	I						
MOV	Muove Dati	O2						
		SR,R/M16						
		R/M16,SR						
MOVSB	Muove Byte							
MOVSW	Muove Word							
MOVSD	Muove Dword							
MOVSX	Muove Signed	R16,R/M8						
		R32,R/M8						
		R32,R/M16						
MOVZX	Muove Unsigned	R16,R/M8						
		R32,R/M8						
		R32,R/M16						
MUL	Moltiplicazione senza segno	R M	C	?	?	?	?	C
NEG	Negazione	R M	C	C	C	C	C	C
NOP	Nessuna operazione							
NOT	Complemento a 1	R M						
OR	OR a livello di bit	O2	0	C	C	?	C	0
POP	Estrai dallo Stack	R/M16						
		R/M32						
POPA	Estrai tutto							
POPF	Estrai FLAGS		C	C	C	C	C	C
PUSH	Metti nello Stack	R/M16						
		R/M32 I						
PUSHA	Metti tutto							
PUSHF	Metti FLAGS							
RCL	Rotazione a Sx con Carry	R/M,I	C					C
		R/M,CL						
RCR	Rotazione a Dx con Carry	R/M,I	C					C
		R/M,CL						
REP	Repeti							
REPE/REPZ	Repeti Se uguale							
REPNE/REPNZ	Repeti Se non uguale							
RET	Retorna							

Nome	Descrizione	Formati	Flags					
			O	S	Z	A	P	C
ROL	Rotazione a Sx	R/M,I R/M,CL	C					C
ROR	Rotazione a Dx	R/M,I R/M,CL	C					C
SAHF	Copia AH in FLAGS			C	C	C	C	C
SAL	Shift a Sx	R/M,I R/M, CL						C
SBB	Sottrai con resto	O2	C	C	C	C	C	C
SCASB	Cerca un Byte		C	C	C	C	C	C
SCASW	Cerca una Word		C	C	C	C	C	C
SCASD	Cerca una Dword		C	C	C	C	C	C
SETA	Imposta se sotto	R/M8						
SETAE	Imposta se sotto o uguale	R/M8						
SETB	Imposta se sopra	R/M8						
SETBE	Imposta se sopra o uguale	R/M8						
SETC	Imposta se Carry	R/M8						
SETE	Imposta se uguale	R/M8						
SETG	Imposta se piu' grande	R/M8						
SETGE	Imposta se piu' grande o uguale	R/M8						
SETL	Imposta se minore	R/M8						
SETLE	Imposta se minore o uguale	R/M8						
SETNA	Imposta se non sopra	R/M8						
SETNAE	Imposta se non sopra o uguale	R/M8						
SETNB	Imposta se non sotto	R/M8						
SETNBE	Imposta se non sotto o uguale	R/M8						
SETNC	Imposta Se No Carry	R/M8						
SETNE	Imposta se non uguale	R/M8						
SETNG	Imposta Se non piu' grande	R/M8						
SETNGE	Imposta Se non piu' grande o uguale	R/M8						
SETNL	Imposta se non minore	R/M8						
SETNLE	Imposta se non minore o uguale	R/M8						

Nome	Descrizione	Formati	Flags					
			O	S	Z	A	P	C
SETNO	Imposta se No Overflow	R/M8						
SETNS	Imposta se No Sign	R/M8						
SETNZ	Imposta se Not Zero	R/M8						
SETO	Imposta se Overflow	R/M8						
SETPE	Imposta se Parita' pari	R/M8						
SETPO	Imposta se Parita' dispari	R/M8						
SETS	Imposta se Sign	R/M8						
SETZ	Imposta se Zero	R/M8						
SAR	Shift aritmetico a Dx	R/M,I R/M, CL						C
SHR	Shift logico a Dx	R/M,I R/M, CL						C
SHL	Shift logico a Sx	R/M,I R/M, CL						C
STC	Imposta se Carry							1
STD	Imposta se Direction Flag							
STOSB	Memorizza Btye							
STOSW	Memorizza Word							
STOSD	Memorizza Dword							
SUB	Sottrae	O2	C	C	C	C	C	C
TEST	Camporazione Logica	R/M,R R/M,I	0	C	C	?	C	0
XCHG	Scambio	R/M,R R,R/M						
XOR	XOR a livello di bit	O2	0	C	C	?	C	0

A.2 Istruzioni in virgola mobile

In questa sezione vengono descritti la maggior parte delle istruzioni per il coprocessore matematico degli 80x86. La colonna "descrizione" descrive brevemente l'operazione eseguita dall'istruzione. Per salvare spazio, le informazioni circa l'estrazione o meno dall'stack da parte dell'istruzione non vengono date nella descrizione.

La colonna "Formato" mostra i tipi di operando che sono usati con ogni istruzione. Sono utilizzate le seguenti abbreviazioni:

STn	Un registro del coprocessore
F	Numero a precisione singola in memoria
D	Numero a precisione doppia in memoria
E	Numero a precisione estesa in memoria
I16	Word intera in memoria
I32	Double word intera in memoria
I64	Quad word intera in memoria

Le istruzioni del processore Pentium Pro o superiore sono marcate con un'asterisco(*).

Instruzione	Descrizione	Formato
`FABS`	`ST0` = |`ST0`|	
`FADD` *src*	`ST0 +=` *src*	STn F D
`FADD` *dest*`, ST0`	*dest* `+= ST0`	STn
`FADDP` *dest*`[,ST0]`	*dest* `+= ST0`	STn
`FCHS`	`ST0` = $-$`ST0`	
`FCOM` *src*	Compara `ST0` e *src*	STn F D
`FCOMP` *src*	Compara `ST0` e *src*	STn F D
`FCOMPP` *src*	Compara `ST0` e `ST1`	
`FCOMI`* *src*	Compara in FLAGS	STn
`FCOMIP`* *src*	Compara in FLAGS	STn
`FDIV` *src*	`ST0 /=` *src*	STn F D
`FDIV` *dest*`, ST0`	*dest* `/= ST0`	STn
`FDIVP` *dest*`[,ST0]`	*dest* `/= ST0`	STn
`FDIVR` *src*	`ST0 =` *src*`/ST0`	STn F D
`FDIVR` *dest*`, ST0`	*dest* `= ST0/`*dest*	STn
`FDIVRP` *dest*`[,ST0]`	*dest* `= ST0/`*dest*	STn
`FFREE` *dest*	Segna come vuoto	STn
`FIADD` *src*	`ST0 +=` *src*	I16 I32
`FICOM` *src*	Compara `ST0` e *src*	I16 I32
`FICOMP` *src*	Compara `ST0` e *src*	I16 I32
`FIDIV` *src*	`ST0 /=` *src*	I16 I32

Instruzione	Descrizione	Formato
FIDIVR *src*	ST0 = *src*/ST0	I16 I32
FILD *src*	Mette *src* nello Stack	I16 I32 I64
FIMUL *src*	ST0 *= *src*	I16 I32
FINIT	Inizializza il Coprocessore	
FIST *dest*	Memorizza ST0	I16 I32
FISTP *dest*	Memorizza ST0	I16 I32 I64
FISUB *src*	ST0 -= *src*	I16 I32
FISUBR *src*	ST0 = *src* - ST0	I16 I32
FLD *src*	Mette *src* nello Stack	STn F D E
FLD1	Mette 1.0 nello Stack	
FLDCW *src*	Carica il registro della word di controllo	I16
FLDPI	Mette π nello Stack	
FLDZ	Mette 0.0 nello Stack	
FMUL *src*	ST0 *= *src*	STn F D
FMUL *dest*, ST0	*dest* *= ST0	STn
FMULP *dest*[,ST0]	*dest* *= ST0	STn
FRNDINT	Arrotonda ST0	
FSCALE	$\mathtt{ST0} = \mathtt{ST0} \times 2^{\lfloor \mathtt{ST1} \rfloor}$	
FSQRT	$\mathtt{ST0} = \sqrt{\mathtt{ST0}}$	
FST *dest*	Memorizza ST0	STn F D
FSTP *dest*	Memorizza ST0	STn F D E
FSTCW *dest*	Memorizza registro della word di controllo	I16
FSTSW *dest*	Memorizza registro della word di controllo	I16 AX
FSUB *src*	ST0 -= *src*	STn F D
FSUB *dest*, ST0	*dest* -= ST0	STn
FSUBP *dest*[,ST0]	*dest* -= ST0	STn
FSUBR *src*	ST0 = *src*-ST0	STn F D
FSUBR *dest*, ST0	*dest* = ST0-*dest*	STn
FSUBP *dest*[,ST0]	*dest* = ST0-*dest*	STn
FTST	Compare ST0 with 0.0	
FXCH *dest*	Scambia ST0 e *dest*	STn

Indice analitico