Parte del corso di *Architetture degli Elaboratori*

su:

Assembler e linguaggio Assembly

Anno Accademico 2012-2013

Docente: ***Simona Rombo***

Il linguaggio e il programma

Questa parte del corso è dedicata allo studio del *linguaggio assemblatore*, meglio conosciuto come *Assembly*, che rappresenta il linguaggio di programmazione più vicino al linguaggio macchina vero e proprio, sebbene non sia coincidente con quest'ultimo. Uno degli errori più diffusi è quello di identificare l'Assembly, ovvero il linguaggio di programmazione, con l'*Assembler*, che è invece il programma utilizzato per convertire il linguaggio Assembly in linguaggio macchina.

Per poter affrontare adeguatamente lo studio del linguaggio Assembly, è bene richiamare alcune nozioni basilari relative al funzionamento dei microprocessori.

Ciascun microprocessore possiede un certo set di istruzioni, cioè può eseguire un numero più o meno grande (ma comunque finito) di operazioni. Un programma è costituito da una sequenza di istruzioni che permette al microprocessore di assolvere un particolare compito di calcolo e/o controllo. Le istruzioni che il microprocessore deve leggere ed eseguire sono immagazzinate nella memoria in forma di codice binario, ovvero, sono codificate proprio in linguaggio macchina. Ogni istruzione è costituita da un certo numero di byte ed il programma, nel suo insieme, non è altro che una successione di byte che va ad occupare una certa porzione di memoria.

Redigere un programma direttamente in codice binario non è, come possiamo immaginare, la più agevole delle soluzioni possibili, inoltre in tal modo verrebbero generati con elevata probabilità numerosi errori. L'utilizzo del linguaggio Assembly consente di superare tali difficoltà attraverso l'adozione di una forma simbolica (*codice mnemonico*) che richiama con una notazione sintetica il modo di operare di ogni istruzione. Possiamo quindi considerare Assembly come un linguaggio più orientato verso l'utilizzo umano di quanto non lo sia il linguaggio macchina, poichè può essere utilizzato più agevolmente e, tuttavia, ne conserva i principali vantaggi in termini di sintesi e velocità di esecuzione. Infatti, ad ogni istruzione in linguaggio Assembly corrisponde una sola istruzione in linguaggio macchina, diversamente dai linguaggi ad alto livello in cui ad una singola istruzione possono corrispondere più istruzioni in linguaggio macchina. D'altro canto, i linguaggi ad alto livello permettono di scrivere programmi in modo più semplice ed intuitivo.

La corrispondenza uno ad uno fra istruzione in linguaggio Assembly ed istruzione in linguaggio macchina fa sì che non ci sia un unico linguaggio Assembly ma che esso sia diverso da microprocessore a microprocessore. Noi ci concentreremo su quella che viene chiamata *Architettura* $\times 86$, di cui richiameremo di seguito dei brevi cenni storici. In particolare, faremo riferimento alla CPU dell'$i386$ in tutta la trattazione di questa parte del corso.

Architettura $\times 86$: qualche cenno storico

Architettura $\times 86$ è un'espressione usata in senso ampio, per designare l'architettura di una famiglia di microprocessori che derivano dall'originale CPU 8086, inizialmente introdotta da Intel. Al momento, questa architettura è la più diffusa nel mercato dei PC desktop, portatili, e server economici. Nel corso degli anni, più di una ditta ha introdotto processori compatibili con l'architettura $\times 86$, ponendosi in concorrenza con Intel. Un altro grande costruttore di microprocessori che si inquadrano in questa architettura è, ad esempio, AMD (*Advanced Micro Devices*). In origine, AMD aveva un accordo con Intel per lo sviluppo e la produzione della CPU 8086. Poi le due società hanno proseguito separatamente, e AMD ha iniziato a produrre CPU compatibili e concorrenziali con quelle prodotte da Intel, seguendo una propria tecnologia.

L'obiettivo dichiarato da Intel al momento dell'introduzione della CPU 8086 era la realizzazione di un microprocessore che migliorasse, di un ordine di grandezza, le prestazioni del precedente microprocessore a 8 bit 8080/8085. L'8086 fu il primo microprocessore di seconda generazione, basato su un'architettura a 16 bit e con uno spazio degli indirizzi di almeno $1M$, ad essere disponibile sul mercato già dal 1978, precedendo così di circa un anno i diretti concorrenti, quali lo $Z8000$ (Zilog) eil 68000 (Motorola). In tal modo, l'Intel fu in grado di conquistare una larga fetta del mercato professionale e industriale.

La famiglia $\times 86$ si impose in modo schiacciante con l'8088, introdotto circa un anno dopo l'8086. Tale microprocessore era completamente compatibile con l'8086, a cui era identico dal punto di vista software, aveva un parallelismo interno di 16 bit, capace di indirizzare fino a 1 MB di memoria, ed un bus dati esterno ridotto a 8 bit, che lo rendeva compatibile con l'hardware sviluppato per le macchine a 8 bit di allora. La principale caratteristica che distingueva in modo apprezzabile l'8088 da 8085, $Z80$, 6800 e da tutti gli altri microprocessori a 8 bit della precedente generazione, consisteva nel fatto che questi ultimi avevano uno spazio di indirizzamento limitato a 64 KB. L'8088 era il primo microprocessore a 8 bit a sfondare tale confine, mettendo a disposizione uno spazio di memoria che avrebbe consentito lo sviluppo di programmi e applicazioni adeguate al soddisfacimento delle esigenze dell'utente generico. L'8088 aveva le prestazioni di una CPU a 16 bit, ma il fatto di avere un bus dati a 8 bit permetteva di ridurre i costi dell'elettronica, senza contare che le periferiche dell'epoca impiegabili su personal computer erano esclusivamente a 8 bit.

Nel 1981, l'IBM introdusse un personal computer, il PC IBM (Figura 1 (a)), basato su CPU 8088, con frequenza di clock pari a $4,77$ MHz. Il PC IBM non era il primo personal computer apparso sul mercato. Da anni, infatti, era in commercio una varietà

di calcolatori personali basati su microprocessori a 8 bit come ad esempio l'8085 o lo $Z80$. Queste macchine impiegavano di solito il sistema operativo CP/M, che risultava essere il punto di riferimento per l'epoca e sembrava non temere alcuna concorrenza.

Nello stesso periodo, la società Apple procedeva lungo un percorso proprio e parallelo rispetto a quello degli altri produttori, distribuendo il personal computer Apple II (Figura 1 (b)) che era basato su microprocessore 6502 (Rockwell), anch'esso a 8 bit, dotato di un proprio sistema operativo. L'Apple II nacque nel 1977 e fu proprio il primo computer per il quale fu usata l'espressione *personal computer*, nonchè il primo modello di successo di tale categoria prodotto su scala industriale. Steve Jobs e Steve Wozniak nel 1976 avevano già costruito nel loro garage l'Apple I, un computer che però poteva essere appetibile solo ad un pubblico di appassionati di elettronica. Jobs desiderava rendere l'informatica accessibile a tutti, pertanto, il progetto dell'Apple I venne rielaborato mettendo tutta la parte elettronica in una scatola di plastica beige comprensiva di tastiera, dando così forma al personal computer che utilizziamo ancora oggi. Il microprocessore utilizzato per l' Apple II fu il 6502, un microprocessore a 8 bit distribuito negli stessi anni dello $Z80$.

(a) (b)

Figura 1: (a) PC IBM (b) Apple II

Quando l'IBM introdusse il PC, per il grande pubblico americano fu come se il PC nascesse in quel momento. Infatti, fino agli inizi degli anni ottanta, il mercato dei calcolatori era rappresentato in modo quasi esclusivo dagli USA, dove la sigla "IBM" e la parola "calcolatore" venivano considerati come sinonimi, identificando il marchio di fabbrica con il prodotto realizzato. Grandi utenti quali industrie, banche e apparati statali, fino ad allora piuttosto refrattari nell'utilizzo dei calcolatori, iniziarono a utilizzare il PC IBM, che vide in tal modo la sua prima, ampia, diffusione. Si pensi che la rivista *Time*, la quale per tradizione dedica la copertina dell'ultimo numero di ogni annata al personaggio maggiormente distintosi sul pianeta nel corso dell'anno, nel 1981, con grande sorpresa dei lettori, dedicò la copertina al PC invece che ad una persona.

Nessuno avrebbe mai sospettato a quei tempi il successo che avrebbe avuto il PC IBM, nemmeno la società stessa, che addirittura aveva progettato e sviluppato il prodotto in una sede periferica dell'azienda, situata in Florida, invece che nella vasta area del Nord dello stato di New York dove la società aveva i principali centri di progettazione e produzione. Inoltre, l'IBM avrebbe voluto una versione *propria* del sistema operativo CP/M in voga a quei tempi, con caratteristiche che la distinguessero dal resto dei produttori. Tuttavia, sembra che il produttore del CP/M (la società Digital Research), forte della sua posizione di predominanza sul mercato dei sistemi operativi per personal computer, abbia tenuto una posizione alquanto "distaccata" nei confronti dell'IBM. I responsabili del colosso informatico si rivolsero allora a una piccola ditta, la Microsoft, nota allora per un diffuso interprete BASIC per i microprocessori a 8 bit dell'epoca. La Microsoft accettò di buon grado di lavorare per IBM. Iniziava così il percorso che avrebbe portato la Microsoft a contendere il primato alla stessa IBM, mentre *un certo* Bill Gates, a quei tempi, era solo poco più che un ragazzino.

Il successo di IBM non passò inosservato: le industrie informatiche delle "tigri orientali" (Taiwan, Singapore, etc.) si misero subito al lavoro per clonare il PC IBM. La clonazione, cioè la duplicazione, fu possibile poichè IBM forniva assieme al PC anche gli schemi elettrici, ed il listato del sistema operativo era facilmente ottenibile, i componenti utilizzati, chip di memoria, processore, unità a disco erano "standard" e disponibili per tutti.

Il passo per la produzione industriale dei cloni fu brevissimo. In pochi anni il mondo fu invaso da enormi quantità di PC clonati, dalle prestazioni sempre più brucianti e dai costi sempre più bassi. Contemporaneamente, la Microsoft controllava il mondo dei sistemi operativi per la famiglia di tutti i microprocessori Intel, diventando nel tempo la più potente software house del mondo. Il duopolio Microsoft e Intel ha suggerito la coniazione del termine WinTel dall'unione di Windows e Intel.

Nel 1982 Intel introdusse il microprocessore 80286 a 16*bit* della famiglia $\times 86$. Il suo predecessore, l'80186, fu poco venduto e poco diffuso, fatta eccezione per il mondo dell'automazione industriale. L'80286 fu il primo microprocessore Intel ad avere una modalità protetta e diversi livelli di privilegio per il codice da eseguire. Aveva un bus dati a 16 bit e un bus indirizzi a 20 bit, che lo rendeva in grado di indirizzare fino a 16 MB di memoria. Restava però ancorato al vecchio schema di indirizzamento segmento/offset, troppo rigido, e non non supportava in hardware nessuno schema di memoria virtuale. Inoltre non era possibile tornare alla modalità reale una volta entrati in modo protetto. Altra caratteristica innovativa era il prefetching delle istruzioni, che lo rendeva molto più veloce, anche a parità di clock, dell'8086.

Nel 1984 Microsoft iniziò ad annunciare l'arrivo di Windows, un'interfaccia grafica

che avrebbe applicato al suo sistema operativo MS-DOS che era venduto con i PC IBM e compatibili dal 1981. Microsoft aveva creato l'interfaccia utente, all'inizio conosciuta col nome Interface Manager, seguendo i prototipi di interfaccia grafica della Xerox e seguì la strada intrapresa dalla Apple con il suo Macintosh.

Nell'ottobre del 1985 Intel presentò la pietra miliare nell'evoluzione della serie di processori ×86, l'80386. L'80386 fu il primo microprocessore di Intel con architettura a 32 bit e modalità protetta con supporto hardware alla memoria virtuale paginata. Fu usato come CPU per personal computer dal 1986 al 1994 e anche in seguito. Durante la fase di progettazione fu chiamato "P3", essendo la terza generazione di processori Intel con architettura ×86, ma viene indicato anche come *i*386.

Rispetto al suo predecessore, l'Intel 80286, il 80386 aveva un'architettura a 32, un bus indirizzi e dati a 32 bit, in grado di gestire fino a 4G di RAM, ed una unità di paginazione della memoria che rese più semplice adottare sistemi operativi dotati di gestione della memoria virtuale. Anche se successivamente la Intel sviluppò e introdusse molti altri modelli, per molto tempo nessuno di questi introdusse novità di portata analoga, fino all'impiego dell'EM64T nel 2004, in netto ritardo rispetto ad altre architetture, come ad esempio DEC Alpha, che già dal 1992 aveva il supporto per i 64 bit. Il *i*386 viene dunque considerato come capostipite di una nuova famiglia, l'"architettura *i*386". Il set di istruzioni di questa architettura è noto come IA-32. L'80386 era compatibile con i vecchi processori Intel: la maggior parte delle applicazioni che giravano sui precedenti PC dotati di processori ×86 (8086 e 80286) funzionavano ancora sulle macchine 80386, e perfino più velocemente.

L'80386 fu il primo processore ad essere distribuito in maniera esclusiva dalla Intel e tale esclusività diede un grande controllo alla Intel sullo sviluppo di questo microprocessore, garantendo all'azienda profitti ancora crescenti. Tuttavia, nel marzo 1991 la AMD, dopo aver superato alcuni ostacoli legali, introdusse il suo processore compatibile *Am*386, facendo così venir meno il monopolio della Intel. Il successo e il basso costo dei PC con *i*386 permisero la diffusione dei primi sistemi operativi in modalità protetta su personal computer. L'80386 fu anche il processore su cui nacque il kernel Linux.

Dopo l'*i*386 Intel introdusse l'*i*486, ultimo a seguire la denominazione ×86. Da un punto di vista software, l'80486 era molto simile al predecessore 80386, se non per l'aggiunta di alcune istruzioni. Da un punto di vista hardware, invece, questo processore fu molto innovativo, avendo una memoria cache di 8 kb unificata per dati e istruzioni, un'ulteriore unità di calcolo in virgola mobile (FPU) (opzionale, inclusa solo nella versione DX, DX2 e DX4), una bus interface unit migliorata, e le caratteristiche di Power management e l'SMM (System Management Mode) che divennero standard nel processore. In condizioni ottimali, questo processore poteva eseguire un'istruzione per ciclo

di clock. Questi miglioramenti permisero all'$i486$ di offrire prestazioni quasi doppie rispetto a quelle del predecessore, a parità di clock. Ciò nonostante, alcuni dei modelli di fascia più bassa, specialmente le prime versioni SX a 16 e 25 MHz, erano effettivamente inferiori rispetto agli $80386DX$ più veloci (33 e 40 MHz).

Il 22 marzo del 1993, come successore dell'Intel 80486, venne introdotto sul mercato il Pentium (Figura 2), appartenente alla quinta generazione di microprocessori con architettura $\times 86$. Inizialmente Intel pensava di continuare la numerazione progressiva indicando questo processore come Intel 80586, o $i586$. Tuttavia, poichè i numeri non possono essere registrati come trademark (ovvero marchio registrato), mentre le parole sì, nel 1992 Intel affidò ad una società specializzata, la Lexicon Branding (famosa per aver creato anche i nomi commerciali del PowerBook di Apple, del Blackberry di RIM, Tungsten e Zire di Palm, e InDesign di Adobe), il compito di coniare un nuovo nome per il processore di quinta generazione. Volendo rimarcare anche nel nome la generazione del processore, questo divenne efficacemente "Pentium" (dato che il prefisso pent- in greco significa proprio cinque).

Rispetto all'$i486$, il Pentium possedeva un'architettura superscalare con due pipeline che gli permettevano di completare più di una operazione per ciclo di clock. Inoltre, aveva un data path a 64 bit, caratteristica che raddoppiava la quantità di informazioni prelevate dalla memoria in ogni operazione di fetch. É importante sottolineare però che questo aspetto non consentiva assolutamente al Pentium di poter eseguire codice a 64 bit, dato che i suoi registri continuavano ad essere 32 bit. I Pentium, potendo eseguire più istruzioni per singolo ciclo di clock, offrivano prestazioni di poco inferiori al doppio di quelle di un $i486$ di pari frequenza. Gli ultimi $Am486$ di AMD,con frequenze di 133 MHz, raggiungevano nel calcolo sugli interi le prestazioni di un Pentium 75 MHz, risultando comunque più lenti nelle operazioni in virgola mobile.

Figura 2: Microprocessore Pentium

Il processore 80386

L'*i*386 presenta i seguenti modi operativi:

- **Real Mode:** emulazione 8086/8088, 1 Mb di memoria gestibile, 20 bit per gli indirizzi, 16 bit per i dati, MS-DOS.
- **Protected Mode:** sono abilitate tutte le protezioni e le priorità dell'*i*386; da questo ambiente possono essere selezionate, a loro volta, le seguenti due modalità operative.
  - 80286 *Protected Mode*: (*i*286 emulation): emula il 286 con 24 bit per gli indirizzi e 16 bit per i dati.
  - *Virtual Mode* 8086: uno o più task possono essere eseguiti in questa modalità; ogni task utilizza una macchina 8086 virtuale in cui può operare il DOS (modalità usata da alcuni sistemi operativi per attivare finestre DOS). Le istruzioni eseguite dal task vengono interpretate secondo lo stile 8086, quindi senza protezioni all'interno del task, ma al di fuori di questo ci sono tutte le protezioni del 386.

Di seguito descriveremo alcuni aspetti dell'ambiente di programmazione del 386 nel caso in cui il processore si trovi in modalità protetta (protected mode). Questa modalità di funzionamento fornisce direttamente dal lato hardware un supporto alla multiprogrammazione, impedendo che un programma possa accedere per errore all'area di memoria assegnata ad un altro programma. Il programma "vede" la memoria disponibile in modo trasparente all'utente, ma quale porzione di memoria fisica stia effettivamente utilizzando è nascosto all'utente.

Organizzazione della Memoria

Nel modello di Von Neumann la memoria centrale contiene sia le istruzioni che i dati. Spetta poi alla logica della CPU leggere correttamente le istruzioni dalla memoria, decodificarle e generare i comandi (segnali) che fanno eseguire le azioni previste da ogni specifica istruzione sui dati considerati. L'elaborazione dell'informazione avviene all'interno della CPU, facendo sì che i dati vengano letti dalla memoria, manipolati nella CPU ed, eventualmente, tornino (magari in forma diversa) ad essere scritti in memoria (fetch-decode-execute).

In generale, la memoria si compone di *celle* o *locazioni*, ciascuna delle quali è a sua volta composta da un numero prefissato di bit. Per poter leggere o scrivere informazioni

in memoria è necessario associare un nome, ovvero un *indirizzo*, a ciascuna di queste locazioni. L'indirizzo è un numero che varia nel range $[0, M-1]$, se M rappresenta l'estensione della memoria.

La memoria fisica di un *i*386 è organizzata come una sequenza di byte, ovvero, ciascuna locazione di memoria contiene esattamente 8 bit. É possibile indirizzare $2^{32}-1$ bytes (ovvero 4G). I programmi dell'80386, tuttavia, sono indipendenti dallo spazio di indirizzamento fisico. Questo vuol dire, come accennato in precedenza, che i programmi possono essere scritti senza avere conoscenza di quanta memoria fisica sia effettivamente disponibile, e senza sapere dove istruzioni e dati vengano allocati nella memoria fisica. Il modello di organizzazione della memoria visto dai programmatori può variare a seconda dell'applicazione, tra due diversi casi:

- modello "flat", che consiste di un singolo array con al massimo 4G,
- modello "segmentato" che consiste di una collezione di al più 16,383 spazi di indirizzi lineari, ciascuno di 4G.

Nel modello di organizzazione della memoria "flat", il programmatore vede la memoria come un unico array di 4G. Sebbene la memoria fisica possa contenere al massimo 4G, in genere essa è molto più piccola. Il processore dovrà quindi mappare i 4G di spazio flat nello spazio di indirizzi fisico, senza che ciò sia visibile al programmatore.

Nel modello di organizzazione della memoria "segmentato", lo spazio degli indirizzi visto dal programmatore (e chiamato *spazio di indirizzi logici* è uno spazio di 64 terabytes (2^{46} bytes). Il processore mappa questo spazio di 64 terabytes nello spazio di 4G degli indirizzi fisici, ancora una volta senza che il programmatore sia a conoscenza dei dettagli del mapping. Quello che vede il programmatore è uno spazio di indirizzi logici che corrisponde ad una collezione di 16,383 sottospazi monodimensionali, ciascuno dei quali viene chiamato segmento. Un segmento è dunque costituito da un insieme di indirizzi di memoria contigui. La dimensione di un segmento può variare da un byte a un massimo di 4G.

Tipi di dato

Come già detto, la memoria può essere vista come un insieme contiguo di celle e ciascuna cella di memoria occupa un byte. Quando è necessario memorizzare dati che occupano più di un byte, qual'è l'ordine con il quale i byte che costituiscono il dato da immagazzinare vengono memorizzati?

Esistono due metodi differenti, denominati big-endian e little-endian, i cui termini derivano dai racconti di Jonathan Swift nel romanzo *I viaggi di Gulliver*. Si diceva

infatti che questi fossero i nomi di due popolazioni che abitavano nelle isole di Lilliput e Blefuscu, entrate in rivalità per il modo in cui bisognava rompere le uova: dalla punta o dal fondo. In particolare, a Lilliput una volta il figlio dell'imperatore si tagliò aprendo un uovo dall'estremità più grande, perciò da quel giorno fu ordinato di aprire le uova dall'estremità più corta, da cui *little endians*. Gli oppositori che volevano conservare la tradizione di rompere le uova dall'estremità più grande si rifugiarono a Blefuscu: *big endians*. A causa di questa differenza (Figura 3) e della sua legittimazione imperiale, era scoppiata tra le due isole una guerra sanguinosa.

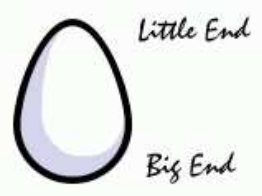
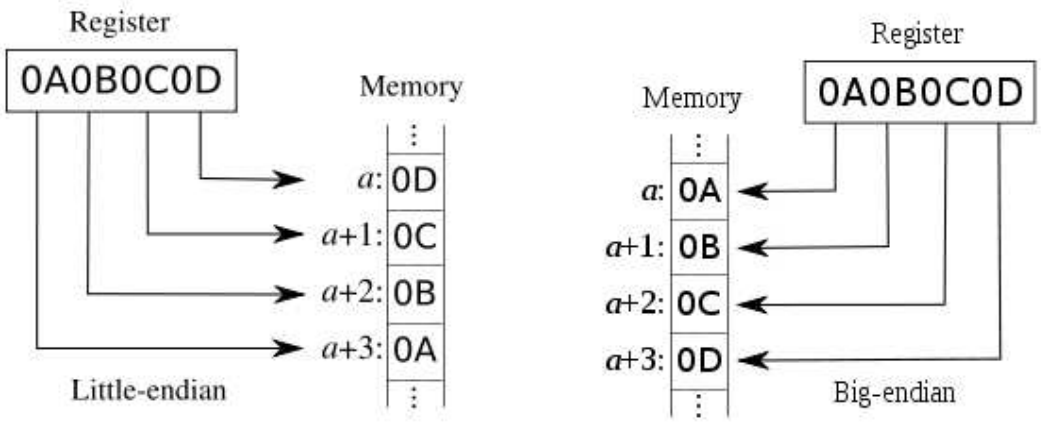


Figura 3: Causa del conflitto tra Lilliput e Blefuscu

Tornando al nostro studio dell'immagazzinare dati in memoria, in analogia alla storia fiabesca, *little-endian* è la memorizzazione che inizia dal byte meno significativo per finire col più significativo, e viene utilizzata dai processori Intel. Invece *big-endian* è la memorizzazione che inizia dal byte più significativo per finire con quello meno significativo e viene utilizzata ad esempio dai processori Motorola. Le due Figure 4 (a) e (b) mostrano alcuni esempi di memorizzazione nelle due diverse modalità.



(a) (b)

Figura 4: (a) Little-endian (b) Big-endian

Nell'80386 i tipi di dato fondamentali per gli operandi sono byte, word e doubleworld.

Un ***byte*** è costituito da otto bit contigui numerati da 0 a 7. Il bit 0 è quello meno significativo.

Una ***word*** è fatta di due byte contigui e contiene 16 bit, numerati da 0 a 15. Il bit 0 è di nuovo quello meno significativo. Il byte che contiene il bit 0 di una word è chiamato *low byte*, mentre quello che contiene il bit 15 è chiamato *high byte*. Ciascun byte di una word ha un proprio indirizzo in memoria, il più piccolo dei due indirizzi è l'indirizzo della word e contiene i bit meno significativi.

Una ***doubleword*** è fatta di due word contigue e contiene 32 bit, numerati da 0 a 31, dove come al solito lo 0 è il bit meno significativo. Analogamente al caso precedente, la word che contiene il bit 0 è chiamata *low word*, quella contenente il bit 31 è la *high word* e il più piccolo degli indirizzi delle due word coincide con l'indirizzo della doubleword.

La Figura 5 illustra i tipi di dato descritti finora, mentre la Figura 6 mostra come questi vengano arrangiati in memoria.

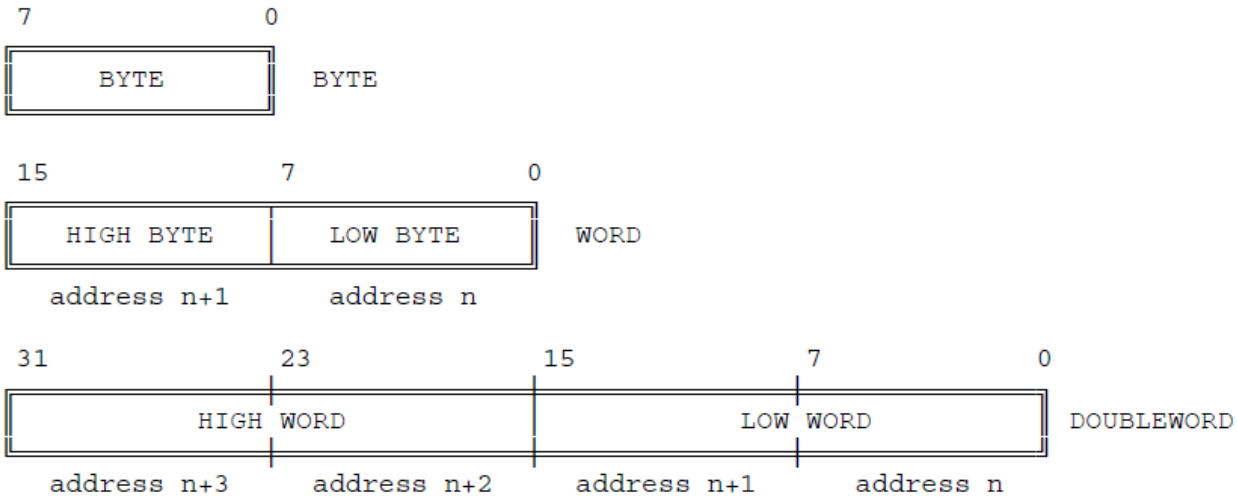


Figura 5: Tipi di dato *byte*, *word* e *doubleword*

Quelli descritti sopra sono i tpi fondamentali degli operandi. Gli operandi possono, in generale, essere interpretati in diversi modi dal processore. Vediamo quali sono le principali possibili interpretazioni supportate dall'*i*386.

Integer: un valore numerico binario con segno contenuto in una doubleword, in una word o in un byte. Tutte le operazioni che coinvolgono interi assumono la rappresentazione in complemento a 2. Il bit segno è collocato sempre nell'ultimo bit (7, 15 o 31) e vale zero per gli interi positivi, uno per i negativi. Quindi con un byte posso rappresentare interi nel range $-128, +127$, con una word da $-32,768$ a $+32,767$, con una doubleword il range è $-2^{31}, +2^{31} - 1$.

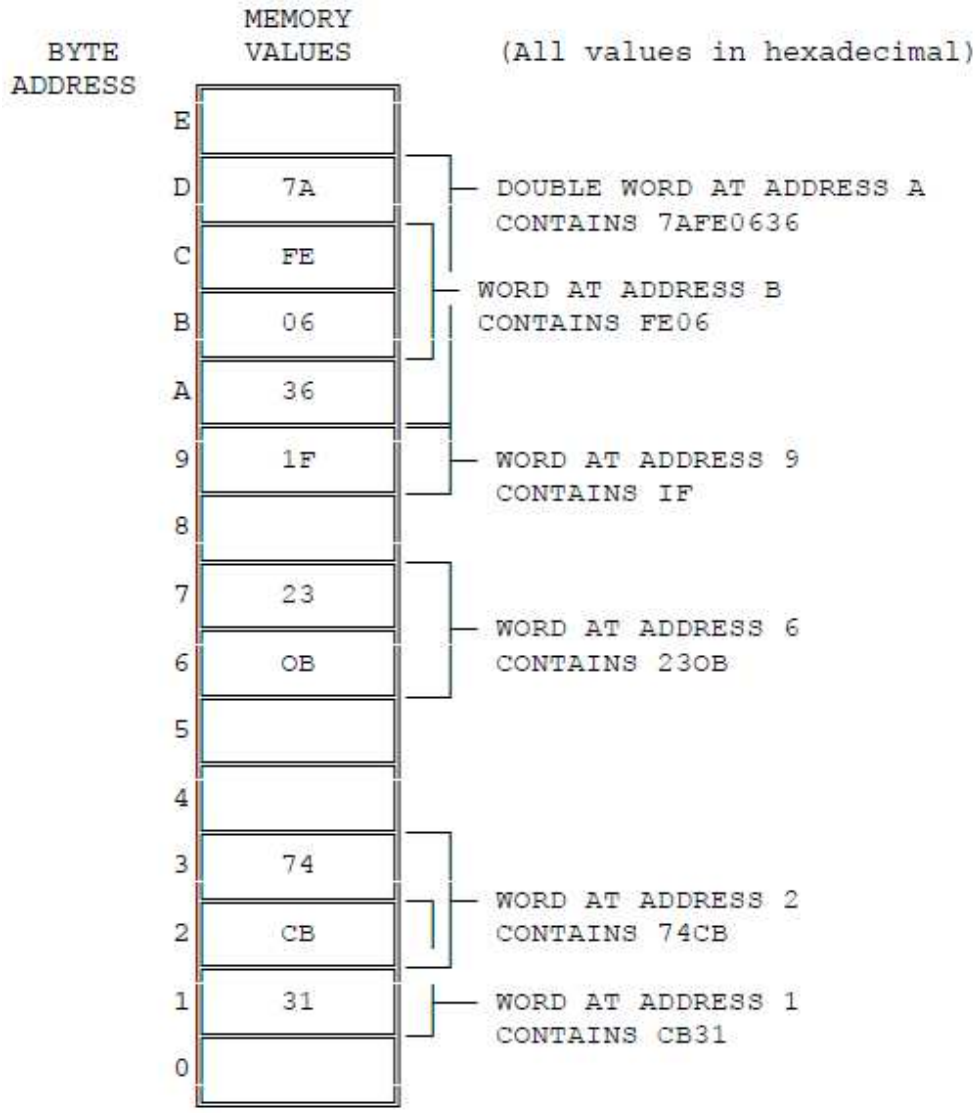


Figura 6: Tipi di dato *byte*, *word* e *doubleword* in memoria

Ordinal: Un valore numerico binario senza segno contenuto in una doubleword, in una word o in un byte. Nei tre casi, i range sono, rispettivamente, $0 - 255$, $65,535$ e $0, 2^{32} - 1$.

Near Pointer: Un indirizzo logico a 32 bit.

String: Una sequenza contigua di of bytes, words, o doublewords. Una stringa può contenere da 0 a $(2^{32} - 1)$ bytes (4G).

Bit field: Una sequenza contigua di bit, che comincia alla posizione iniziale di un byte e può contenere al massimo 32 bit.

Registri

Il processore 80386 contiene in tutto 16 registri che il programmatore può utilizzare. Tali registri possono essere raggruppati in tre principali categorie:

- **Registri ad uso generico**. Sono 8 registri di 32 bit utilizzati per scopi generali, tra cui soprattutto quello di contenere operandi per operazioni aritmetiche e logiche.
- **Registri segmento**. Sono 6 registri speciali che permettono ai progettisti di sistemi software di scegliere tra organizzazione di memoria flat o segmentata, determinando, in un certo istante di tempo, quali segmenti di memoria sono correntemente indirizzabili.
- **Registri istruzione e di stato**. Sono due registri usati per gestire e/o modificare alcuni aspetti relativi allo stato del processore *i*386.

Ci concentreremo solo sui registri ad uso generico, che sono quelli che andremo a utilizzare per programmare in Assembly. Come illustrato in Figura 7, gli 8 registri ad uso generico a 32 bit a disposizione sono: EAX, EBX, ECX, EDX, EBP, ESP, ESI, ed EDI. Questi registri possono essere utilizzati in modo interscambiabile per contenere operandi di operazioni logiche e aritmetiche, come anche per contenere operandi relativi a operazioni di indirizzamento (fatta eccezione per ESP in quest'ultimo caso).

Osservando bene la Figura 7, ci si accorgerà che la word che occupa i 16 bit meno significativi di ciascuno dei registri generali ha un proprio nome e può essere considerata un'unità a sè stante. Questa caratteristica è particolarmente utile sia per gestire eventuali dati a 16 bit, che per garantire compatibilità con i processori 8086 e 80286. Tali 8 registri di 16 bit ciascuno prendono i nomi: AX, BX, CX, DX, BP, SP, SI, e DI.

Analogamente, ciascuno dei due byte che compone uno dei quattro registri a 16 bit AX, BX, CX, e DX, ha un proprio nome e può essere utilizzato come registro a sè per manipolare caratteri e altri dati a 8 bit. In particolare, facendo riferimento al byte più significativo, abbiamo: AH, BH, CH e DH; considerando, invece, il byte meno significativo, abbiamo: AL, BL, CL e DL.

Oltre alle operazioni di indirizzamento e di calcolo, alcuni dei registri generali possono essere utilizzati per delle funzioni dedicate. Nell'architettura dell'*i*386, i registri utilizzati a tal fine sono scelti implicitamente. Questo fa sì che le istruzioni possano essere

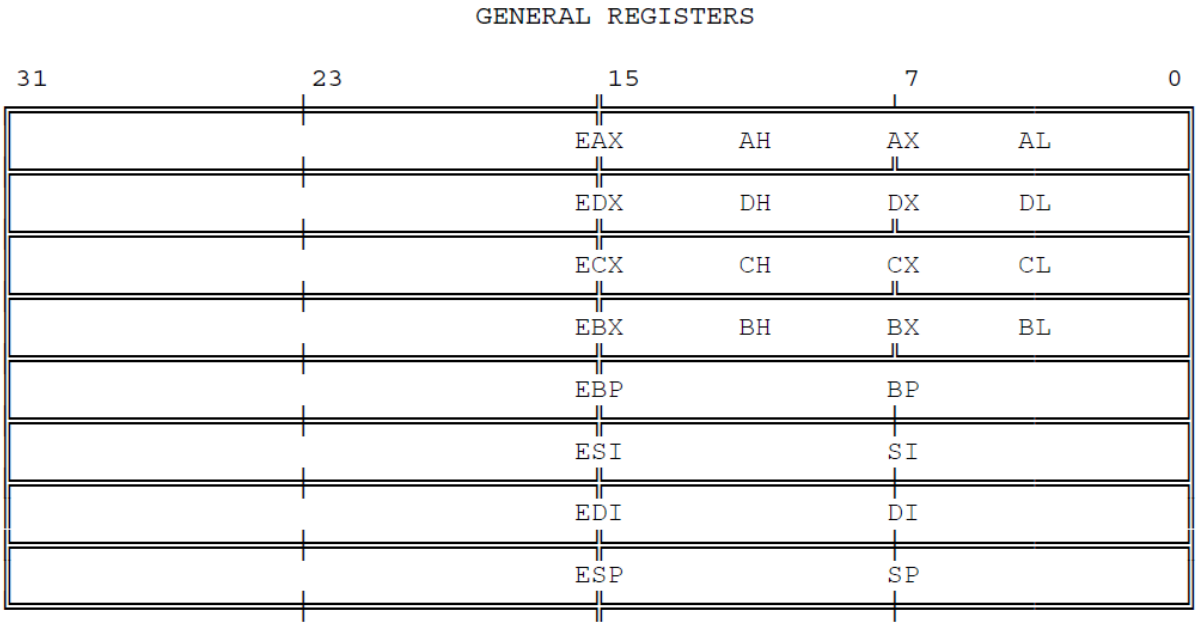


Figura 7: Registri generali dell'*i*386

codificate in modo più compatto. Le istruzioni che utilizzano specifici registri includono: divisione a precisione doppia e multipla, I/O, istruzioni che riguardano stringhe, loop, istruzioni che coinvolgono lo stack. A titolo di esempio, più avanti vedremo che per gestire lo stack si fa uso dei due registri ESP ed EBP.

Noi utilizzeremo la modalità protected mode – flat model dell'*i*386. L'uso del modo protetto richiede un estensivo intervento del S.O., ad esempio MS-DOS non è in grado di gestirlo, Linux lo ha sempre supportato e Windows NT è stato il primo dei sistemi operativi Microsoft a gestirlo. In questa modalità i registri segmento sono considerati parte del S.O. e non sono modificabili dai programmi applicativi, possono essere considerati *protetti*. In particolare, il programma "vede" 4G di memoria in modo trasparente all'utente, ovvero, quale porzione di memoria fisica stia effettivamente usando è nascosto all'utente.

Assemblatore, Linker e Debugger

Il processo di produzione di uno strumento *software* si articola in diverse fasi:

1. Analisi del problema da risolvere e sua scomposizione in sottoproblemi.
2. Progettazione di un opportuno programma.
3. Scrittura del programma.

La fase che ci interessa maggiormente in questo momento è la numero 3 e, in particolare, ci interessa capire in maggiore dettaglio quali sono i passaggi attraverso i quali l'uomo riesce a far sì che la macchina risolva in modo automatico il problema di partenza.

Un programma, per come il calcolatore elettronico può interpretarlo, è una sequenza di istruzioni. Il calcolatore è in grado però di interpretare ed eseguire istruzioni molto semplici, che tipicamente sono codificate in linguaggio macchina. Ovviamente, il programmatore però non scriverà mai il suo programma in linguaggio macchina, sia perchè questo potrebbe comportare numerosi errori, sia perchè dovrebbe in tal caso parlare la stessa lingua della macchina, ovvero, in binario.

Invece i programmatori sviluppano i loro programmi, che implementano algoritmi di risoluzione talvolta anche molto complessi, in quelli che vengono chiamati linguaggi ad alto livello, come ad esempio Java e C++. In genere, sebbene ciò non necessariamente costituisca una regola, tanto più semplice, intuitiva e sintetica risulta per il programmatore la scrittura di un programma, altrettanto meno efficiente sarà la sua esecuzione da parte del calcolatore. Questo dipende sempre dal fatto che l'uomo e il calcolatore parlano due lingue molto diverse e, tanto più un programma è facilmente comprensibile dall'uomo, tanto più vuol dire che è lontano dalla lingua che parla la macchina.

La forma più rudimentale di linguaggio di programmazione in cui l'uomo possa cimentarsi senza essere troppo lontano da ciò che il calcolatore debba interpretare, prende il nome di Assembly. In particolare, utilizzando l'Assembly è possibile scrivere un programma fatto di istruzioni in forma simbolica, ciascuna delle quali corrisponde ad una specifica istruzione in linguaggio macchina. Questa corrispondenza stretta tra istruzione in forma simbolica e istruzione in binario è garanzia dell'efficienza notevolmente superiore rispetto alla scrittura di un linguaggio ad alto livello. Vedremo, infatti, come una singola istruzione di un linguaggio ad alto livello corrisponda talvolta a un elevato numero di istruzioni in Assembly (e, quindi, in linguaggio macchina).

Sebbene le istruzioni in Assembly siano molto vicine a ciò che un calcolatore può interpretare, esse tuttavia devono essere tradotte in linguaggio macchina per poter essere

eseguite. Compito dell'assemblatore, o assembler, è proprio quello di fungere da traduttore tra ciò che il programmatore ha scritto e ciò che il calcolatore può interpretare. L'assemblatore non è altro che un programma che si comporta come il *compilatore* dei linguaggi ad alto livello, ovvero, individua eventuali errori presenti nel modulo sorgente e lo traduce in linguaggio macchina.

Il modulo sorgente, in genere stilato attraverso un opportuno editor di testo, rappresenta l'input dell'assembler, che restituisce in output un modulo oggetto che solitamente viene inserito in un file. La versione eseguibile del modulo oggetto viene costruita dal programma linker, che mette insieme i diversi moduli oggetto componenti nel caso in cui siano più di uno. Un modulo oggetto in generale può contenere riferimenti a procedure esterne, variabili interne o esterne, il codice del programma, eventuali informazioni di debug e/o utili per facilitare il linking.

Infine, spesso risulta conveniente utilizzare un opportuno programma debugger, che consente di controllare l'esecuzione del programma prodotto. Questa azione di controllo viene esercitata sia nel senso di stabilire e/o modificare le modalità con cui far procedere l'esecuzione del programma controllato, sia nel senso di verificare la correttezza delle operazioni che esso esegue. Ad esempio, il debugger può agevolare nell'individuazione di eventuali cause di risultati errati dovuti ad una inesatta definizione degli algoritmi oppure a sviste nella scrittura che non hanno dato luogo ad errori sintattici e non sono state, pertanto, rilevate in fase di assemblaggio.

In Figura 8 viene illustrato un workflow relativo ai programmi appena descritti. Si osservi che le operazioni che portano dalla scrittura all'esecuzione di un programma funzionante in modo corretto prevedono percorsi ciclici che spesso possono essere intrapresi anche più volte prima di raggiungere l'obiettivo. Infatti, può essere necessario tornare alla fase di editing per apportare modifiche al modulo sorgente sia durante la fase di traduzione che durante l'esecuzione, per esempio, ed anche il linker può in alcuni casi segnalare eventuali errori.

Istruzioni

Le istruzioni possono essere rappresentate in forma simbolica come costituite dai seguenti campi:

- un'etichetta, o *label*, ovvero un identificativo seguito dai due punti;
- un prefisso, che è opzionale e in genere è un nome riservato;
- uno mnemonico, che è un nome riservato per identificare la funzione di una certa istruzione (ovvero l'operazione che deve compiere);

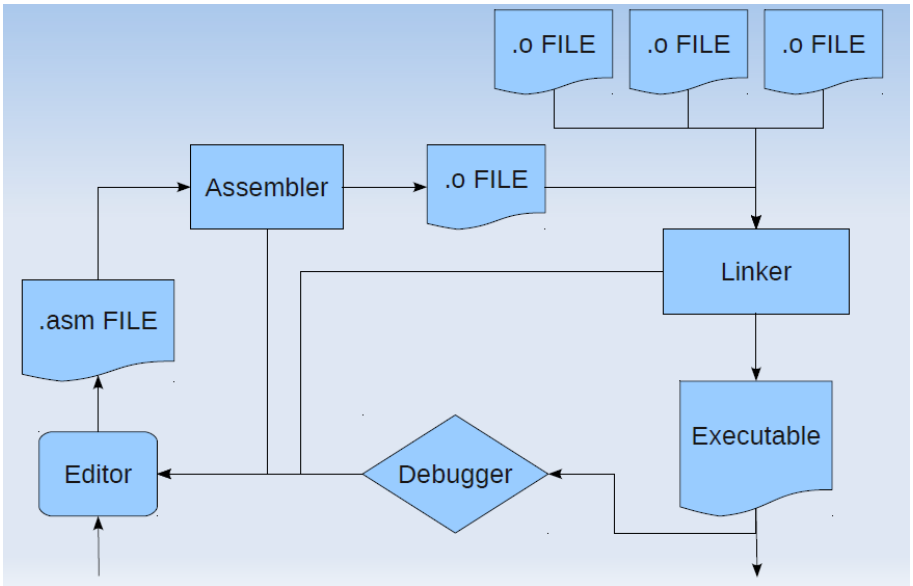


Figura 8: Workflow

- gli (eventuali) operandi, che nella CPU considerata possono essere da zero a tre e sono separati da virgole.

Gli operandi di una certa istruzione possono essere memorizzati all'interno dell'istruzione stessa, nel caso di operandi immediati, in un registro della CPU, in una locazione di memoria. É possibile anche gestire opportunamente lo scambio di dati con l'I/O, che tuttavia richiede una trattazione più complessa.

Segue un esempio di istruzione.

Esempio 1 Consideriamo la seguente istruzione:
`LOADREG: MOV EAX, SUBTOTAL`
In particolare, `LOADREG` è l'etichetta, `MOV` è lo mnemonico, in `EAX` è memorizzato l'operando destinazione, in cui cioè verrà inserito il risultato, `SUBTOTAL` contiene l'operando sorgente, da cui viene prelevato il contenuto. L'effetto dell'operazione `MOV`, come vedremo più avanti in maggiore dettaglio, è quello di spostare il contenuto dalla sorgente alla destinazione.

L'assemblatore NASM

Il *Netwide Assembler*, NASM, è un assembler 80×86 progettato sulla base di principi quali la portabilità e la modularità. Il NASM supporta numerosi formati di file oggetto, tra cui Linux e *BSD a.out, ELF, COFF, Mach−O, Microsoft 16−bit OBJ, Win32 and Win64. Può anche generare in output file binari piani. La sua sintassi è progettata per essere semplice e facile da capire, simile a quella Intel ma meno complessa. Il NASM supporta tutte le estensioni architetturali note ×86 e fornisce supporto anche per le macro. La pagina ufficiale di NASM è `http://www.nasm.us/`.

Di seguito vengono spiegate le principali modalità di installazione e utilizzo.

Installazione sotto Windows

Una volta ottenuto l'archivio appropriato per NASM, ad esempio `nasm−XXX−win32.zip` (dove `XXX` indica il numero di versione di NASM contenuto nell'archivio), scompattarlo nella propria directory (ad esempio `c:\NASM`). L'archivio conterrà una serie di file eseguibili: `nasm.exe` è l'eseguibile dell'assemblatore NASM, `ndisasm.exe` è quello del disassemblatore NDISASM; inoltre potranno esserci eventuali programmi di utilità per gestire il formato di file RDOFF.

Solo l'eseguibile di NASM dovrà essere lanciato, quindi è necessario copiare il file `nasm.exe` in una directory del proprio PATH, o in alternativa modificare `autoexec.bat` per aggiungere la directory NASM al PATH (per fare questo in Windows XP, andare su Start > Pannello di controllo > sistema > Avanzate > variabili d'ambiente; queste istruzioni possono andar bene anche per altre versioni di Windows).

Installazione sotto Unix

Una volta ottenuto l'archivio appropriato per NASM, `nasm−XXX.tar.gz` (dove `XXX` indica il numero di versione di NASM contenuto nell'archivio), scompattarlo in una directory, ad esempio `/usr/local/src`. Quando l'archivio verrà estratto, creerà la sua sottodirectory `nasm-XXX`.

NASM è un pacchetto auto-configurante: dopo essere stato scompattato, basterà fare `cd` nella directory in cui è stato scompattato e lanciare `./configure`. In tal modo verrà trovato il migliore compilatore C da utilizzare per il build di NASM e per impostare opportunamente i Makefile. Una volta che NASM è stato configurato, digitando `make` verranno compilati i file binari NASM e NDISASM, mentre attraverso `make install` li si potrà installare in `/usr/local/bin` e, allo stesso modo, i manuali `nasm.1` e `ndisasm.1` verranno installati in `/usr/local/man/man1`.

Nei sistemi Debian-based, in alternativa a quanto sopra, si può utilizzare direttamente il comando: `sudo apt-get install nasm`.

Utilizzare NASM

Per assemblare un file, va utilizzato un comando della forma:

```
nasm −f <format> <filename> [−o <output>]
```

L'opzione `-f <format>` specifica il formato del file oggetto, che di solito è `elf32` per le versioni più recenti di Linux ed è `obj` per Windows. L'output di default è `<filename>.o` per Linux, `<filename>.obj` per Windows. Ad esempio, il comando:

```
nasm −f elf32 myfile.asm
```

assembla il file `myfile.asm` in un file oggetto di tipo ELF (Executable and Linking Format), il formato binario standard per Linux, che si chiamerà `myfile.o`.

Analogamente, il comando:

```
nasm −f bin myfile.asm −o myfile.com
```

assembla `myfile.asm` in un file binario raw `myfile.com`.

Consideriamo adesso il seguente comando:

```
nasm -f elf32 -g -F stabs <filename>.asm
```

L'opzione `-g` serve per abilitare le informazioni di debug, mentre l'opzione `-F <format>` specifica il formato delle informazioni di debug. Oltre al formato STABS, più in voga quando anche sotto Linux i file oggetto erano nel formato OBJ piuttosto che il più recente ELF, un altro formato più recente per i file di debug è il formato DWARF.

L'opzione `-v` consente di conoscere la versione di NASM che si sta utilizzando, digitando `nasm -v`.

Altre due opzioni utili sono `-Z`, che consente di inviare ad un file eventuali errori generati a partire da un file sorgente e `-s`, che serve ad inviare invece tali errori ad `stdout`. Seguono due esempi di utilizzo.

```
nasm −Z myfile.err −f obj myfile.asm
```

```
nasm −s −f obj myfile.asm | more
```

In particolare, `myfile.err` è il file dove verranno stampati gli errori relativi al file `myfile.asm` nel primo caso, mentre nel secondo caso il file `myfile.asm` verrà assemblato e i suoi output saranno impilati nel programma `more`.

Dopo che il file oggetto viene creato, è necessario effettuare il linking per poter generare l'eseguibile. Linux fornisce come linker nativo `ld` (abbreviazione di *load*). Ad esempio, il comando:

```
ld -m elf_i386 <filename>.o -o <filename>
```

permette di specificare il tipo di emulazione, attraverso l'opzione `-m`, e di generare l'eseguibile `<filename>`. Invece sotto Windows si può utilizzare `alink`:

```
alink.exe <filename>.obj -o <filename>.exe
```

Dopo aver assemblato un file, è anche possibile *disassemblarlo*, per riottenere dal file eseguibile (binario) un file in Assembly. Vediamo come:

```
objdump -d <filename>
```

dove l'opzione `-d` serve per abilitare il disassemblatore. Usato in questo modo, `objdump` disassembla il programma e mostra anche le istruzioni nel linguaggio macchina vero e proprio, con gli indirizzi di memoria virtuale che verrebbero utilizzati durante il funzionamento.

Per richiedere espressamente di disassemblare utilizzando una notazione Intel:

```
objdump -d -M intel <filename>
```

Scrivere programmi in Assembly

Per scrivere un programma in Assembly e poi utilizzare NASM per assemblarlo è necessario seguire delle precise regole nella stesura del listato, in modo tale che NASM possa interpretarlo correttamente. Ad esempio, i commenti vengono sempre preceduti da un `;` mentre il listato può essere suddiviso in tre specifiche sezioni:

1) `.data`

   Questa sezione contiene la definizione dei dati inizializzati, ossia i dati che hanno un valore prima che il programma venga eseguito. I valori sono scritti nell'eseguibile insieme al codice e assegnati dal loader, nessun ciclo macchina viene speso per l'inizializzazione e la dimensione dell'eseguibile cresce all'aumentare del numero di dati inizializzati. La direttiva `.data` avverte l'assemblatore dell'inizio della zona usata per descrivere l'uso della memoria.

2) `.bss` (*Block Started by Symbol*)

   Questa sezione contiene la definizione dei dati non inizializzati e, pertanto, può anche essere vuota (nel qual caso specificarla o non specificarla non influisce sulla dimensione del file eseguibile).

3) `.text`

   Qui sono contenute le istruzioni che compongono il programma. In questa sezione non vanno definiti dati e vengono utilizzati simboli di dati definiti precdentemente o simboli che identifcano label (ad esempio, per gestire i salti).

Sebbene l'ordine con cui le tre sezioni si susseguono non sia importante, per convenzione di solito si segue l'ordine indicato sopra.

Nel seguito di questa dispensa, assumeremo di usare NASM sotto una distribuzione Debian di Linux (alcuni approfondimenti per Windows sono disponibili in Appendice).

Le etichette (o *labels*) sono dei "segnaposti" che identificano una riga di codice che ha un significato speciale (ad esempio, una label è rappresentativa di una operazione o di un gruppo di operazioni) e possono essere richiamate in diverse parti del programma. Sono *case sensitive*. Un'etichetta deve cominciare per lettera oppure con il simbolo "_" e, quando viene definita, deve essere seguita da ":" (dove viene referenziata invece i due punti non vanno usati). Dopo i due punti vanno inserite le operazioni associate a quella particolare etichetta. Se si vuole rendere una label visibile all'esterno deve essere dichiarata *global*. Qualunque programma in Assembly deve presentare un'etichetta `_start`, che individua l'indirizzo della prima istruzione del programma. L'etichetta `_start` deve essere resa pubblica, perché `ld` deve sapere da che parte si comincia (soprattutto quando più file oggetto devono essere fusi in un unico file eseguibile).

In Figura 9 è mostrato il listato di un programma molto semplice.

```
1.  ;
2.  ; Primo esempio semplice
3.  ;
4.  section .data
5.  section .text
6.  global _start
7.  ;
8.  _start:
9.      mov eax, 1
10.     mov ebx, 7
11.     int 0x80
```

Figura 9: Esempio di programma in Assembly

Le prime tre righe e le righe 3 e 7 sono dei commenti. La quarta riga contiene la direttiva che indica l'inizio della zona usata per descrivere l'uso della memoria (che in questo caso non viene usata affatto); la quinta riga avverte l'assemblatore dell'inizio del codice del programma. La riga 6 serve ad avvisare l'assemblatore che il simbolo rappresentato dall'etichetta denominata `_start` individua l'indirizzo dell'istruzione iniziale da rendere pubblica, pertanto deve rimanere rintracciabile nel file oggetto generato dalla compilazione. L'ottava riga dichiara l'etichetta `_start`.

Prima di spiegare in dettaglio le ultime tre istruzioni del programma illustrato in Figura 9, dobbiamo aprire una parentesi sulle chiamate a funzioni del Sistema Operativo.

Per interagire con le componenti della macchina, è necessario effettuare la richiesta di un servizio al Sistema Operativo, che offre un'apposita interfaccia per richiamare determinate funzioni attraverso l'utilizzo delle *interruzioni*. É presente un vettore delle interruzioni nelle prime 1,024 locazioni di memoria cui non è possibile accedere nè in lettura nè in scrittura. Esiste un gestore dei servizi che si chiama *service dispatcher* e che si trova all'indirizzo 80_{16} (Figura 10). Per comunicare con il *service dispatcher* e, dunque, per interfacciarsi con il Sistema Operativo, Assembly mette a disposizione il comando `int`.

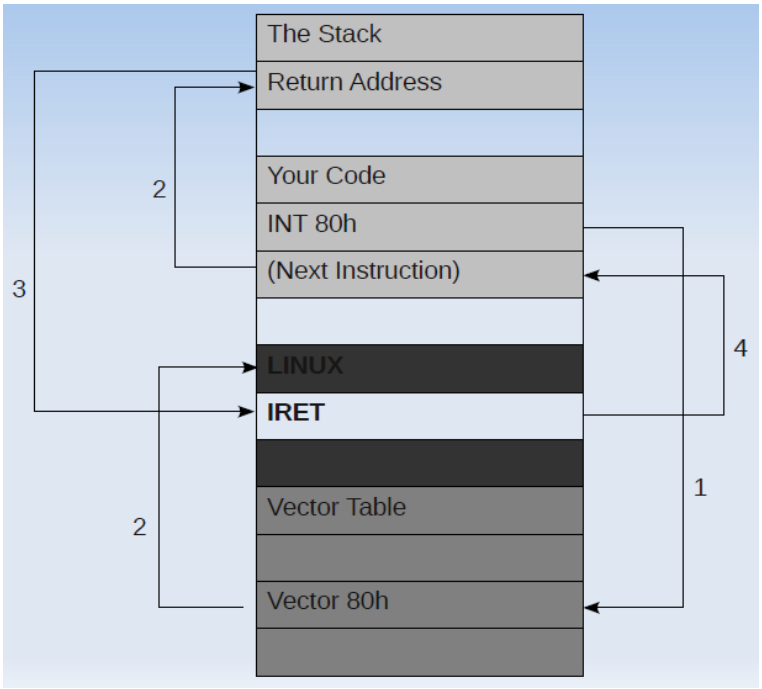


Figura 10: Gestione delle interruzioni

In particolare, è necessario prima di tutto specificare quale dei circa 200 servizi a disposizione si vuole richiedere. Per fare ciò, bisogna memorizzare il numero del servizio

desiderato all'interno del registro EAX. Se il servizio richiede altri parametri, questi andranno specificati in appositi registri, a seconda del servizio in considerazione.

Tornando al nostro esempio di programma, nella riga 9 si assegna il valore uno al registro EAX, per individuare una particolare funzione del Sistema Operativo che corrisponde all'EXIT dal programma. Questa funzione richiede che il parametro rappresentato dal valore di uscita venga collocato nel registro EBX (il valore 0 indica uscita senza errori). Nella decima riga del programma, assegnamo il valore sette al registro EBX. Infine, nell'ultima riga, si esegue un'interruzione all'indirizzo 80_{16} per effettuare la chiamata a funzione desiderata.

Il programma generato si limita a chiamare una funzione del sistema operativo, con la quale conclude il suo lavoro restituendo il valore numerico sette. Lo si può verificare ispezionando il parametro $? della shell[1], dopo aver lanciato l'eseguibile del programma, come illustato di seguito.

```
nasm -f elf32 -o echo7.o echo7.asm
ld -o echo7 echo7.o
echo $?
```

In generale, è buona norma terminare i programmi che scriveremo in Assembly con le seguenti istruzioni:

```
mov ebx,0; exit code, 0=normal
mov eax,1; exit command to kernel
int 0x80; interrupt 80 hex, call kernel
```

in modo da esplicitare l'uscita, dicendo al kernel che il programma è terminato e può essere rimosso dallo scheduling di esecuzione (anche se un buon Sistema Operativo termina comunque un programma in corrispondenza dell'ultima istruzione dello stesso).

Quando si programma in Assembly spesso è conveniente utilizzare un debugger, ovvero uno strumento che permette di eseguire passo per passo il proprio programma, consentendo di verificare lo stato dei registri ed eventualmente della memoria. Infatti, con un linguaggio assemblatore, operazioni "semplici" come l'emissione di informazioni attraverso lo schermo diventano invece molto complicate.

Nei sistemi GNU è disponibile GDB (GNU debugger). Per capire come utilizzarlo, si modofichi il programmino analizzato come illustrato in Figura 11.

[1]La shell di Linux ha la variabile d'ambiente speciale $? che viene settata al codice d'uscita dell'ultimo programma eseguito.

```
 1.  ;
 2.  ; Primo esempio semplice
 3.  ;
 4.  section .data
 5.  section .text
 6.  global _start
 7.  ;
 8.  _start:
 9.      mov eax, 1
10.  bp1:
11.      mov ebx, 7
12.  bp2:
13.      int 0x80
```

Figura 11: Modifiche per utilizzare un debugger (righe 10 e 12)

In particolare, sono state aggiunte due etichette sulle righe 10 e 12 opportunamente collocate tra le istruzioni che si traducono in codici del linguaggio macchina. I nomi delle etichette non sono importanti, li abbiamo scelti in modo da ricordare la parola *breakpoint*. Riassemblando (meglio se si usano le opzioni -g e -F viste prima) e linkando nuovamente il file, è possibile avviarlo all'interno di GDB attraverso il comando `gdb echo7`, che permetterà di entrare in una modalità da cui è possibile inserire dei comandi in modo interattivo all'interno della shell. Se si vogliono aprire più finestre contemporaneamente si può usare `gdb -tui echo7`.

La prima cosa da fare è associare dei breakpoint alle etichette aggiunte sulle righe 10 e 12 del sorgente, per stabilire dove l'esecuzione del programma deve essere sospesa automaticamente. Lo faremo digitando prima `break bp1` e poi `break bp2`, rispettivamente (Figura 12).

Una volta fissati gli stop, si può avviare il programma digitando `run`. Il programma verrà sospeso in corrispondenza del primo dei due breakpoint, come illustrato in Figura 13.

Digitando `info registers` si potranno ispezionare i registri, verificando che il registro EAX contiene il valore uno, come dovrebbe effettivamente essere a questo punto dell'esecuzione. Per far proseguire il programma fino al prossimo stop si usa il comando `continue`. Ispezionando nuovamente i registri si vedrà che a questo punto il registro EBX risulta impostato con il valore previsto (ovvero il valore 7). Si può dunque lascire concludere il programma (`continue`) e terminare l'attività con GDB attraverso il comando `quit`. In generale, per eseguire il programma con GDB eseguendo un'istruzione alla volta si può usare il comando `stepi`.

```
$nasm -f elf32 -g -F dwarf echo7.asm -o echo7.o
$ld echo7.o -o echo7
$gdb echo7
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/simona/scrivania/echo7...done.
(gdb) break bp1
Breakpoint 1 at 0x8048065: file echo7.asm, line 12.
(gdb) break bp2
Breakpoint 2 at 0x804806a: file echo7.asm, line 14.
(gdb)
```

Figura 12: Fissare i breakpoint

Esistono diversi programmi frontali che sono basati su GDB ma mettono a disposizione un'interfaccia grafica che consente di tenere sotto controllo più cose, simultaneamente. Tra questi, ricordiamo ad esempio DDD e KDBG. Per utilizzarli, è sufficiente installarli con ad esempio `sudo apt-get install ddd` e poi lanciare l'eseguibile analogamente a quanto fatto per GDB (e.g. `ddd echo7`).

Variabili

All'interno della sezione `.data` vengono dichiarate le variabili inizializzate (*defined*) per le quali è necessario specificare quanto spazio occupano:

`db`	8 bit	(1 byte)
`dw`	16 bit	(1 word)
`dd`	32 bit	(1 doubleword)
`dq`	64 bit	(1 quadword, ovvero 2 doubleword)

Anche per le variabili non inizializzate (*reserved*) dichiarate nella sezione `.bss` bisogna specificare la dimensione al momento della dichiarazione:

`resb`	8 bit	(1 byte)
`resw`	16 bit	(1 word)
`resd`	32 bit	(1 doubleword)
`resq`	64 bit	(1 quadword, ovvero 2 doubleword)

Una stringa è rappresentata da una sequenza di caratteri in memoria, e può essere definita associando un'etichetta a dove inizia la stringa. Le stringhe possono essere

```
(gdb) run
Starting program: /home/simona/scrivania/echo7

Breakpoint 1, bp1 () at echo7.asm:12
12              mov ebx, 7
(gdb) info register
eax            0x1      1
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbffff420       0xbffff420
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048065        0x8048065 <bp1>
eflags         0x212    [ AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
(gdb)
```

Figura 13: Stop al primo breakpoint bp1

racchiuse tra apici e/o virgolette, ed è possibile concatenarle interponendo una virgola tra due stringhe. Un numero concatenato verrà interpretato come codice ASCII.

Per definire una costante si utilizza la direttiva `equ`, mentre il simbolo `$` è un token speciale che indica *here*, ovvero, segna il punto dove l'assemblatore è arrivato.

Segue qualche esempio di dichiarazione.

```
section .data
    var1   db   10; dichiariamo la variabile var1 che occupa 1 byte ed è
    ;               inizializzata con il valore 10
    NumMesi   equ   12; dichiariamo una costante
    Message   db   ‘‘ciao mondo!’’,10; dichiariamo una stringa e la
    ;               concateniamo con il codice ASCII del numero 10
    MsgLen   equ   $-Message
section .bss
    var2   resd; dichiariamo la variabile non inizializzata var2 che occupa
    ;            1 doubleword
```

Programmare in Assembly

Iniziamo adesso ad entrare più in dettaglio sulla programmazione in assembly. A tal fine, ci servirà studiare le operazioni fondamentali che ci permetteranno di scrivere i nostri programmi.

Una prima operazione che già abbiamo accennato e che adesso andremo ad analizzare meglio è la `MOV`. La `MOV` è un'operazione a due operandi, e serve a trasferire il contenuto del secondo operando (sorgente) nel primo (destinazione). Non tutte le coppie di operandi però sono lecite: ad esempio, due variabili non possono essere utilizzate contemporaneamente come operandi della `MOV`. Infatti questa operazione non può essere utilizzata per effettuare spostamenti da *memoria* a *memoria*.

Vediamo quali sono i trasferimenti leciti:

- da *registro* a *registro*

```
mov di, ax; 16 bit
mov ecx, edx; 32 bit
```

- da *registro* a *memoria*

```
mov [Msg], eax
mov [eax], ebx
```

- da *memoria* a *registro*

```
mov eax, [ebx]
mov eax, [Msg]; spostamento dati (contenuto di Msg)
mov eax, Msg; spostamento indirizzi (indirizzo di Msg)
```

- da *immediato* a *registro*

```
mov eax, 42h
```

- da *immediato* a *memoria*

```
mov [eax], byte 42h
mov [eax], dword Msg
```

Va osservato che il nome di una variabile rappresenta sempre l'indirizzo della cella in memoria dove inizia quella variabile. Per accedere a una cella di memoria, si utilizzano le parentesi quadre. Ad esempio `[eax]` fa sì che si acceda alla cella di memoria che si

trova all'indirizzo contenuto dentro il registro EAX. Ricordiamo che stiamo assumendo che le celle di memoria siano ad 8 bit mentre gli indirizzi siano a 32 bit. Per prelevare il contenuto di una variabile va dunque usato `[nome variabile]`, ad esempio `[Msg]`.

Osserviamo che quando si effettua uno spostamento *verso* un registro, la dimensione di ciò che si sposta è dettata dalla dimensione del registro destinazione. In particolare, il numero di bit della sorgente deve essere inferiore alla dimensione del registro destinazione. Analogamente, quando si effettua un trasferimento *verso* la memoria, bisogna specificare (se ciò non è chiaro) quanti byte bisognerà spostare in memoria. In particolare se, in tal caso, la sorgente è un registro, la dimensione dei dati è implicitamente nota. Altrimenti sarà necessario esplicitarla utilizzando `byte`, `word`, `dword`, `qword`.

Vediamo qualche esempio di trasferimento non consentito:

```
mov ax, ebx
mov eax, bx
```

In entrambi i casi, le dimensioni dei registri sorgente e destinazione non coincidono.

```
mov [eax], [ebx]
```

In questo caso c'è un doppio passaggio in memoria, che non è consentito.

```
mov Msg, eax
```

Msg è un indirizzo, non un *contenitore*. Il seguente esempio conclude la trattazione della `MOV`.

Esempio 2 Consideriamo il seguente programma in Assembly.

```
section .data
    temp dd 1612
section .text
    global _start
    _start:
        mov eax, [temp]
        mov [temp], dword 12
        mov ecx, temp
        mov edx, [temp]
    exit:
        mov ebx, 0
```

```
mov eax, 1
int 0x80
```

Inserendo un breakpoint su `exit`, la Figura 14 illustra lo stato dei registri quando l'esecuzione arriva in tale punto del programma.

```
eax            0x64c    1612
ecx            0x80490ac        134516908
edx            0xc      12
ebx            0x0      0
esp            0xbffff3b0       0xbffff3b0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804809f        0x804809f <exit>
eflags         0x212    [ AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

Figura 14: Esempi di utilizzo di `MOV`

Stampa su video (write to stdout)

Per effettuare la stampa su video, in generale, è necessario come già accennato richiamare un'interruzione. In particolare, nel primo programma che abbiamo analizzato ci limitavamo a stampare un numero che avevamo scritto nel registro EBX, sovrascrivendo il parametro della funzione EXIT dal programma.

Supponiamo di voler stampare su video una stringa, che si trova dentro la variabile `Msg`. In tal caso, il servizio da richiedere al Sistema Operativo è il numero 4 e riceve tre argomenti, come illustrato nel seguente listato.

```
mov edx, Len; arg3, length of string to print
mov ecx, Msg; arg2, pointer to string
mov ebx, 1; arg1, where to write, screen
mov eax, 4; write sysout command to int 80h
int 0x80 (80h); interrupt 80 hex, call kernel
```

Siamo pronti adesso per scrivere, finalmente, il nostro programma *Hello world* in Assembly, illustrato in Figura 15.

```
1.  ;
2.  ; Hello world program
3.  ;
4.  section .bss; bss section
5.  section .data; data section
6.      Msg:  db 'Hello World!', 10; the string
7.      Len:  equ $ - Msg
8.  section .text; code section
9.      global _start; make label available to linker
10.     _start:
11.         mov edx, Len; arg3, length of string to print
12.         mov ecx, Msg; arg2, pointer to string
13.         mov ebx, 1; arg1, where to write, screen
14.         mov eax, 4; write sysout command to int 80h
15.         int 0x80; interrupt 80 hex, call kernel
16.     exit:
17.         mov ebx, 0; exit code, 0=normal
18.         mov eax, 1; exit command to kernel
19.         int 0x80; interrupt 80 hex, call kernel
```

Figura 15: Programma *Hello World*

Le prime tre righe sono dei commenti. La quarta e quinta riga definiscono le sezioni dei dati non inizializzati (vuota) e inizializzati. Tra questi ultimi, abbiamo una variabile `Msg` che contiene la stringa da stampare, concatenata con il numero 10 che rappresenta il codice ASCII dello '$\backslash n$'. Invece che il numero 10 avremmo potuto usare '$\backslash n$', usando come apici quelli che si ottengono digitando contemporaneamente sulla tastiera il tasto `Alt Gr` e il tasto apice.

La seconda dichiarazione presente nella sezione `.data` è una costante `Len`, che rappresenta la lunghezza della stringa intesa come numero di byte che questa occupa in memoria. Infatti, all'indirizzo corrente, su cui si trova cioè il cursore subito dopo aver aver scritto la stringa in memoria, sottraiamo l'indirizzo della prima cella in memoria a partire dalla quale la stringa viene memorizzata. Ovviamente ciò funziona solo se lo facciamo in questo punto del programma.

Il resto del programma riprende porzioni di codice che già conosciamo, in particolare la stampa su video e l'uscita dal programma.

Istruzioni di confronto, di salto, aritmetiche e shift

In questa sezione vediamo alcune delle istruzioni più importanti per effettuare confronti, saltare in punti precisi di un programma ed effettuare operazioni aritmetiche.

In generale, possiamo dire che vale di gran lunga la regola che due passaggi in memoria nell'ambito della stessa operazione non sono consentiti, quindi nelle operazioni a più operandi solo uno di essi può rappresentare una specifica locazione di memoria, esattamente come già introdotto per l'operazione MOV.

Prima di procedere, è bene sottolineare che nell'architettura ×86 è presente un ulteriore registro a 32 bit oltre a quelli già considerati, che si chiama EFLAGS. Un *flag* è un singolo bit di informazione il cui valore è fissato indipendentemente dagli altri bit. Alcuni bit del registro EFLAGS sono indefiniti, altri sono riservati al Sistema Operativo, altri ancora sono direttamente utilizzabili dal programmatore. La Figura 16 mostra in che modo vengono utilizzati i bit interni dell'EFLAGS.
L'operazione:

```
CMP OP1, OP2
```

sottrae OP2 da OP1 e altera i bit di EFLAGS a seconda del risultato di questo confronto, senza alterare in alcun modo gli operandi. É necessario che i due operandi abbiano lo stesso numero di bit per poter essere confrontati, sia che si tratti di valori immediati, registri o memoria.

L'operazione:

```
NEG OP1
```

sottrae OP1 da zero, invertendo così il segno dell'operando (complemento a 2).

Spesso è utile raggruppare alcune istruzioni del programma riferendosi ad esse attraverso un'*etichetta*. All'occorrenza, sarà possibile *saltare* a quel gruppo di istruzioni utilizzando un'operazione di salto individuata dallo mnemonico JMP. Questa operazione ha come operando un'etichetta. Esiste anche la possibilità di effettuare un salto *condizionato*, ovvero, saltare a quella porzione del programma solo se una specifica condizione è verificata. A questo livello, la verifica di una condizione può essere fatta ad esempio attraverso l'ispezione dei bit di EFLAGS che sono stati alterati a seguito di un'operazione di confronto. La Figura 17 illustra le istruzioni di salto di quest'ultima categoria. Si osserva che di solito gli *Unsigned Conditional Transfers* sono meno utilizzati, poichè servono quando si sa di lavorare con interi positivi così grandi che 31 bit non sono sufficienti ed è necessario guadagnare un ulteriore bit.

Segue un esempio di utilizzo.

0	CF	Carry Flag	0: no carry 1: carry
1	***	Udefined	
2	PF	Parity Flag	0: # of 1s is odd 1: # of 1s is even (last 8 bit)
3	***	Udefined	
4	AF	Auxiliary Carry Flag	0: no carry in BCD 1: carry in BCD
5	***	Udefined	
6	ZF	Zero Flag	0: op not zero 1: operand became zero
7	SF	Sign Flag	0: non-negative 1: negative
8	TF	Trap Flag	Reserved
9	IF	Interrupt Enable Flag	Reserved
10	DF	Direction Flag	0: autoincrement is up-memory 1: down
11	OF	Overflow Flag	0: no overflow 1: overflow
12	IOPL	I/O Privilege Flag Lev 0	Reserved
13	IOPL	I/O Privilege Flag Lev 1	Reserved
14	NT	Nested Task Flag	Reserved
15	***	Udefined	
16	RF	Resume Flag	Reserved
17	VM	Virtual-86 Mode Flag	Reserved
18	AC	Alignment Check Flag	Reserved
19	VIF	Virtual Interrupt Flag	Reserved
20	VIP	Virtual Interrupt Pending	Reserved
21	ID	CPU ID	0: CPUID not available 1: CPUID available
22-31	***	Udefined	

Figura 16: Utilizzo bit dell'EFLAGS

Unsigned Conditional Transfers

Mnemonic	Condition Tested	Jump If...
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNP/JPO	PF = 0	not parity/parity odd
JP/JPE	PF = 1	parity/parity even

Signed Conditional Transfers

Mnemonic	Condition Tested	Jump If...
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNO	OF = 0	not overflow
JNS	SF = 0	not sign (positive, including 0)
JO	OF = 1	overflow
JS	SF = 1	sign (negative)

Figura 17: Istruzioni di salto

Esempio 3 Supponiamo di voler scambiare i valori contenuti in EAX ed EBX, utilizzando una variabile `temp`, solo se il valore contenuto in EAX è più piccolo di quello contenuto in EBX. A tal fine, potremo utilizzare il seguente codice.

```
section .data
    temp dw 0
section .text
    global _start
    _start:
        cmp eax,ebx
        jge exit
    switch:
        mov [temp], eax
        mov eax, ebx
        mov ebx, [temp]
    exit:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

Per verificare che il programma sia corretto, ad esempio possiamo aggiungere come prima istruzione dell'etichetta `start` la seguente:

```
mov ebx, 1
```

e quindi inserire subito dopo un'etichetta `bp1`, per fissare su di essa un breakpoint. A questo punto, in fase di debug, fisseremo due breakpoint, uno sull'etichetta `bp1` e l'altro su `exit`: ispezionando per ciascuno dei due lo stato dei registri potremo verificare che il programma fa quello che noi vogliamo.

Vediamo adesso le più importanti operazioni aritmetiche a disposizione nell'*i*386. Sia `ADD` che `SUB` ricevono due operandi `OP1, OP2` e restituiscono `OP1 +/- OP2`, rispettivamente. Invece `INC` e `DEC` hanno l'effetto di incrementare e decrementare, rispettivamente, di uno l'unico operando che ricevono. Di seguito un esempio.

Esempio 4 Date due variabili word x e y, memorizzare nel registro EDX il numero di variazioni (incrementi o decrementi) che bisogna apportare ad x per ottenere y. Si assuma di non voler utilizzare a tal fine le operazioni `ADD` e `SUB` ma solo `INC` e `DEC`.

```
section .data
    x dw 8
    y dw 3
section .text
    global _start
    _start:
        mov edx, 0
        mov eax,[x]
        mov ecx, [y]
        cmp eax, ecx
        jl somma
        je fine
    sottrai:
        dec eax
        inc edx
        cmp eax, ecx
        jg sottrai
        je fine
    somma:
        inc eax
        inc edx
        cmp eax, ecx
        jl somma
    fine:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

Moltiplicazioni e divisioni possono essere effettuate sia con segno che senza segno, attraverso le operazioni `MUL`, `DIV` e `IMUL`, `IDIV`, rispettivamente. La Tabella 1 illustra l'utilizzo degli operandi per queste quattro operazioni.

Le operazioni logiche `AND`, `OR` e `XOR` ricevono due operandi `OP1` e `OP2` e restituiscono il risultato in `OP1`. L'operazione logica `NOT` riceve un operando di cui nega i bit (complemento a 1). Per tutte le operazioni logiche, gli operandi possono essere registri generali, variabili o immediati.

Vediamo adesso le istruzioni di shift, o scorrimento. I bit di un byte, di una word o di una doubleword possono essere shiftati aritmeticamente o logicamente, fino a un

Istruzione	Numero bit	Risultato	Operando1	Operando2
	byte	AX	AL	r/m(8)
`MUL / IMUL`	word	DX:AX	AX	r/m(16)
(1 operando)	dword	EDX:EAX	EAX	r/m(32)
`IMUL`	word	r(16)	r(16)	r/m(16),imm(8,16)
(2 operandi)	dword	r(32)	r(32)	r/m(32),imm(8,16,32)
`IMUL`	word	r(16)	r/m(16)	imm(8,16)
(3 operandi)	dword	r(32)	r/m(32)	imm(8,16,32)
	byte	AL (q) AH (r)	AX	r/m(8)
`DIV / IDIV`	word	AX (q) DX (r)	DX:AX	r/m(16)
(3 operandi)	dword	EAX (q) EDX (r)	EDX:EAX	r/m(32)

Tabella 1: Moltiplicazioni e divisioni con e senza segno

massimo di 31 bit. Un'istruzione di shift serve a shiftare i bit di un numero verso destra o verso sinistra, di un posto o di più posti. Se non viene specificato di quanto shiftare, lo shift è di un bit. Altrimenti, si può specificare di quanto shiftare o attraverso un valore immediato, o utilizzando i 5 bit meno significativi del registro CL. L'ultimo bit che è stato shiftato finisce sempre in CF.

Lo shift aritmetico verso sinistra `SAL` coincide con l'analogo shift logico `SHL`, e corrisponde a una moltiplicazione per due in binario, per ciascuno dei bit shiftati. La Figura 18 ne illustra il funzionamento.

```
                OF   CF                 OPERAND

BEFORE SHL      X    X     10001000100010001000100010001111
OR SAL

AFTER SHL       1    1 ← 0001000100010001000100010001111 0 ← 0
OR SAL BY 1

AFTER SHL       X    0 ← 0010001000100010001111000000000 0 ← 0
OR SAL BY 10
```

Figura 18: Operazioni di shift verso sinistra `SAL` e `SHL`

Lo scorrimento aritmetico verso destra `SAR` è invece diverso da quello logico `SHR`, poichè quest'ultimo riempie i bit lasciati vuoti a sinistra con degli zeri, il primo invece ripete il segno del numero da shiftare. Questa volta l'operazione associata è la divisione per due in binario. Le due Figure 19 e 20 illustrano il funzionamento di queste due operazioni.

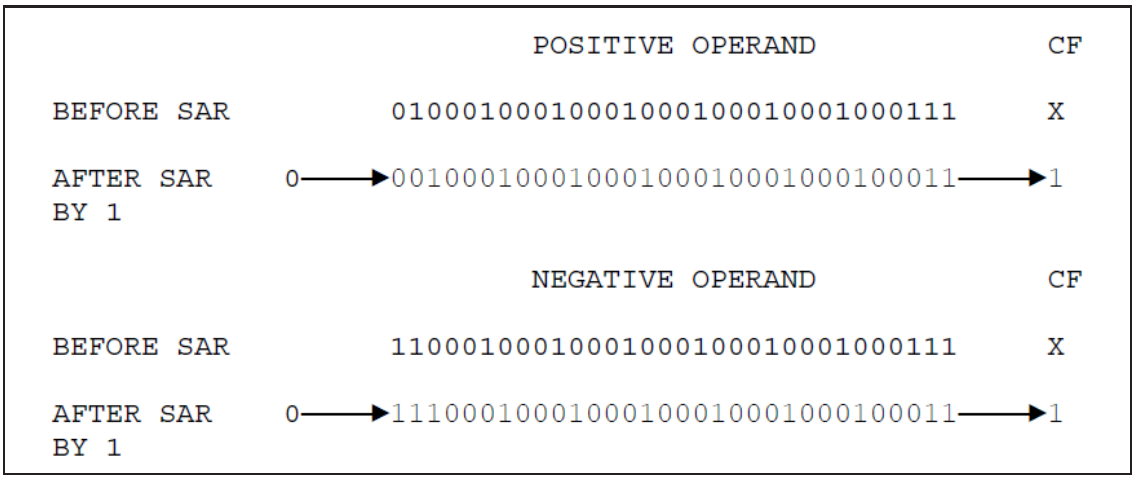


Figura 19: Operazione di shift aritmetico verso destra `SAR`

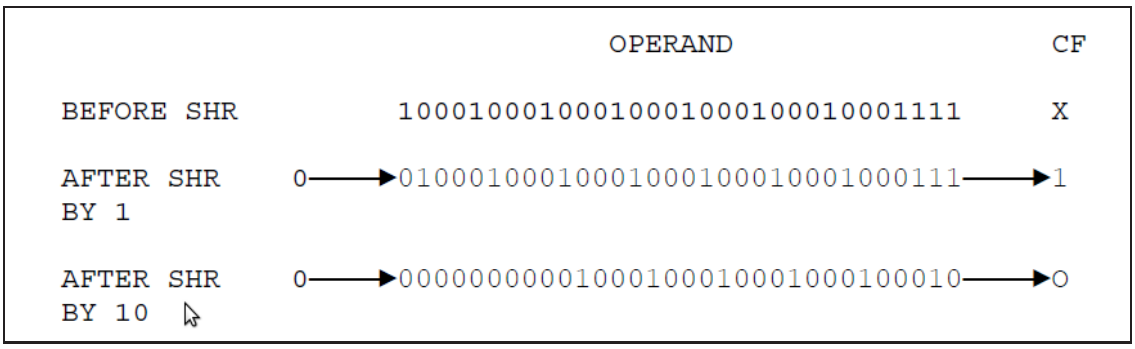


Figura 20: Operazione di shift logico verso destra `SHR`

Consideriamo adesso l'operazione individuata dallo mnemonico `LOOP`. `LOOP` riceve un operando che è un'etichetta, e decrementa un registro contatore, senza cambiare nessuno dei flag, controllando che sia verificata una specifica condizione. Se la condizione è verificata, salta all'etichetta che riceve come operando. In particolare, questa operazione serve qualora si voglia eseguire un ciclo di n passi: il valore n deve essere inserito dentro il registro ECX. Seguono degli esempi di funzionamento.

Esempio 5 Contare il numero di bit che sono *on* (cioè uguali ad uno) nel registro EAX.

```
section .data
section .text
    global _start
    _start:
        mov dl, 0; bl conterrà il numero di bit on
        mov ecx, 32; ecx è il contatore del ciclo
```

```
count_loop:
    shl eax, 1; sposta i bit nel carry flag
    jnc skip_inc; se CF == 0, vai a skip_inc
    inc dl
skip_inc:
    loop count_loop
exit:
    mov ebx, 0
    mov eax, 1
    int 0x80
```

Array

In Assembly il nome di un vettore o di una matrice rappresenta l'indirizzo della cella di memoria (di 8 bit) a partire dalla quale è memorizzato il contenuto dell'array. Nel 386 esiste un meccanismo efficiente per gestire l'indicizzazione del vettore.

In particolare, è possibile effettuare spostamenti da e/o verso la memoria utilizzando un `<effective address>` così composto:
`BASE + (INDICE * SCALA) + SPIAZZAMENTO = <registro base> + <registro indice> * 1,2,4,8 + <offset>`.

Come registro base, si possono utilizzare i registri EAX, EBX, ECX, EDX, ESI e EDI. Come registro indice, i registri EAX, EBX, ECX, EDX, ESI e EDI. Infine, lo spiazzamento è un indirizzo in memoria (ad esempio, il nome del vettore). In tal modo si può ad esempio indicizzare il vettore `V` di doubleword inizializzando a zero il valore di EAX, e considerando l'indirizzo in memoria `[V + EAX * 4]`. In tal caso stiamo usando `V` come spiazzamento, EAX come indice (che verrà incrementato sempre di una unità per scorrere il vettore) e il valore 4 come scala, dal momento che un elemento del vettore è composto da quattro celle di memoria.

Quello indicato sopra non è l'unico modo di gestire lo scorrimento degli array, come vedremo anche dagli esercizi proposti di seguito. Nel caso in cui si ha a che fare con delle matrici (array bidimensionali) il registro base può essere comodo per memorizzare il numero di righe già analizzate, in modo che attraverso un utilizzo opportuno di indice e scala si possa scorrere la riga corrente muovendosi lungo le colonne.

Esempio 6 Dato un vettore di doubleword, inserire in una variabile max il massimo del vettore.

```
section .data
```

```
    V dd 72, 54, 89, 21, 0, 12
    n equ 6
section .bss
    max resd 1
section .text
     global _start
     _start:
        mov eax, 0; usiamo eax come indice per scorrere il vettore
        mov ebx, [V]; usiamo ebx per tenere il massimo temporaneo
        mov ecx, n; mettiamo in ecx la n del ciclo for
        dec ecx; poichè partiamo dall'elemento 1 e non da quello 0
        jmp lp
    change_max:
        mov ebx, [V+eax*4]
        loop lp
    lp:
        inc eax
        cmp ebx, [V+eax*4]
        jl change_max
        loop lp
    return_max:
        mov [max], ebx
    exit:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

Esempio 7 Dato un vettore array1 di byte composto da cinque elementi, per ciascun elemento i di array1 copiare il prodotto tra array1[i] e array1[n-1-i] nell'elemento i di un altro vettore di 5 byte (si assuma che il prodotto delle coppie di elementi sia ancora memorizzabile in un byte).

```
section .data
    array1 db 5, 4, 3, 2, 1 ;array di 5 byte
section .bss
    n equ 5
```

```
    array2 resb n
section .text
    global _start
    _start:
        mov esi, 0; registro da usare come indice
        mov ecx, n;
        mov edi, n;
   lp:
        mov al, [array1+esi]
        dec edi; adesso n-1-i si trova in edi
        mov dl, [array1+edi]
        imul dl
        mov [array2+esi], al
        inc esi
        loop lp
   fine:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

Esempio 8 Data una matrice quadrata M di interi a 16 bit e un numero k, scrivere un programma Assembly che inserisca nel registro ecx il valore della somma degli elementi di M.

```
section .data
    M dw 4, 2, 8, 21, 0, 2, 39, 15, 12
    dim equ $-M
    n equ 3
section .text
    global _start
    _start:
        mov eax, 0; in eax teniamo lo spazio occupato dalle righe
        mov ebx, 0; in ebx teniamo l'indice di colonna
        mov ecx, 0; in ecx teniamo la somma parziale
        mov esi, n
        shl esi, 1; in esi teniamo n*2
```

```
    scan_row:
        cmp ebx, n
        jge change_row
        add cx, [eax+ebx*2+M]
        inc ebx
        jmp scan_row
    change_row:
        mov ebx, 0
        add eax, esi; se siamo qui dobbiamo aggiungere al contenuto di eax
                      n elementi di 16 bit, ovvero n byte per 2
        cmp eax, dim
        jl scan_row
    exit:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

Esempio 9 Data una matrice quadrata M di interi a 16 bit e un numero k, scrivere un programma Assembly che stampi "vero" se il numero di occorrenze di k in M è uguale o superiore a k e stampi "falso" altrimenti.

```
section .data
    M dw 2, 2, 89, 21, 0, 2, 39, 15, 0
    dim equ $-M
    n equ 3
    k equ 2
    vero db ''VERO'', 10
    len_vero equ $ - vero
    falso db ''FALSO'', 10
    len_falso equ $ - falso
section .text
    global _start
    _start:
        mov eax, 0; in eax teniamo lo spazio occupato dalle righe
        mov ebx, 0; in ebx teniamo l'indice di colonna
        mov ecx, 0; in ecx teniamo il numero di occorrenze di k in M
        mov esi, n
```

```
        shl esi, 1; in esi teniamo n*2
        jmp scan_row
inc_occ:
        inc ecx
scan_row:
        cmp ebx, n
        jge change_row
        mov dx, [eax+ebx*2+M]
        inc ebx
        cmp dx, k
        je inc_occ
        jmp scan_row
change_row:
        cmp eax, dim
        je compare
        add eax, esi; se siamo qui dobbiamo aggiungere al contenuto di eax
                      n elementi di 16 bit, ovvero n byte per 2
        mov ebx, 0
        jmp scan_row
compare:
        cmp ecx, k
        jl print_false
print_true:
        mov edx, len_vero; arg3, lunghezza stringa da stampare
        mov ecx, vero; arg2, indirizzo in memoria della stringa
        mov ebx, 1; arg1, stampa su schermo
        mov eax, 4; numero del servizio
        int 0x80; chiamata al SO
        jmp exit
print_false:
        mov edx, len_falso; arg3, lunghezza stringa da stampare
        mov ecx, falso; arg2, indirizzo in memoria della stringa
        mov ebx, 1; arg1, stampa su schermo
        mov eax, 4; numero del servizio
        int 0x80; chiamata al SO
exit:
        mov ebx, 0
```

```
        mov eax, 1
        int 0x80
```

Esempio 10 Data una matrice quadrata M di interi a 32 bit, scrivere un programma Assembly che stampi "vero" se la somma degli elementi della diagonale principale è maggiore della somma degli elementi delle diagonali sopra e sotto la diagonale principale, stampi "falso" altrimenti.

```
section .data
    M dd 50, 2, 9, 21, 10, 12, 9, 15, 25
    n equ 3
    vero db ''VERO'', 10
    len_vero equ $ - vero
    falso db ''FALSO'', 10
    len_falso equ $ - falso
section .text
    global _start
    _start:
        mov ebx, 4;
        mov ecx, [M]; in ecx teniamo la somma degli elementi della
                      diagonale principale
        mov edx, [M+ebx]; in edx teniamo la somma degli elementi
                          delle altre due diagonali
        mov ebx, 0
        mov eax, n
        shl eax, 2
        mov edi, n
        dec edi
   scan_diagonals:
        mov esi, [eax+ebx*4+M]
        add edx, esi
        inc ebx
        mov esi, [eax+ebx*4+M]
        add ecx, esi
        inc ebx
        cmp ebx, edi
```

```
        jg compare
        mov esi, [eax+ebx*4+M]
        add edx, esi
        dec ebx
        mov esi, n
        shl esi, 2
        add eax, esi
        jmp scan_diagonals
compare:
        cmp ecx, edx
        jle print_false
print_true:
        mov edx, len_vero; arg3, lunghezza stringa da stampare
        mov ecx, vero; arg2, indirizzo in memoria della stringa
        mov ebx, 1; arg1, stampa su schermo
        mov eax, 4; numero del servizio
        int 0x80; chiamata al SO
        jmp exit
print_false:
        mov edx, len_falso; arg3, lunghezza stringa da stampare
        mov ecx, falso; arg2, indirizzo in memoria della stringa
        mov ebx, 1; arg1, stampa su schermo
        mov eax, 4; numero del servizio
        int 0x80; chiamata al SO
exit:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

Approfondimenti

1 Giacomo Bucci. Architettura e organizzazione dei calcolatori elettronici – Fondamenti. McGraw-Hill Companies, 2005.

2 Sergio Congiu. Architettura degli elaboratori – Organizzazione dell'hardware e programmazione in linguaggio assembly. Pátron, 2007.

3 Intel 80386. Programmer's reference manual, 1986.

4 Daniele Giacomini. Appunti di informatica libera. `http://informaticalibera.net/`

Appendice

Se si utilizza il Sistema Operativo Windows, è possibile stampare su video attraverso il seguente insieme di istruzioni:

```
push cs
pop ds
mov ah, 9
mov dx, Msg; Stringa terminata da $
int 21h
```

Per uscire dal programma invece bisogna utilizzare:

```
mov al, 0; exit code, 0=norma
mov ah, 09h; exit command to kernel
int 21h; interrupt 80 hex, call kernel
```