Programmazione Assembly

Giacomo Fiumara
giacomo.fiumara@unime.it

Anno Accademico 2013-2014

La famiglia dei processori Pentium

- Processore 4004 (1969)
  - bus indirizzi a 12 bit
  - bus dati a 4 bit
- Processore 8086 (1979)
  - bus indirizzi a 20 bit
  - bus dati a 16 bit
- Processore 8088 (1980)
  - bus indirizzi a 20 bit
  - bus dati a 8 bit
  - (per il resto identico all'8086)
  - Possono indirizzare fino a 4 segmenti di memoria di 64 kB ciascuno
- 80186 (1982)
  - bus indirizzi a 20 bit
  - bus dati a 16 bit
  - Instruction set migliorato rispetto all'8086
  - Scarsamente utilizzato

La famiglia dei processori Pentium (segue)

- 80286 (1982)
  - bus indirizzi a 24 bit
  - bus dati a 16 bit
  - Può indirizzare fino a 16 MB di RAM
  - Con questo modello si introduce la **modalità protetta**
- 80386 (1985)
  - bus indirizzi a 32 bit
  - bus dati a 32 bit
  - Può indirizzare fino a 4 BG di RAM
  - Permette la definizione di segmenti di dimensioni fino a 4 GB
- 80486 (1989)
  - bus indirizzi a 32 bit
  - bus dati a 32 bit
  - Combina le funzionalità che fino al modello precedente erano delegate al coprocessore matematico
  - Presenta la capacità di decodificare ed eseguire le istruzioni in parallelo
  - Presenta cache L1 ed L2 di 8 KB

La famiglia dei processori Pentium (segue)

- Pentium (1993)
  - bus indirizzi a 32 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Architettura superscalare a due pipeline (due istruzioni per ciclo di clock)
  - Cache L1 da 16 KB (8 KB per i dati e 8 KB per le istruzioni)
- Pentium Pro (1995)
  - bus indirizzi a 36 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Architettura superscalare a tre pipeline (tre istruzioni per ciclo di clock)
  - Cache L1 da 16 KB (8 KB per i dati e 8 KB per le istruzioni), cache L2 da 256 KB

La famiglia dei processori Pentium (segue)

- Pentium II (1997)
  - bus indirizzi a 32 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Architettura superscalare a due pipeline (due istruzioni per ciclo di clock)
  - Cache L1 da 32 KB (16 KB per i dati e 16 KB per le istruzioni), cache L2 da 512 KB
- Pentium III (1999)
  - bus indirizzi a 36 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Architettura superscalare a due pipeline (due istruzioni per ciclo di clock)
  - Cache L1 da 16 KB (8 KB per i dati e 8 KB per le istruzioni), cache L2 da 512 KB

La famiglia dei processori Pentium (segue)

- Pentium 4 (2001)
  - bus indirizzi a 32 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Architettura superscalare a due pipeline (due istruzioni per ciclo di clock)
  - Cache L1 da 8 KB, cache L2 da 512 KB, cache L3 da 2 MB
- Pentium D (2005)
  - bus indirizzi a 36 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Architettura superscalare a due pipeline (due istruzioni per ciclo di clock)
  - Cache L1 da 32 KB (8 KB per i dati e 8 KB per le istruzioni), cache L2 da 4 MB
  - Primo processore dual core

La famiglia dei processori Pentium (segue)

- Core 2 Duo (2006)
  - bus indirizzi a 64 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Registri a 64 bit
  - Cache L1 da 128 KB, cache L2 da 6 MB
- Core i3 (2010)
  - bus indirizzi a 36 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Registri a 64 bit
  - Cache L1 da 32 KB (8 KB per i dati e 8 KB per le istruzioni), cache L2 da 4 MB
  - Due core

La famiglia dei processori Pentium (segue)

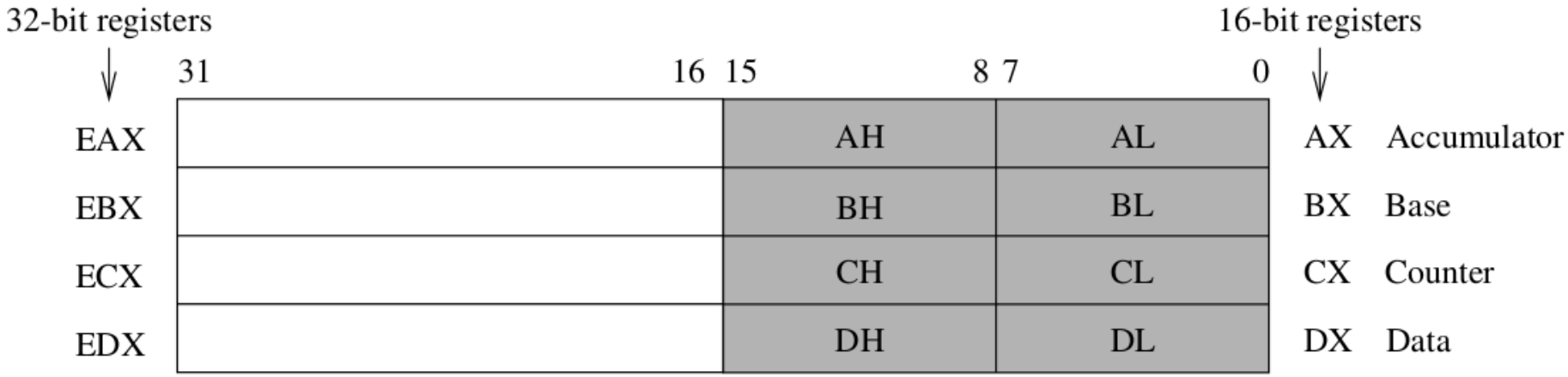
- Core i5 (2009)
  - bus indirizzi a 64 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Registri a 64 bit
  - Cache L1 da 128 KB, cache L2 da 6 MB
  - Quattro core
- Core i7 (2008)
  - bus indirizzi a 36 bit
  - bus dati a 64 bit (ma registri a 32 bit)
  - Registri a 64 bit
  - Cache L1 da 32 KB (8 KB per i dati e 8 KB per le istruzioni), cache L2 da 4 MB
  - Due/Sei core

Registri

Introduzione

- Una delle cose più importanti di un processore è l'insieme dei **registri**
- Quelli accessibili al programmatore sono detti **general purpose**
- Nel caso delle architetture x86 sono presenti, per motivi di compatibilità, registri a 8, 16, 32 bit

Registri



- È la situazione dei registri `eax`, `ebx`, `ecx`, `edx`
- Ognuna della porzioni può essere considerata un registro vero e proprio
- Ognuna delle porzioni può essere utilizzata indipendentemente (e contemporaneamente) dalle altre

Registri

Registri a 32 bit	Nome	16/8 bit	Descrizione
eax	Accumulator	ax, ah, al	Arithmetic and Logic
ebx	Base	bx, bh, bl	Arrays
ecx	Counter	cx, ch, cl	Loops
edx	Data	dx, dh, dl	Arithmetics
esi	Source Index	si	Strings and arrays
edi	Destination Index	di	Strings and arrays
esp	Stack Pointer	sp	Top of stack
ebp	Base Pointer	bp	Base of stack
eip	Instruction Pointer	ip	Points to next instruction
eflags	Flag	flag	Status and control flag

Registro **eflags**

- Si tratta di un registro particolare, nel senso che ogni singolo bit che lo compone può essere riferito indipendentemente dagli altri
- Ogni bit controlla un'operazione della CPU o fornisce informazioni sul risultato dell'operazione
- Si tratta comunque di un registro che può essere acceduto in lettura e scrittura:
  - In lettura per verificare, come detto, il risultato di un'operazione
  - In scrittura per impostare lo stato di un flag del processore

Registro **eflags** /2

- Il registro **eflags** è un registro a 16 bit
- 7 bit non sono utilizzati
- 6 flag condizionali (contengono informazioni sullo stato delle operazioni della CPU):
  - Carry Flag (CF)
  - Zero Flag (ZF)
  - Parity Flag (PF)
  - Sign Flag (SF)
  - Auxiliary Carry Flag (AF)
  - Overflow Flag (OF)
- 3 flag di controllo:
  - Direction Flag (DF)
  - Interrupt Flag (IF)
  - Trap Flag (TF)

Registro **eflags** /3

Flag condizionali

Carry Flag (CF)
Viene settato se si verifica un riporto sul bit più significativo durante un calcolo. Il riporto può verificarsi sul bit 7 (operazioni a 8 bit) oppure sul bit 15 (operazioni a 16 bit)

Zero Flag (ZF)
Viene settato quando il risultato di un'operazione logico-aritmetica è zero. Per esempio: quando si decrementa il contenuto di un registro (di solito il registro CX, che viene usato come contatore), il contenuto del registro sarà zero. In quel momento ZF viene settato. Un altro caso è rappresentato dal confronto tra due numeri.

Parity Flag (PF)
Quando questo flag viene settato, si ha un numero pari di bit nel byte meno significativo del registro destinazione. Non viene molto usato.

Registro **eflags** /4

Flag condizionali /2

Sign Flag (SF)
Viene settato se il risultato di un'operazione logico-aritmetica è negativo. Di fatto contiene il MSB (most significant bit) del risultato, ovvero il bit di segno nelle operazioni aritmetiche con segno.

Auxiliary Carry Flag (AF)
Identico a CF, salvo il fatto che viene settato quando si produce riporto sul bit 3, cioè quando si produce riporto sui 4 bit meno significativi.

Overflow Flag (OF)
Viene settato se:

- si produce overflow nel MSB con riporto
- si produce overflow senza riporto

Registro **eflags** /5

Flag di controllo

Direction Flag (DF)
Controlla la direzione (sx-dx o viceversa) nella manipolazione delle stringhe. Quando è settato, le operazioni che auto-incrementano i registri sorgente e destinazione (SI e DI) incrementano entrambi i registri e il flusso dei caratteri che compongono le stringhe avviene da sinistra verso destra. Quando è posto uguale a zero, il flusso avviene da destra verso sinistra.

Interrupt Flag (IF)
Quando viene settato da programmi con sufficienti privilegi, gli interrupt hardware mascherabili possono essere gestiti.

Trap Flag (TF)
Viene settato per scopi di debug: i programmi vengono eseguiti un'istruzione per volta e si arrestano. Ovviamente, questo consente di esaminare lo stato dei registri.

Registri di segmento

Architettura della memoria

- Il processore 8086 ha un bus degli indirizzi a 20 bit
- Questo significa che la memoria indirizzabile ammonta a $2^{20} = 1.048.576$ byte
- Cioè dall'indirizzo $00000H$ all'indirizzo $fffffH$
- La memoria è vista come suddivisa in quattro segmenti:
  - Segmento dati
  - Segmento codice
  - Segmento stack
  - Segmento extra

Registri di segmento /2

- Ognuno dei segmenti è accessibile mediante l'indirizzo contenuto nel corrispondente registro di segmento
- Ogni registro memorizza l'indirizzo base (di partenza) del segmento corrispondente
- Considerata la differenza di dimensione (indirizzo a 20 bit, registro a 16), nei registri vengono memorizzati i 16 bit più significativi dell'indirizzo
- L'indirizzo fisico (a 20 bit) di un dato viene calcolato a partire dall'indirizzo base (contenuto nel registro)
- Per esempio, l'indirizzo logico $2222H : 0016H$ diventa $22220H + 0016H = 22236H$
- Si noti che il registro DS (Data Segment) contiene il valore $2222H$, e che la quantità $0016H$ viene detta scostamento (offset)

Registri di segmento /3

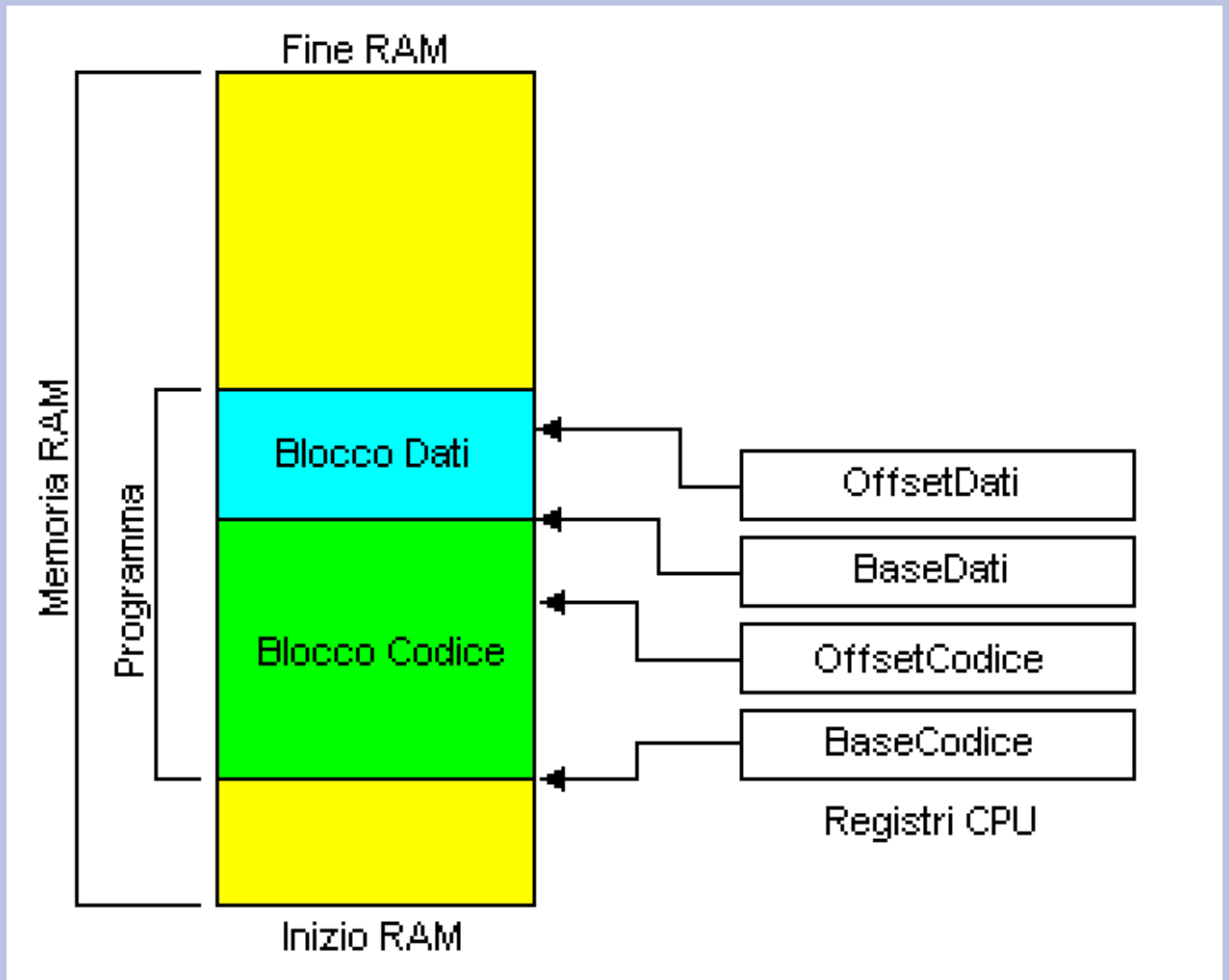
Segmento di Codice
Il segmento di codice (Code Segment, CS) è l'area di memoria che contiene soltanto il codice di un programma. L'offset rispetto al segmento di codice è fornito dall'Instruction Pointer (IP), che punta sempre all'istruzione successiva. L'indirizzo logico di una istruzione è pertanto CS:IP

Segmento di Stack
Il segmento di stack (Stack Segment, SS) è l'area di memoria che contiene lo stack. È gestita seconda la logica *last-in-first-out*. Il registro Stack Pointer punta alla cima dello stack. Un indirizzo logico $4466H : 0122H$ significa che il registro SS contiene il valore $4466H$ e che il registro IP $0122H$: l'indirizzo fisico si ottiene come $44660H + 01220H = 44782H$.

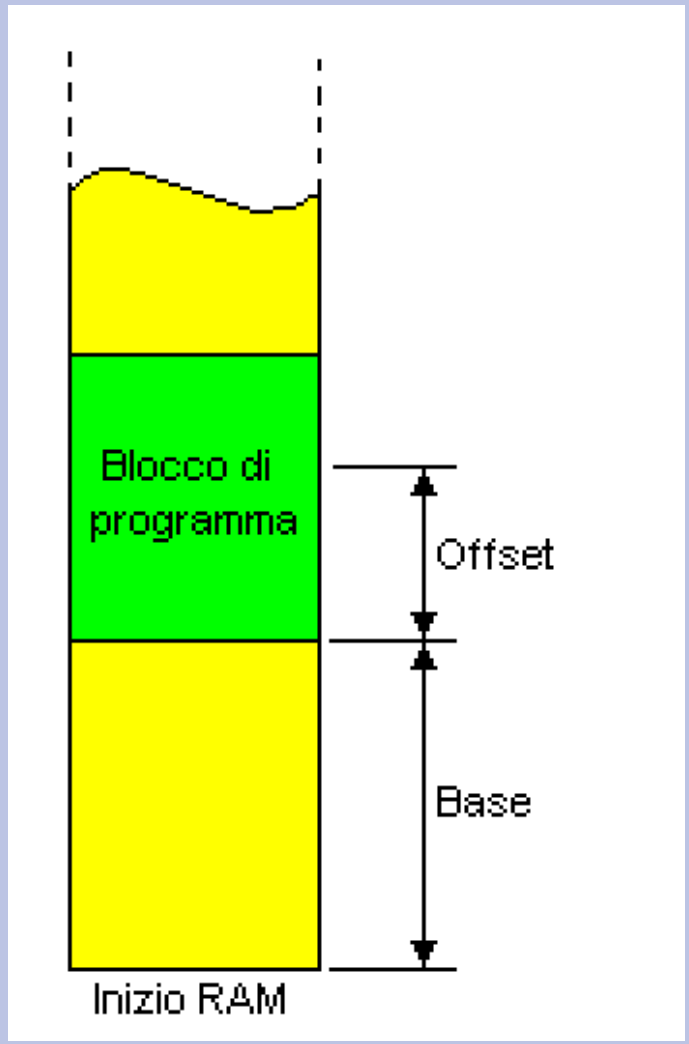
Registri di segmento /4

Segmento Dati e Segmento Extra
Contengono entrambi dati, ma in alcuni casi particolari (quando si gestiscono stringhe) può essere necessario gestirli in segmenti separati. I registri corrispondenti sono DS (Data Segment) e ES (Extra Segment).

Struttura della memoria



Struttura della memoria /2



Introduzione

- Il linguaggio Assembly è un linguaggio di programmazione a basso livello
- È direttamente influenzato dall'instruction set e dall'architettura del processore

- Il codice Assembly viene tradotto in linguaggio macchina da un software chiamato *assembler*

Introduzione

- I linguaggi di alto livello, rispetto al linguaggio assembly, hanno alcuni vantaggi:
  - sviluppo più veloce
  - manutenzione più facile
  - maggiore portabilità
- Al contrario, il linguaggio assembly presenta due vantaggi:
  - efficienza
  - accessibilità all'hardware del sistema

Introduzione

Istruzioni assembly

I programmi assembly sono costituiti da tre tipi di istruzioni:

- le istruzioni (propriamente dette)
  — ogni istruzione consiste di un codice operativo (*opcode*) e viene tradotta in un'istruzione in linguaggio macchina
- le direttive (o pseudo-operazioni)
  — forniscono alcune informazioni all'assembler e **non** producono linguaggio macchina
- le macro
  — rappresentano una scorciatoia rispetto alla codifica di un gruppo di istruzioni. Nel corso della traduzione, le macro vengono sostituite dalle istruzioni equivalenti che, a loro volta, vengono tradotte in linguaggio macchina.

Introduzione

Istruzioni assembly

- Le istruzioni vengono inserite in un file "sorgente" una per riga
- Il formato dei tre tipi di istruzioni è il seguente:

```
[label] mnemonic [operands] [;comment]
```

Per esempio:

```
repeat: inc result ;incrementa di 1 result
```

```
CR EQU 0DH ;carattere di carriage return
```

- Nei linguaggi di alto livello, l'allocazione dello spazio per le variabili viene fatto indirettamente specificando il tipo di dato di ogni variabile da usare nel programma.
- Per esempio:

```
char carattere; /* 1 byte */
int intero;     /* 2 byte */
float reale;    /* 4 byte */
double doppio;  /* 8 byte */
```

Introduzione

Allocazione dei dati

- Queste dichiarazioni non soltanto specificano lo spazio da destinare alla memorizzazione del singolo dato
- Specificano anche l'interpretazione dei numeri memorizzati
- Per esempio:

```
unsigned val1;
int      val2;
```

- Alle due variabili vengono riservati 2 byte di spazio
- Ma il numero 1000 1101 1011 1001
- viene interpretato come 36281 se memorizzato in *val*1 e -29255 se memorizzato in *val*2

Introduzione

Allocazione dei dati

- In assembly, l'allocazione dello spazio viene fatta mediante l'uso della **direttiva** `define`
- che può assumere le forme

```
DB  Define Byte         ;alloca 1 byte
DW  Define Word         ;alloca 2 byte
DD  Define Doubleword   ;alloca 4 byte
DQ  Define Quadword     ;alloca 8 byte
DT  Define Ten Bytes    ;alloca 10 byte
```

- Il formato generico di un'istruzione di allocazione è:

```
[varname] define-directive initial-value [,initial-value], ...
```

- Alcuni esempi

```
char1 DB 'y'
```

- Alloca un byte di spazio e lo inizializza al carattere `y`
- È anche possibile allocare dello spazio senza alcuna inizializzazione, per esempio:

```
char2  DB ?
```

- Le seguenti allocazioni:

```
char1 DB 'y'

char1 DB 79h

char1 DB 1111001b
```

- Producono esattamente lo stesso risultato pratico
- Si noti che il suffisso `h` indica un numero espresso in base esadecimale, mentre `b` indica la base binaria
- Il suffisso `d` indica la base decimale, ma non è mai usato perchè rappresenta la base di default

Introduzione

Allocazione dei dati - Ordinamento little endian

- L'allocazione:

```
val1 DW 25159
```

- Produce la conversione (automatica) nel numero $6247H$
- A causa dell'ordinamento little endian, la memorizzazione sarà:

```
locazione    x    x+1
contenuto    47   62
```

Introduzione

Allocazione dei dati - Numeri negativi

- La seguente assegnazione:

```
neg1 DW -29255
```

- Produce la conversione (automatica) nel numero $8DB9$
- **(I numeri negativi vengono memorizzati in complemento a due)**
- A causa dell'ordinamento little endian, la memorizzazione sarà:

```
locazione    x    x+1
contenuto    B9   8D
```

Introduzione

Allocazione dei dati - Numeri double word

- La seguente assegnazione:

```
big DD 542803535
```

- Produce la conversione (automatica) nel numero $205A864FH$
- A causa dell'ordinamento little endian, la memorizzazione sarà:

```
locazione    x    x+1    x+2    x+3
contenuto    4F   86     5A     20
```

Introduzione

Allocazione dei dati - Intervalli

- L'intervallo di variabilità degli operandi numerici dipende dal numero di byte allocati

Direttiva	**Intervallo di variabilità**
DB	Da -128 a 255 (da -2^7 a $2^8 - 1$)
DW	Da -32768 a 65535 (da -2^{15} a $2^{16} - 1$)
DD	Da $-2.147.483.648$ a $4.294.967.295$ (da -2^{31} a $2^{32} - 1$, 32 bit)
DQ	Da -2^{63} a $2^{64} - 1$, 64 bit

Introduzione

Allocazione dei dati - Intervalli

- L'assegnazione ad una variabile di un valore esterno all'intervallo può produrre un errore dell'assemblatore oppure un valore errato.
- Per esempio:

```
byte1 DB 256
byte2 DB -200
```

- Nel primo caso si ottiene un errore da parte dell'assemblatore
- Nel secondo caso, il numero in complemento a due è $FF38H$ di cui viene memorizzato (su 8 bit) soltanto il byte meno significativo ($38H$)

Introduzione

Allocazione dei dati - Definizioni multiple

- Quando si effettua un'assegnazione multipla, come per esempio:

```
char1 DB 'y'          ; 79H
val1  DW 25159        ; 6247H
real1 DD 542803535    ; 205A864FH
```

- L'assemblatore assegna locazioni di memoria contigue, cioè:

```
indirizzo    x      x+1    x+2      x+3    x+4    x+5    x+6
contenuto    79     47     62       4F     86     5A     20
                    ---------       ----------------------
             char1    val1                real1
```

Introduzione

Allocazione dei dati - Definizioni multiple

- Le definizioni multiple possono essere abbreviate, per esempio:

```
msg     DB 'C'
        DB 'I'
        DB 'A'
        DB 'O'
```

- Può essere abbreviata come:

```
msg     DB  'C','I',A','O'
msg     DB  'CIAO'
```

Introduzione

Allocazione dei dati - Definizioni multiple

- Un altro esempio:

```
msg     DB 'C'
        DB 'I'
        DB 'A'
        DB 'O'
        DB 0dh
        DB 0ah
```

- Può essere reso anche come:

```
msg     DB  'CIAO',0dh, 0ah
```

- Ovviamente il discorso vale anche per definizioni di valori numerici:

```
vec1    DW  0
        DW  0
        DW  0
        DW  0
```

- Può essere reso anche come:

```
vec1    DW  0, 0, 0, 0
```

Introduzione

Allocazione dei dati - Inizializzazioni multiple

- Se l'array dell'esempio precedente fosse numeroso, sarebbe il caso di ottimizzare la definizione
- A tale scopo si usa la direttiva `DUP`, che permette inizializzazioni multiple
- Per esempio:

```
vec1    DW  4 DUP (0)
```

- La direttiva `DUP` è utile per definire array e matrici, per esempio:

```
table1  DW  10 DUP(?)    ; array di 10 word, non inizializzato
name1   DB  30 DUP('?') ; 30 byte, ognuno inizializzato a ?
name2   DB  30 DUP(?)    ; 30 byte, non inizializzato
msg     DB  3  DUP('Ciao ')
; 15 byte, inizializzato a 'Ciao Ciao Ciao '
```

Introduzione

Allocazione dei dati - Tabella dei simboli

- Quando si alloca dello spazio mediante una direttiva di definizione, di solito si associa un nome simbolico come riferimento
- L'assemblatore, durante la traduzione, assegna un offset ad ogni nome simbolico, per esempio:

```
.DATA
pippo   DW  0                          ; 2  byte
pluto   DD  0                          ; 4  byte
qui     DW  10 DUP (?)                 ; 20 byte
msg     DB  'Inserire il valore: ',0   ; 21 byte
char1   DB  ?                          ; 1  byte
```

Introduzione

Allocazione dei dati - Tabella dei simboli

```
.DATA
pippo   DW  0                          ; 2  byte
pluto   DD  0                          ; 4  byte
qui     DW  10 DUP (?)                 ; 20 byte
msg     DB  'Inserire il valore: ',0   ; 20 byte
char1   DB  ?                          ; 1  byte
```

- A questa assegnazione corrisponde la seguente tabella dei simboli:

Nome	Offset
pippo	0
pluto	2
qui	6
msg	26
char1	46

- Molte istruzioni del linguaggio Assembly richiedono che gli operandi vengano specificati
- Il linguaggio Assembly del processore Pentium fornisce molti modi per specificare gli operandi
- Queste modalità vengono chiamate *modi di indirizzamento*
- Un operando si può trovare in una delle seguenti locazioni:
  - In un registro (*indirizzamento mediante registro*)
  - Nell'istruzione stessa (*indirizzamento immediato*)
  - Nella memoria principale (all'interno del segmento dati)
  - In una porta di I/O

Introduzione

Operandi - Indirizzamento mediante registro

- In questa modalità i registri della CPU contengono gli operandi, per esempio

```
mov EAX, EBX
```

- La sintassi di questa istruzione è:

```
mov destination, source
```

- L'effetto di questa istruzione è la copia del contenuto di `source` in `destination`. Si noti che il contenuto di `source` **non** viene distrutto

Introduzione

Operandi - Indirizzamento mediante registro

- Nell'esempio precedente l'istruzione `mov` operava su registri a 32 bit
- È anche possibile operare su registri a 16 o 8 bit, per esempio:

```
mov BX, CX
mov AL, CL
```

- L'indirizzamento mediante registro è il modo più efficiente di specificare gli operandi perchè gli operandi si trovano nei registri e non è richiesto accesso alla memoria

Introduzione

Operandi - Indirizzamento immediato

- In questa tecnica di indirizzamento gli operandi fanno parte dell'istruzione
- Pertanto gli operandi fanno parte del segmento codice (e non del segmento dati)
- Questa tecnica di indirizzamento viene usata per specificare costanti
- Per esempio:

```
mov AL, 39
```

- Questa tecnica di indirizzamento è veloce perchè l'operando viene prelevato (*fetch*) insieme all'istruzione e non deve essere specificato alcun indirizzo di memoria

Introduzione

Operandi - Indirizzamento diretto

- L'indirizzamento diretto richiede l'accesso alla memoria principale (segmento dati)
- Questo significa specificare (implicitamente) l'indirizzo base del segmento data e il valore di offset interno al segmento
- Nell'indirizzamento diretto l'offset viene specificato come parte dell'istruzione
- Nei linguaggi assembly il valore viene indicato dal nome della variabile (l'assemblatore traduce il nome nel valore dell'offset durante il processo di assemblaggio)
- Questa tecnica di indirizzamento viene usata per accedere a semplici variabili, ma non si presta per la manipolazinoe di strutture dati più complesse (per esempio array)

Introduzione

Operandi - Indirizzamento diretto

- Si considerino per esempio le seguenti inizializzazioni:

```
risposta DB 'S'
table1   DW 20 dup(0)
name1    DB 'Giuseppe Verdi'
```

- Ed alcune operazioni di copia:

```
mov al, risposta    ; 'y' copiato in AL
mov risposta 'N'    ; 'N' nel byte indicato da risposta
mov name1, 'K'      ; 'K' primo carattere della stringa name1
mov table1, 61      ; 61 primo elemento dell'array table1
```

Introduzione

Operandi - Indirizzamento diretto

- Il processore Pentium (come tutti i suoi predecessori) non consente operazioni in cui **entrambi** gli operandi risiedano in memoria
- Per esempio, l'operazione:

```
a = b + c + d
```

- Si traduce in una sequenza di istruzioni Assembly con indirizzamento diretto:

```
mov eax, b
add eax, c
add eax, d
mov a, eax
```

Introduzione

Operandi - Indirizzamento indiretto

- L'indirizzamento indiretto è utile per accedere ad elementi di strutture dati complesse
- Al contrario, con l'indirizzamento indiretto l'indirizzo di partenza della struttura dati viene memorizzato nel registro BX (per esempio)
- Ogni manipolazione del contenuto del registro BX implica l'accesso ad elementi differenti della struttura

```
mov AX, [BX]    ; indirizzamento indiretto
mov AX, BX      ; copia del contenuto di BX in AX
```

- Attenzione: nei segmenti a 16 bit, soltanto i registri BX, BP, SI, DI possono essere utilizzati per registrare l'offset
- Nei segmenti a 32 bit, tutti i registri a 32 bit (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) possono ospitare l'offset

Introduzione

Operandi - Indirizzamento indiretto

- Si pone il problema di ottenere l'indirizzo iniziale della struttura, infatti l'istruzione:

```
mov BX, table1
```

- Avrebbe come effetto soltanto la copia in BX del primo elemento di `table1`
- Per questo motivo si introduce la direttiva `offset`, che copia l'offset della struttura specificata nel registro indicato
- Per esempio:

```
mov BX, offset table1
```

- Nell'esempio seguente viene assegnato il valore 100 al primo elemento di `table1` e 99 al secondo elemento
- BX viene incrementato di 2 perchè ogni elemento dell'array occupa 2 byte

```
mov BX, offset table1
mov [BX], 100
add BX, 2
mov [BX], 99
```

Introduzione

Operandi - Segmenti di default

- L'indirizzamento indiretto permette di specificare dati memorizzati nel **segmento dati** o nel **segmento stack**
- **indirizzi a 16 bit**:
  - L'indirizzo effettivo nei registri BX, SI o DI viene considerato come offset nel segmento dati
  - Se invece vengono considerato i registri BP o SP, il valore di offset viene riferito a dati contenuti nel segmento di stack
- **indirizzi a 32 bit**:
  - L'indirizzo effettivo nei registri EAX, EBX, ECX, EDX, ESI, EDI viene considerato come offset nel segmento dati
  - Se invece vengono considerato i registri EBP o ESP, il valore di offset viene riferito a dati contenuti nel segmento di stack
- In ogni caso, operazioni di `push` e `pop` fanno riferimento allo stack

Introduzione

Operandi - Override dei segmenti di default

- È possibile "scavalcare" l'associazione al segmento di default mediante il prefisso di segmento, per esempio:

```
add AX, SS:[BX]
```

- Può essere usato per accedere a dati contenuti nello stack il cui offset (rispetto al segmento SS) è contenuto nel registro BX (e non in BP o SP)
- Analogamente, è possibile accedere a dati contenuti nel segmento dati mediante il registro BP (o SP):

```
add AX, DS:[BP]
```

- Anche i prefissi dei segmenti CS ed ES possono essere utilizzati per l'override delle associazioni di default
- Per esempio:

- In alternativa alla direttiva `offset`, l'indirizzo effettivo può essere caricato in un registro mediante l'istruzione `lea` (Load Effective Address), che ha sintassi:

```
lea register, source
```

- Per esempio:

```
lea BX, table1
```

- La differenza è che `lea` calcola il valore di offset a *run-time*, mentre la direttiva `offset` ad *assembly-time*
- L'istruzione `lea` è molto più flessibile. Per esempio, la seguente istruzione non può essere realizzata con la direttiva `offset`

```
lea BX, array[SI]
```

Trasferimento dati

Istruzione MOV

- L'istruzione `mov` richiede due operandi e ha la seguente sintassi:

```
mov destination, source
```

- I dati vengono **copiati** da `source` a `destination`
- L'operando `source` non viene modificato
- Gli operandi devono avere la stessa dimensione
- Le operazioni memoria — memoria non sono consentite

Trasferimento dati

Istruzione MOV

- L'istruzione `mov` può avere una delle seguenti cinque forme:
- `mov register, register`
  - Il registro di destinazione non può essere CS o IP
  - Nessuno dei due registri può essere un registro di segmento (CS, DS, SS, ES)
- `mov register, immediate`
  - Il registro non può essere un registro di segmento (CS, DS, SS, ES)
- `mov memory, immediate`
- `mov register, memory`
- `mov memory, register`

Trasferimento dati

Istruzione XCHG

- L'istruzione `xchg` serve per scambiare tra loro gli operandi
- Può essere applicata ad operandi a 8, 16, 32 bit (ma omogenei)
- Per esempio:

```
xchg EAX, EDX
xchg risposta, CL
xchg totale, DX
```

- Come l'istruzione `mov` neanche `xchg` consente scambi tra operandi che si trovino entrambi in memoria

Istruzioni aritmetiche

Incremento e decremento

- Si tratta di istruzioni ad un solo operando che hanno come effetto l'incremento o il decremento dell'unità dell'operando
- L'operando può essere a 8, 16 o 32 bit
- Il formato è:

```
inc destination
dec destination
```

Istruzioni aritmetiche

Incremento e decremento

- Per esempio:

```
.DATA
count   DW   0
val1    DB   25

.CODE
inc count     ; adesso count vale 1
dec val1      ; adesso val1 vale 24
```

Istruzioni aritmetiche

Incremento e decremento

- Per esempio:

```
.CODE
mov BX, 1057H
mov DL, 5AH
inc BX    ; adesso BX vale 1058H
dec DL    ; adesso DL vale 59H

mov BX, FFFFH
mov DL, 00H
inc BX    ; adesso BX vale 0000H
dec DL    ; adesso DL vale FFH
```

Istruzioni aritmetiche

Addizione

- L'istruzione `add` può essere usata per addizionare operandi a 8, 16 o 32 bit (omogenei)
- Il formato è:

```
add destination, source
```

- L'istruzione `add` può essere applicata nelle cinque forme, analogamente all'istruzione `mov`
- Inoltre, la semantica dell'istruzione è:

```
destination = destination + source
```

Istruzioni aritmetiche

Addizione

- Per esempio:

```
.DATA
b1  DB  0F0H
w1  DW  3746H

.CODE
add AX, DX     ; AX = 1052H DX = AB62H  =>  AX = BBB4H
add BL, CH     ; BL = 76H   CH = 27H    =>  BL = 9DH
add b1, 10H    ;             b1 = F0H   =>  b1 = 00H
add DX, w1     ; w1 = 3746H DX = c8B9H  =>  DX = FFFFH
```

Istruzioni aritmetiche

Addizione con riporto

- La seconda versione dell'addizione è `adc`, addizione con riporto
- La sua sintassi è:

```
adc destination, source
```

- L'operazione che viene eseguita è:

```
destination = destination + source + CF
```

- La differenza tra `add` e `adc` è che nell'ultima istruzione viene aggiunto il contenuto del carry flag (CF)
- L'istruzione `adc` è molto utile quando si devono addizionare numeri "grandi" (con più di 32 bit)

Istruzioni aritmetiche

Addizione con riporto

- Il carry flag può essere manipolato mediante tre istruzioni:

```
stc       set carry flag          CF = 1
clc       clear carry flag        CF = 0
cmc       complement carry flag
```

- Queste istruzioni modificano **soltanto** il carry flag e non gli altri flag del registro

Istruzioni aritmetiche

Addizione con riporto

- Esempio: addizione a 64 bit
- I registri EBX:EAX e EDX:ECX contengono due numeri a 64 bit
- (EBX:EAX indica che i bit più significativi del primo numero sono contenuti in EBX)
- L'addizione a 64 bit può essere eseguita come segue:

```
add EAX, ECX  ; addizione dei 32 bit meno significativi
adc EBX, EDX  ; addizione (con riporto) dei 32 bit
              ; piu' significativi
```

- Il risultato dell'addizione si troverà in EBX:EAX

Istruzioni aritmetiche

Sottrazione

- L'istruzione `sub` viene utilizzata per la sottrazione di numeri a 8, 16 o 32 bit
- La sua sintassi è:

```
sub destination, source
```

- All'operando `destination` viene sottratto l'operando `source` e il risultato viene memorizzato in `destination`
- Cioè:

```
destination = destination - source
```

Istruzioni aritmetiche

Sottrazione

- Qualche esempio:

```
sub   AX, DX   DX = 77ABH  AX = CDEFH  =>  AX = 5644H
sub   BL, CH   CH = 28H    BL = 7BH    =>  BL = 53H
sub   v1, 10H              v1 = F0H    =>  v1 = E0H
sub   DX, w1   w1 = 3764H  DX = C8B9H  =>  DX = 9155H
```

Istruzioni aritmetiche

Sottrazione con prestito

- L'istruzione `sbb` viene utilizzata per la sottrazione con riporto di numeri a 8, 16 o 32 bit
- La sua sintassi è:

```
sbb destination, source
```

- All'operando `destination` viene sottratto l'operando `source` e il risultato viene memorizzato in `destination`, cioè:

```
destination = destination - source - CF
```

- La seconda sottrazione viene effettuata soltanto se CF contiene 1
- Come nel caso di `adc`, anche `sbb` trova utilizzo nei calcoli con numeri grandi

Istruzioni aritmetiche

Negazione

- L'istruzione `neg` viene utilizzata per sottrarre l'operando da zero
- La sua sintassi è:

```
neg destination
```

- In altre parole, viene invertito il segno di un numero intero, cioè:

```
destination = 0 - destination
```

- L'istruzione `neg` ha come effetto l'aggiornamento di tutti i sei flag di stato

Esecuzione condizionata

Istruzione `cmp`

- L'istruzione `cmp` (compare) viene utilizzata per confrontare due operandi

```
cmp destination, source
```

- Effettua la sottrazione dell'operando `source` dall'operando `destination`, ma non altera alcuno dei due operandi
- I registri di flag vengono modificati come nel caso dell'istruzione `sub`
- Lo scopo principale dell'istruzione `cmp` è di modificare i registri di flag in modo che una successiva istruzione di salto condizionato possa verificare lo stato di questi registri
- Nonostante `sub` e `cmp` possano sembrare interscambiabili, `cmp` è di solito più veloce perchè non richiede aggiornamento degli operandi

Esecuzione condizionata

Salti incondizionati

- L'istruzione `jmp` viene utilizzata per informare il processore che la prossima istruzione da eseguire deve essere quella indicata dall'etichetta parte dell'istruzione
- La sua sintassi è:

```
jmp label
```

- Per esempio:

```
     . . .
     mov cx, 10
     jmp Lf
L1:  mov cx, 20
Lf:  mov ax, cx
Lb:  dec cx
     . . .
     jmp Lb
     . . .
```

Esecuzione condizionata

Salti incondizionati

- Nel frammento di codice:

```
    . . .
    mov cx, 10
    jmp Lf
L1: mov cx, 20
Lf: mov ax, cx
Lb: dec cx
    . . .
    jmp Lb
    . . .
```

- L'istruzione `jmp Lf` viene chiamata *forward jump*, mentre l'istruzione `jmp Lb` viene chiamata *backward jump*

Esecuzione condizionata

Salti condizionati

- L'istruzione `j<cond> label` viene utilizzata per informare il processore che la prossima istruzione da eseguire deve essere quella indicata dall'etichetta parte dell'istruzione **soltanto** se una condizione risulta verificata
- La sua sintassi è:

```
j<cond> label
```

- Dove `<cond>` individua la condizione che deve essere verificata affinchè l'esecuzione del programma prosegua all'istruzione `label`
- Di norma la condizione da verificare è il risultato di un test logico-matematico

Esecuzione condizionale

Salti condizionati

- Esempio:

```
    . . .
    cmp al, 0dh
    je cr_ins
    inc cl
    jmp read_ch
cr_ins:
    mov dl, al
    . . .
```

- La coppia di istruzioni `cmp` e `je` permette la realizzazione dell'esecuzione condizionata

Esecuzione condizionale

Salti condizionati

- Alcune delle condizioni più utilizzate:

je	jump if equal
jg	jump if greater
jl	jump if less
jge	jump if greater or equal
jle	jump if less or equal
jne	jump if not equal
jz	jump if zero (ZF = 0)
jnz	jump if not zero (ZF = 1)
jc	jump if carry (CF = 1)
jnc	jump if not carry (CF = 0)

Esecuzione condizionale

Indirizzo relativo

- L'indirizzo specificato in un'istruzione `jmp` non è l'indirizzo assoluto dell'istruzione destinazione
- Si tratta invece dello spostamento relativo (in byte) tra l'istruzione destinazione e l'istruzione successiva all'istruzione di jump
- L'esecuzione di `jmp` implica la modifica di IP (Instruction Pointer) dal valore corrente alla locazione dell'istruzione destinazione

Esecuzione condizionale

Istruzioni di iterazione

- Le iterazioni possono essere implementate mediante le istruzioni di salto
- Per esempio:

```
    mov cl, 50
L1: ...
    ...
    dec cl
    jnz L1
    ...
```

- Le istruzioni (non presenti nell'esempio) vengono ripetute 50 volte fino a quando non è vera la condizione `cl = 0`

Esecuzione condizionale

Istruzioni di iterazione /2

- Le iterazioni possono essere implementate l'istruzione `loop`
- La sua sintassi è:

```
    loop target
```

- Per esempio:

```
    mov cx, 50
L1: ...
    ...
    loop L1
    ...
```

- Il registro `CX` **deve** contenere il numero di iterazioni da effettuare e non può/deve essere modificato

Esecuzione condizionale

Istruzioni di iterazione /3

- Un problema con l'istruzione `loop` consiste nel fatto che se il registro `cx` contiene zero, il numero di iterazioni sarà 2^16
- Questa situazione può essere evitata effettuando prima un test del registro `cx` e condizionando il salto al suo contenuto
- A tale scopo si usa l'istruzione `jcxz`, la cui sintassi è:

```
jcxz target
```

- Che verifica il contenuto del registro `cx` e trasferisce il controllo all'istruzione `target` se il contenuto è zero
- L'istruzione `jcxz` è equivalente a:

```
cmp cx, 0
jz target
```

- `jcxz` **NON** modifica lo status flag

Istruzioni logiche

Introduzione

- Il Pentium fornisce 5 istruzioni logiche, la cui sintassi è:

```
and  destination, source
or   destination, source
xor  destination, source
test destination, source
not  destination
```

AL	BL	and AL, BL	or AL, BL	xor AL, BL	not AL, BL
1010 1110	1111 0000	1010 0000	1111 1110	0101 1110	0101 001
0110 0011	1001 1100	0000 0000	1111 1111	1111 1111	1001 1100
1100 0110	0000 0011	0000 0010	1100 0111	1100 0101	0011 1001
1111 0000	0000 1111	0000 0000	1111 1111	1111 1111	0000 1111

Istruzioni logiche

Uso delle istruzioni logiche

- L'istruzione `and` viene usata per azzerare uno o più bit, per esempio:

```
and ax, 3fffh
```

- Azzera i due bit più significativi del registro `AX`

Istruzioni logiche

Uso delle istruzioni logiche

- L'istruzione `or` viene usata per settare uno o più bit, per esempio:

```
or ax, 8000h
```

- Setta il bit più signicativo del registro `AX`
- Si noti che gli altri 15 bit rimangono inalterati

Istruzioni logiche

Uso delle istruzioni logiche

- L'istruzione `xor` viene usata per invertire uno o più bit, per esempio:

```
xor AX, 5555h
```

- Inverte lo stato dei bit pari del registro `AX`

Istruzioni di shift

Shift logico

- L'istruzione `shl` (shift left) può essere usata per traslare a sinistra un operando destinazione
- A seguito di ogni istruzione di shift sinistro, il bit più significativo viene spostato nel carry flag, mentre il bit meno significativo viene riempito con uno zero
- L'istruzione `shr` funziona analogamente: la traslazione avviene verso destra

```
shl destination, count    shr destination, count
shl destination, CL       shr destination, CL
```

Istruzioni di shift

Shift logico

- L'operando `destination` può essere a 8, 16 o 32 bit ed essere memorizzato in un registro oppure in memoria
- Il secondo operando specifica invece il numero di posizioni da traslare
- Il valore di `count` può essere compreso tra 0 e 31

```
shl AL, 1     10101110   01011100 (CF = 1)
shr AL, 1     10101110   01010111 (CF = 0)

mov CL, 3
shl AL, CL    01101101   01101000 (CF = 1)

mov CL, 5
shr AL, CL    01011001   00000010 (CF = 1)
```

Istruzioni di shift

Shift aritmetico

- Le istruzioni di shift aritmetico `sal` e `sar` possono essere usate per la traslazione (sinistra o destra) di numeri con segno
- Queste istruzioni vengono utilizzate per raddoppiare un numero (shift sinistro di una posizione) oppure per dimezzarlo (shift destro di una posizione)
- Come per lo shift logico, il registro `CL` può contenere il numero di posizioni da traslare

```
sal destination, count    sar destination, count
sal destination, CL       sar destination, CL
```

Istruzioni di shift

Shift aritmetico

- Per esempio:

```
0000 1011    +11
0001 0110    +22
0010 1100    +44
0101 1000    +88

1111 0101    -11
1110 1010    -22
1101 0100    -44
1010 1000    -88
```

Istruzioni di shift

Doppio Shift

- Il Pentium fornisce due istruzioni di shift a 32 o 64 bit
- Queste istruzioni operano su operando di tipo `word` o `doubleword`
- Richiedono tre operandi

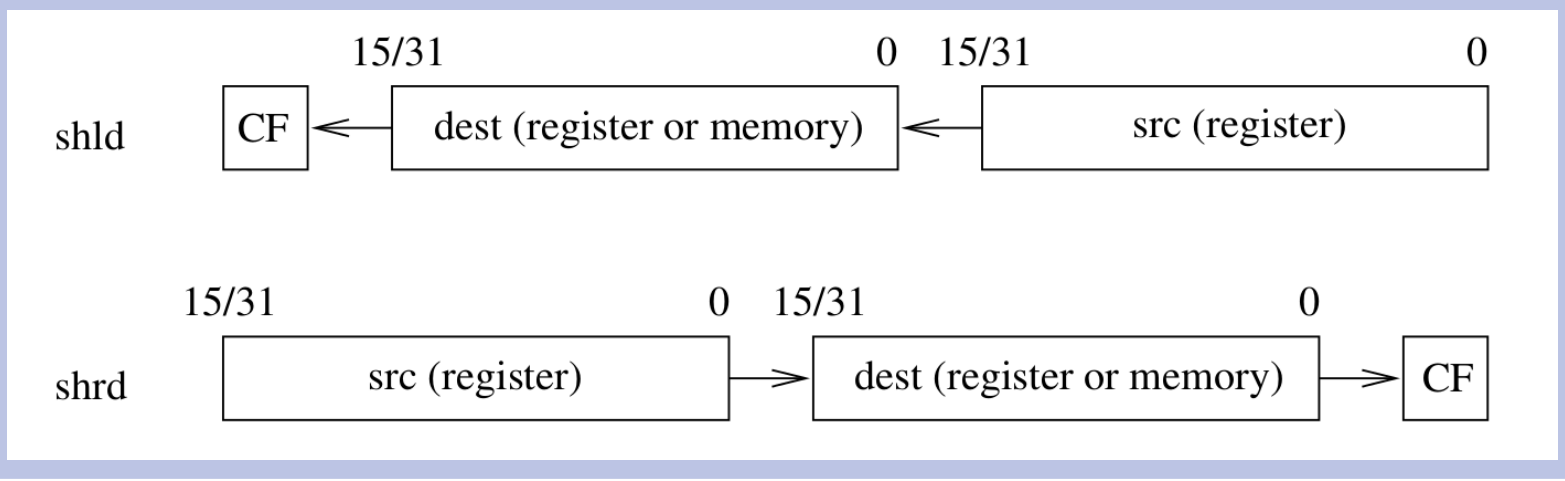
```
shld dest, src, count
shrd dest, src, count
```

- L'operando `dest` può trovarsi in un registro oppure in memoria
- L'operando `src` **deve** trovarsi in un registro

Istruzioni di shift

Doppio Shift

```
shld dest, src, count
shrd dest, src, count
```



Istruzioni di rotazione

Rotazione senza riporto

- In alcune situazioni si desidera mantenere il bit che, invece, viene perso nelle istruzioni di shift
- A tale scopo si possono utilizzare le istruzioni di rotazione
- Ne esistono di due tipi: con o senza coinvolgimento del carry flag (CF)

```
rol (rotate left)
ror (rotate right)
```

- Il formato di queste istruzioni è simile a quello delle istruzioni di shift

```
rol destination, count      ror destination, count
rol destination, CL         ror destination, CL
```

Istruzioni di rotazione

Rotazione senza riporto

- Esempio:

```
rol AL, 1          1010 1110      0101 1101    CF = 1
ror AL, 1          1010 1110      0101 0111    CF = 0

mov CL, 3
rol AL, CL         0110 1101      0110 1011    CF = 1

mov CL, 4
ror AL, CL         0101 1001      1001 0101    CF = 1
```

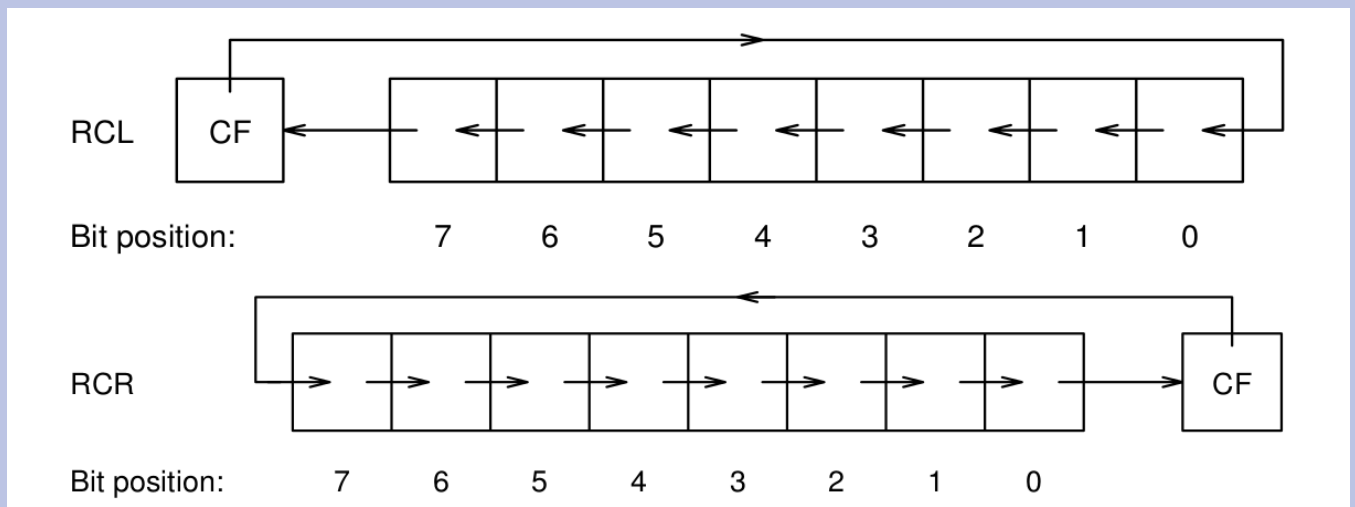
Istruzioni di rotazione

Rotazione con riporto

- Le istruzioni

```
rcl (rotate through carry left)
rcr (rotate through carry right)
```

- Includono il carry flag nel processo di rotazione
- Il bit che viene ruotato fuori viene scritto sul carry flag e quello che si trovava nel carry flag viene scritto al suo posto



Istruzioni di rotazione

Rotazione con riporto

- Esempio:

```
rcl AL, 1          1010 1110      0101 1100    CF = 1
rcr AL, 1          1010 1110      1101 0111    CF = 0

mov CL, 3
rcl AL, CL         0110 1101      0110 1101    CF = 1

mov CL, 4
rcr AL, CL         0101 1001      1001 0101    CF = 1
```

Istruzioni di rotazione

Rotazione con riporto

- Le istruzioni `rcl` e `rcr` forniscono flessibilità nel riarrangiamento di bit
- Inoltre si tratta delle due uniche istruzioni che permettono di usare il carry flag come input
- Come esempio si consideri lo shift destro di un numero a 64 bit memorizzato in EDX:EAX

```
shr EDX, 1
rcr EAX, 1
```

- L'istruzioni `shr` sposta il bit meno significativo di EDX nel carry flag
- L'istruzione `rcr` copia il valore del carry flag nel bit più significativo di EAX durante la rotazione

Istruzioni di rotazione

Applicazione

- Lo shift di un numero a 64 bit può essere eseguito in modi diversi
- `shl` e `rcl`

```
    mov CX, 4
LL: shl EAX, 1
    rcl EDX, 1
    loop LL
```

- il MSB (Most Significant Bit) di EAX viene spostato nel CF
- il bit contenuto in CF viene posto nel LSB (Least Significant Bit) di EDX

Istruzioni di rotazione

Applicazione

- Lo shift di un numero a 64 bit può essere eseguito in modi diversi
- `shld` e `shl`

```
shld EDX, EAX, 4
shl EAX, 4
```

- EAX (operando source di shld) non viene modificato da `shld`

Istruzioni di rotazione

Applicazione

- Lo shift di un numero a 64 bit può essere eseguito in modi diversi
- `shr` e `rcr`

```
    mov CX, 4
LR: shr EDX, 1
    rcr EAX, 1
    loop LR
```

- Il LSB di EDX viene spostato in CF
- Il contenuto di CF viene spostato nel MSB di EAX

Istruzioni di rotazione

Applicazione

- Lo shift di un numero a 64 bit può essere eseguito in modi diversi
- `shrd` e `shr`

```
shrd EAX, EDX, 4
shr EDX, 4
```

- In questo caso EDX non viene modificato da `shrd`

Costanti

- Gli assemblatori mettono a disposizione due **direttive**, `EQU` e `=`, per definire costanti (numeriche o letterali)
- `EQU` può essere usata per definire costanti numeriche o letterali, `=` soltanto per le costanti numeriche
- La sintassi di `EQU` é:

```
name EQU expression
```

- Per esempio:

```
N1 EQU 32
```

Costanti

- Per consuetudine le costanti vengono indicate con lettere maiuscole per distinguerle dalle variabili
- Una volta definita, una costante può essere utilizzata, per esempio:

```
mov CX, N1
. . .
cmp AX, N1
```

- I vantaggi derivanti dall'uso delle costanti è duplice:
  - Aumento della leggibilità del codice
  - Facilità di modifica delle occorrenze ripetute di una costante

Costanti

- L'operando di una costante viene valutato ad *assembly time*
- Per esempio:

```
B EQU 4
H EQU 6
AREA EQU B * H
```

- E' equivalente a definire la costante `AREA` ed assegnarle il valore 24

- Il simbolo cui è stato assegnato un valore numerico o letterale mediante la direttiva `EQU` **NON** può essere ridefinito!

Costanti

- Le parentesi angolari $<$ e $>$ impediscono all'assemblatore di interpretare una eventuale espressione che, al contrario, viene vista come una stringa
- Per esempio

```
B EQU 4
H EQU 6
AREA EQU <B * H>
```

- Indica all'assemblatore che la costante `AREA` è una stringa definita come "$B * H$"

Costanti

Direttiva =

- La direttiva = è simile a `EQU`. La sua sintassi è:

```
name = expression
```

- Esistono due differenze fondamentali:
  - Un simbolo definito mediante la direttiva = può essere ridefinito
  - La direttiva = **non** può essere usata per assegnare stringa o ridefinire istruzioni
- Per esempio:

```
count = 0
. . .
count = 1

J EQU jmp (Ma non J = jmp)
```

Macro

- Le macro forniscono un sistema per rappresentare un blocco di codice con un nome (il cosiddetto *macro name*)
- Quando l'assemblatore incrontra la macro, sostituisce il blocco di codice al nome (*espansione della macro*)
- In altre parole, una macro rappresenta soltanto un comdo meccanismo per la sostituzione di codice
- In linguaggio Assembly, una macro viene definita mediante le direttive `MACRO` e `ENDM`

```
nome_macro MACRO [param1, param2, ...]
           ...
           ENDM
```

- I parametri sono opzionali
- Per invocare una macro è sufficiente usare il suo nome con gli eventuali parametri

Macro

Macro senza parametri

- Un esempio:

```
moltAX_per_16 MACRO
              sal AX, 4
              ENDM
```

- Il codice della macro consiste di una sola istruzione, che verrà sostituita laddove la macro sarà invocata, cioè:

```
. . .
mov AX, 21
moltAX_per_16
. . .
```

Macro

Macro senza parametri

```
. . .
mov AX, 21
moltAX_per_16
. . .
```

- Al momento dell'espansione della macro diventerà:

```
. . .
mov AX, 21
sal AX, 4
. . .
```

Macro

Macro con parametri

- L'impiego di parametri in una macro consente di scrivere codice più flessibile e utile
- Mediante l'uso dei parametri, una macro può operare su operandi di vario tipo (byte, word, doubleword, ...) memorizzati in un registro oppure in memoria
- Il numero di parametri è limitato da quanti ne entrano in una linea di codice

```
molt_per_16 MACRO opsal
            sal opsal, 4
            ENDM
```

- L'operando `opsal` può essere qualsiasi tipo di operando purché compatibile con l'istruzione `sal`

Macro

Macro con parametri

- Per esempio:

```
molt_per_16 DL        ; sal DL, 4

molt_per_16 count     ; sal count, 4
```

- Dove la variabile `count` può avere dimensioni varie (byte, word, doubleword, ...)

- Altro esempio:

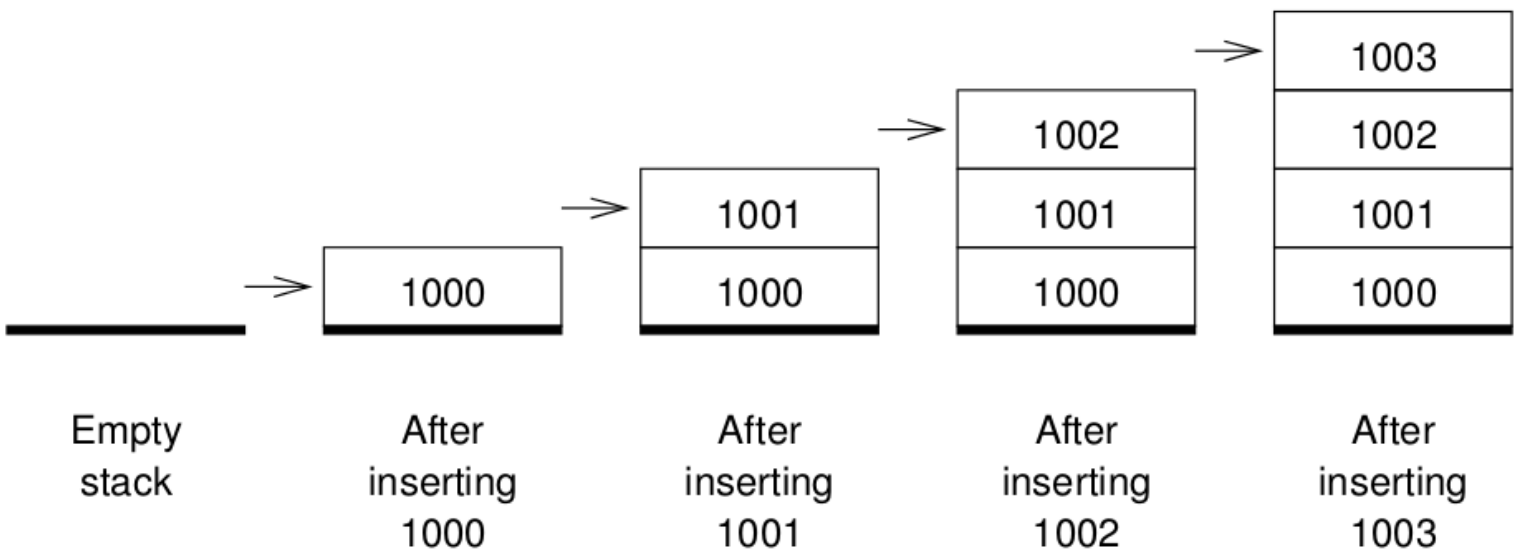
```
m2mxchg MACRO op1, op2
        xchg AX, op1
        xchg AX, op2
        xchg AX, op1
        ENDM
```

- Effettua lo scambio di due operandi che si trovano entrambi in memoria, lasciando il registro AX inalterato

Stack

- Uno stack è una struttura dati di tipo LIFO (Last In First Out)
- Si tratta cioè di una struttura in cui l'entità ultima ad entrare è anche la prima ad uscire
- Due operazioni si possono associare ad uno stack: inserimento e rimozione
- L'unico elemento direttamente accessibile è quello posto in cima allo stack (TOS, Top Of Stack)
- Nella terminologia dello stack, le operazioni di inserimento e rimozione prendono il nome di `push` e `pop`

Stack


1000
1001
1000
1002
1001
1000
1003
1002
1001
1000
Empty stack
After inserting 1000
After inserting 1001
After inserting 1002
After inserting 1003

Stack

- Lo stack viene implementato nello spazio di memoria denominato *segmento stack*
- Nel Pentium, lo stack viene implementato mediante i registri SS ed (E)SP
- Il TOS, che punta all'ultimo elemento inserito nello stack, viene indicato da `SS:SP` (SS indica l'inizio del segmento stack, SP è il registro che fornisce l'offset dell'ultimo elemento inserito)

Stack

Caratteristiche implementative

- Sullo stack vengono salvati soltanto word e doubleword (mai byte)
- Lo stack cresce verso indirizzi di memoria inferiori (cresce verso il "basso")
- Il TOS punta sempre all'ultimo elemento inserito sullo stack
- E' possibile stabilire la quantità di memoria riservata per lo stack mediante l'istruzione:

```
.STACK 100H
```

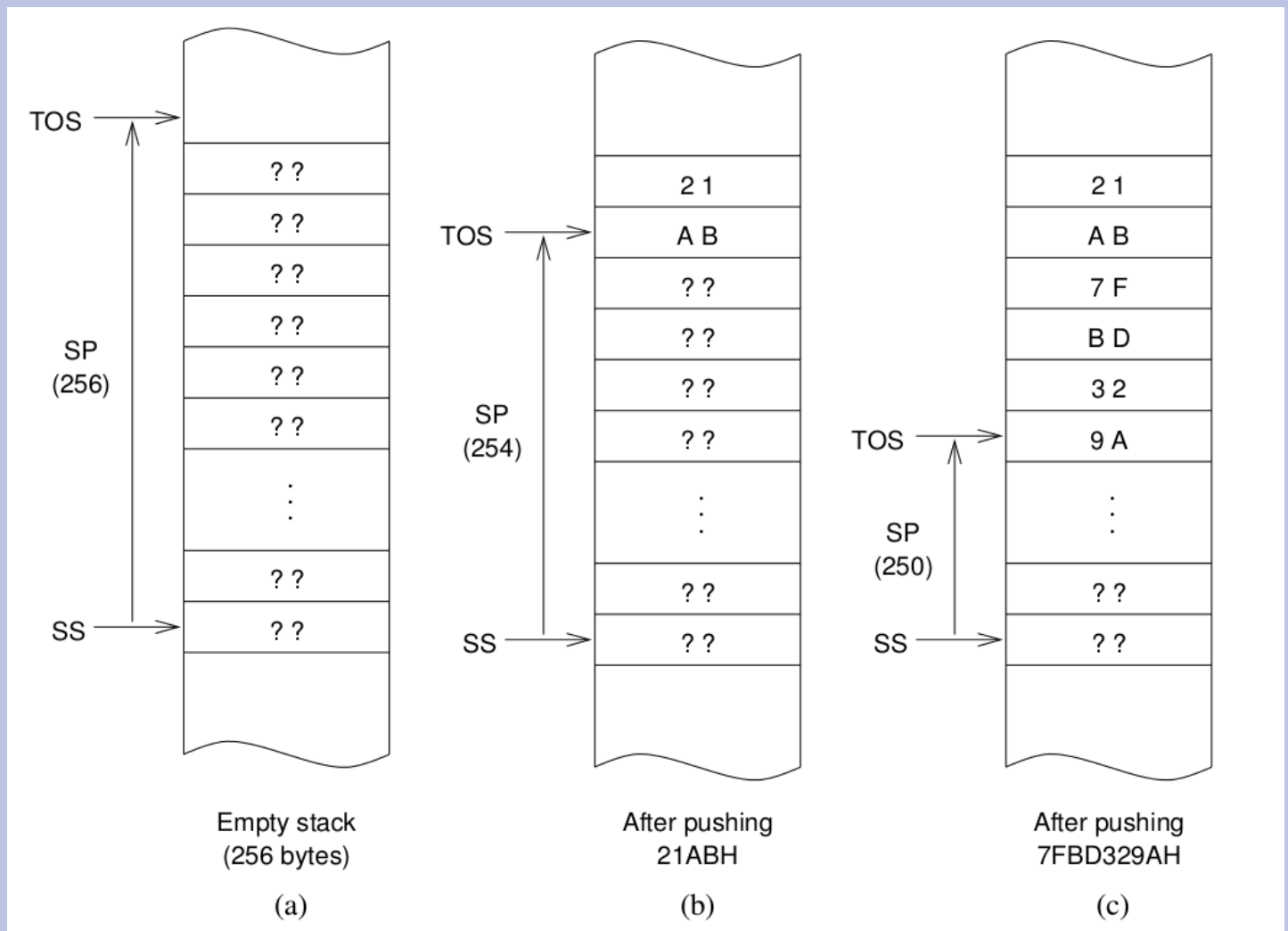
Stack

Caratteristiche implementative

```
.STACK 100H
```

- Crea uno stack vuoto di 256 byte
- Quando lo stack viene inizializzato, TOS punta ad un byte esterno all'area riservata allo stack
- Pertanto leggere da uno stack vuoto comporta un errore noto come *stack underflow*
- La condizione di stack pieno viene indicata dallo stato del registro SP, che contiene `0000H`
- Inserire ulteriori elementi provoca un errore noto come *stack overflow*

Stack


TOS
SP
(256)
SS
? ?
? ?
? ?
? ?
? ?
? ?
? ?
? ?
Empty stack
(256 bytes)
(a)
TOS
SP
(254)
SS
2 1
A B
? ?
? ?
? ?
? ?
? ?
? ?
After pushing
21ABH
(b)
TOS
SP
(250)
SS
2 1
A B
7 F
B D
3 2
9 A
? ?
? ?
After pushing
7FBD329AH
(c)

Stack

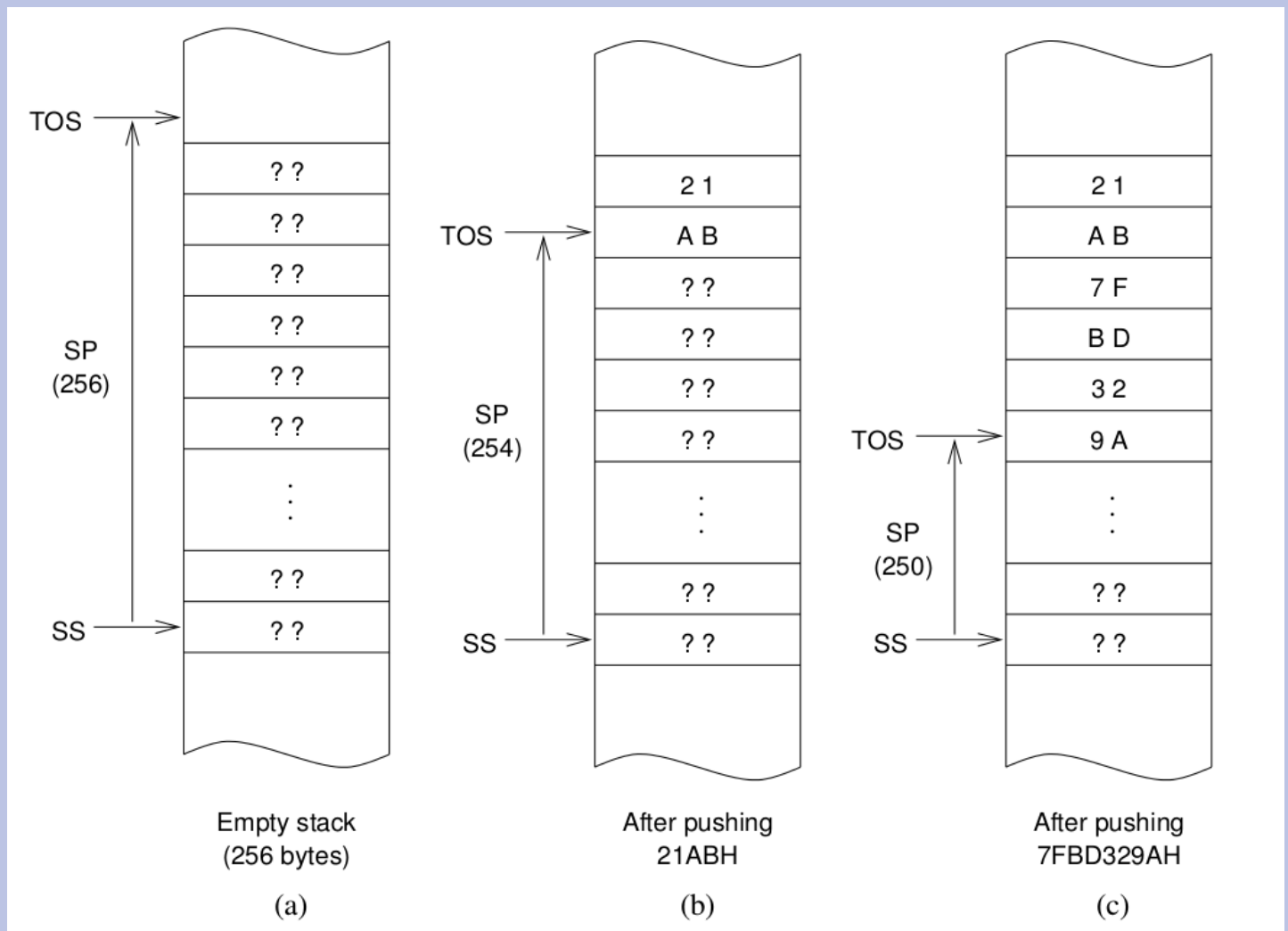
Operazioni

```
push source
pop destination
```

- Gli operandi di queste istruzioni possono essere registri (di uso generale) a 16 o 32 bit, registri di segmento, word o doubleword presenti in memoria
- L'operando `source` può essere anche un operando immediato a 8, 16 o 32 bit
- Si considerino per esempio le istruzioni:

```
push 21abh
push 7fbd329ah
```

Stack


TOS
SP
(256)
SS
? ?
? ?
? ?
? ?
? ?
? ?
? ?
? ?
Empty stack
(256 bytes)
(a)
TOS
SP
(254)
SS
2 1
A B
? ?
? ?
? ?
? ?
? ?
? ?
After pushing
21ABH
(b)
TOS
SP
(250)
SS
2 1
A B
7 F
B D
3 2
9 A
? ?
? ?
After pushing
7FBD329AH
(c)

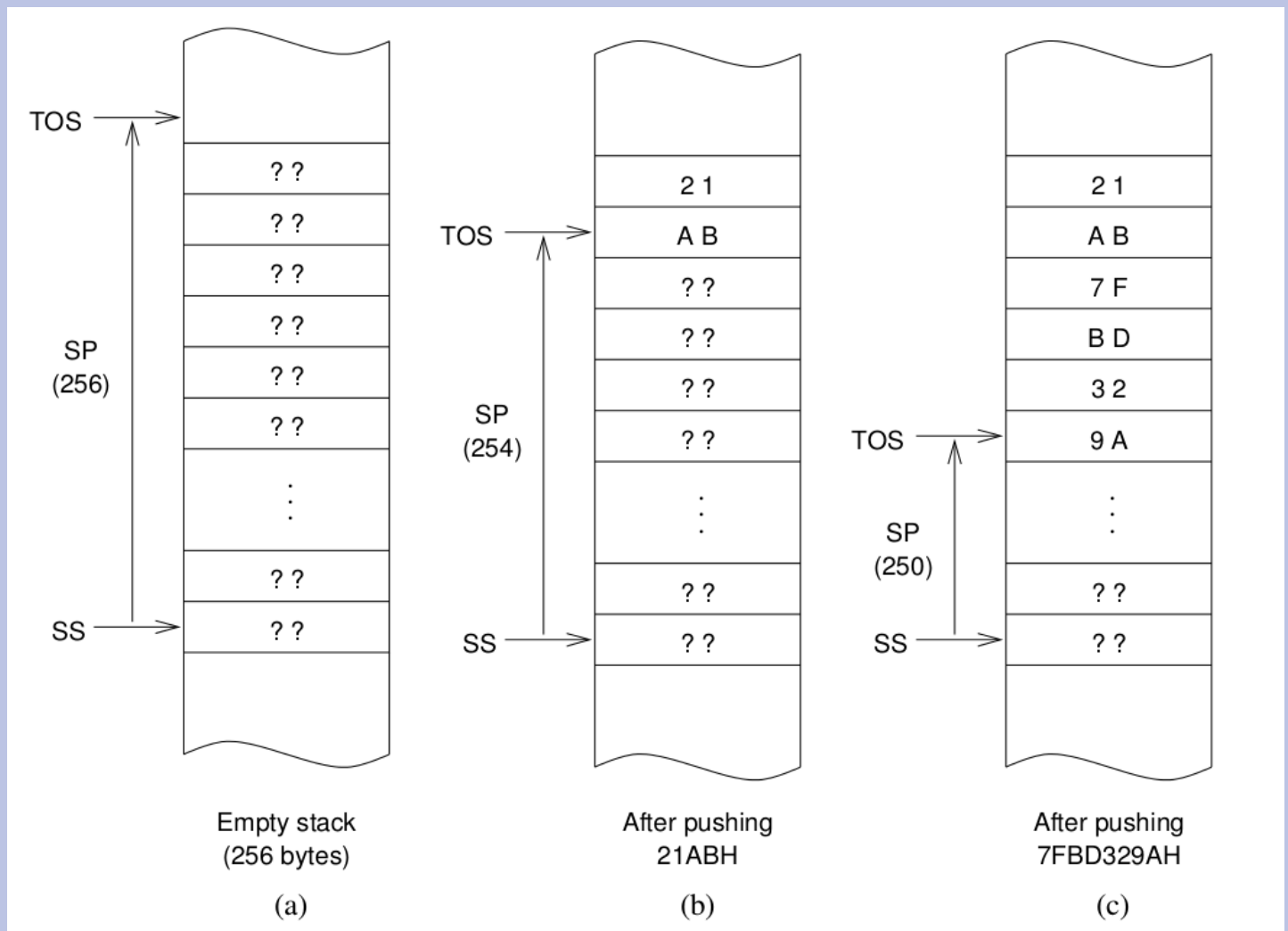
Stack

- L'istruzione:

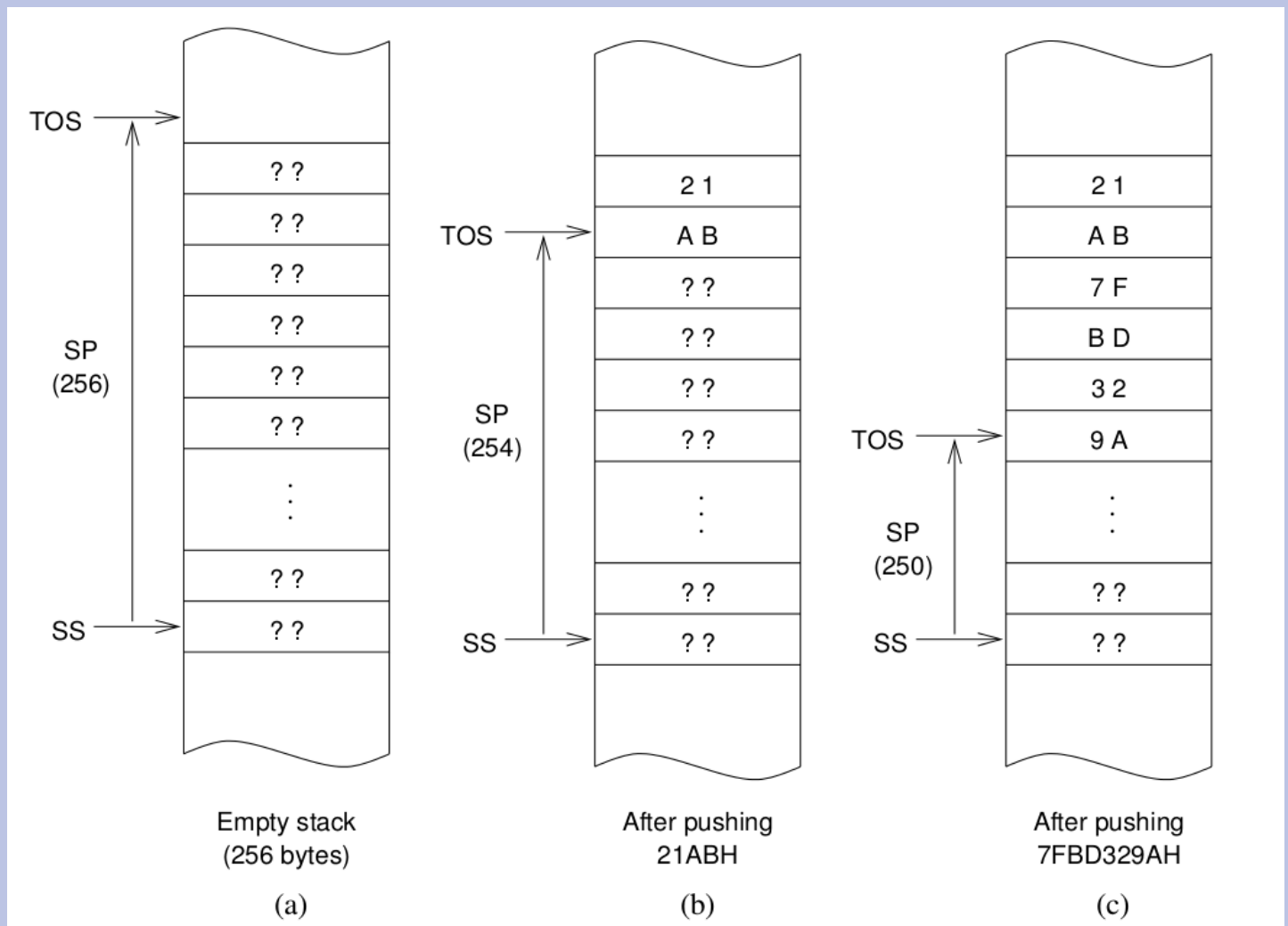
```
pop EBX
```

- Ha come conseguenze:
  - Il registro EBX riceve il dato `7fbd329ah`
  - Lo stack si riporta nella condizione (b)

Stack


TOS
SP
(256)
SS
? ?
? ?
? ?
? ?
? ?
? ?
? ?
? ?
Empty stack
(256 bytes)
(a)
TOS
SP
(254)
SS
2 1
A B
? ?
? ?
? ?
? ?
? ?
? ?
After pushing
21ABH
(b)
TOS
SP
(250)
SS
2 1
A B
7 F
B D
3 2
9 A
? ?
? ?
After pushing
7FBD329AH
(c)

Stack


TOS
SP
(256)
SS
? ?
? ?
? ?
? ?
? ?
? ?
? ?
? ?
Empty stack
(256 bytes)
(a)
TOS
SP
(254)
SS
2 1
A B
? ?
? ?
? ?
? ?
? ?
? ?
After pushing
21ABH
(b)
TOS
SP
(250)
SS
2 1
A B
7 F
B D
3 2
9 A
? ?
? ?
After pushing
7FBD329AH
(c)

Stack

- `push source16`
  - $SP = SP - 2$
  - SP viene decrementato di 2 per modificare TOS. Il dato a 16 bit viene copiato sullo stack al nuovo TOS. Lo stack cresce di 2 byte
- `push source32`
  - $SP = SP - 4$
  - SP viene decrementato di 4 per modificare TOS. Il dato a 32 bit viene copiato sullo stack al nuovo TOS. Lo stack cresce di 4 byte
- `pop dest16`
  - $SP = SP + 2$
  - Il dato posto in TOS viene copiato in `dest16`. SP viene incrementato di 2 per aggiornare TOS. Lo stack decresce di 2 byte
- `pop dest32`
  - $SP = SP + 4$
  - Il dato posto in TOS viene copiato in `dest32`. SP viene incrementato di 4 per modificare TOS. Lo stack decresce di 4 byte

Stack

Operazioni sui flag

- Le operazioni `push` e `pop` non possono essere usate per lavorare sul flag register
- A tale scopo esistono due istruzioni speciali

```
pushf
popf
```

- Per operare sul flag register a 32 bit (EFLAGS) si usano invece:

```
pushfd
popfd
```

Stack

Operazioni sui registri general-purpose

- Il Pentium fornisce due istruzioni speciali (`pusha` e `popa`) per le operazioni sui registri general-purpose
- `pusha` copia sullo stack i registri AX, CX, DX, BX, SP, BP, SI e DI
- `popa` copia dallo stack gli stessi registri (ad eccezione di SP)
- Le istruzioni per i registri a 32 bit sono `pushad` e `popad`
- Queste istruzioni sono **molto** utili al momento di invocare una procedura, oppure al ritorno da una procedura

Utilizzi dello stack

- Area di deposito temporaneo di dati
- Per il trasferimento del controllo di un programma
- Per il passaggio di parametri durante l'invocazione di una procedura

Utilizzi dello stack

Area di deposito temporaneo di dati

- Lo stack può essere usato come area di deposito temporaneo di dati
- Si consideri per esempio il problema dello scambio del valore di due variabili a 32 bit poste in memoria
- L'istruzione `xchg val1, val2` **non** è consentita!
- Il codice

```
mov EAX, val1
mov EBX, val2
mov val1, EBX
mov val2, EAX
```

- Funziona, ma richiede due registri a 32 bit e 4 operazioni e NON salva il contenuto originario dei registri coinvolti

Utilizzi dello stack

Area di deposito temporaneo di dati /2

- Una prima soluzione, che ha il vantaggio di salvare il contenuto originario dei registri, è la seguente:

```
push EAX
push EBX

mov EAX, val1
mov EBX, val2
mov val1, EBX
mov val2, EAX

pop EAX
pop EBX
```

Utilizzi dello stack

Area di deposito temporaneo di dati /3

- Una soluzione (decisamente) migliore è:

```
push val1
push val2
pop  val1
pop  val2
```

Utilizzi dello stack

Trasferimento di controllo

- Al momento dell'invocazione di una procedura, l'indirizzo di ritorno dell'istruzione viene memorizzato nello stack
- Così facendo alla conclusione delle operazioni della procedura è possibile restituire il controllo al programma chiamante

Passaggio di parametri

- Lo stack è uno strumento per il passaggio di parametri ad una procedura
- In tal senso viene usato estensivamente dai programmi di alto livello

Procedure

- Unità di codice che svolge un compito particolare
- Le procedure ricevono una lista di argomenti ed effettuano un calcolo sulla base degli argomenti
- Per esempio, in C:

```
int somma (int x, int y)
{
  return x + y;
}
```

- I parametri `x` e `y` vengono detti *formali*

```
totale = somma (num1, num2);
```

- Per distinguerli dai parametri *attuali* `num1` e `num2` usati nel calcolo della funzione

Procedure

Meccanismi per il passaggio dei parametri

Passaggio per valore

- Alla funzione invocata viene passato soltanto il valore dell'argomento
- In questo modo il valore dei parametri attuali non viene modificato nella funzione invocata

Passaggio per riferimento

- Alla funzione invocata viene passato soltanto l'indirizzo (puntatore) degli argomenti
- La funzione invocata può modificare il contenuto di questi parametri
- Queste modifiche saranno visibili dalla funzione chiamante

Procedure

Passaggio per riferimento

- Per esempio:

```
void swap (int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

- Questa funzione verrà invocata mediante:

```
swap (&val1, &val2);
```

Procedure

Direttive Assembly

- Le direttive per definire le procedure in Assembly sono:

```
proc-name PROC NEAR
. . .
proc-name ENDP
```

```
proc-name PROC FAR
. . .
proc-name ENDP
```

- La procedura viene invocata mediante l'istruzione:

```
call proc-name
```

Procedure

Direttive Assembly

- Le direttive NEAR e FAR permettono di definire procedure di tipo NEAR o di tipo FAR
- Una procedura si definisce di tipo NEAR quando la procedura chiamante e la procedura chiamata si trovano nello stesso segmento di codice
- Se al contrario si trovano in differenti segmenti di codice, la procedura deve essere definita come FAR
- Le direttive NEAR e FAR possono essere omesse (di default una procedura si intende di tipo NEAR)

Procedure

Passaggio di parametri

- In Assembly, la procedura chiamante pone tutti i parametri necessari alla procedura chiamata in un'area mutuamente accessibile (registri o memoria)
- Il passaggio di parametri in Assembly può essere di due tipi: mediante registro o mediante stack

Procedure

Passaggio di parametri mediante registri

- Secondo questa modalità, i parametri vengono passati utilizzando i registri general-purpose
- Per esempio:

```
main PROC
    . . .
    mov CX, num1
    mov DX, num2
    . . .
    call sum
    . . .
main ENDP

sum PROC
    mov AX, CX
    add AX, DX
    ret
sum ENDP
```

Procedure

Passaggio di parametri mediante registri

Vantaggi

- Metodo semplice per il passaggio di un piccolo numero di parametri
- Metodo veloce perché tutti i parametri si trovano nei registri

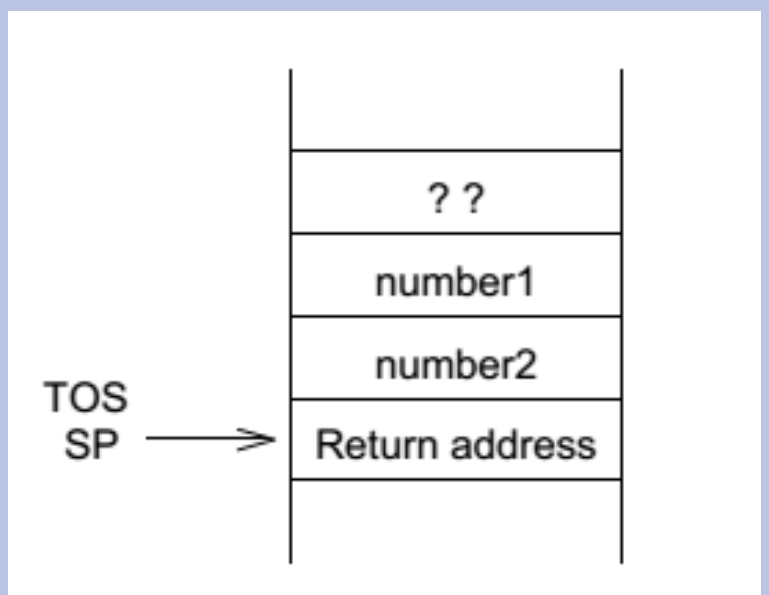
Svantaggi

- Soltanto pochi parametri possono essere passati mediante i registri, visto che i registri general-purpose sono in numero limitato
- I registri general-purpose sono di solito già utilizzati dalla procedura chiamante per altri scopi (è pertanto necessario salvare il contenuto di questi registri sullo stack prima dell'utilizzo dei registri e recuperarne il valore dopo)

Procedure

Passaggio di parametri mediante stack

- I parametri vengono posti sullo stack prima che la procedura venga invocata
- La procedura invocata deve provvedere ad effettuare il pop dei parametri copiati nello stack
- Il problema risiede nel fatto che, dopo i parametri passati, nello stack viene copiato anche l'indirizzo di ritorno della procedura chiamante



Procedure

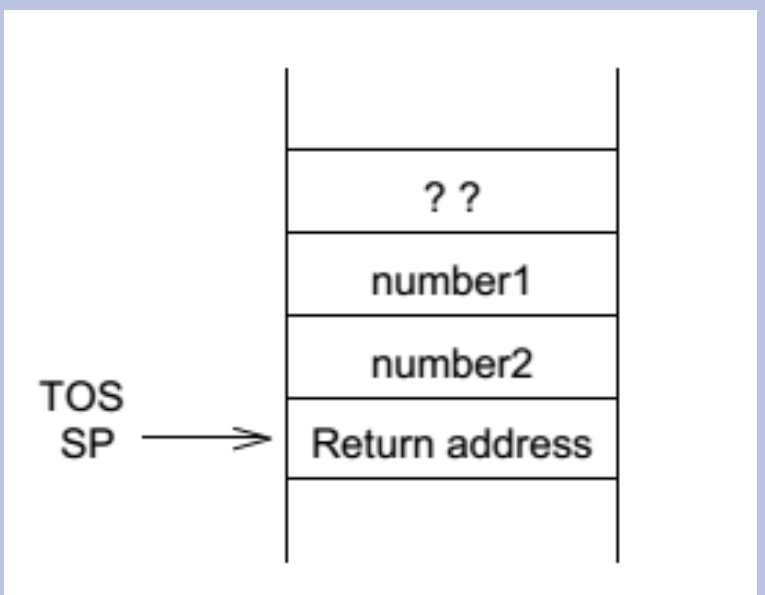
Passaggio di parametri mediante stack

- La lettura degli argomenti passati alla procedura non è immediata perché bisogna prima estrarre il valore di `IP` per poi passare alla lettura dei parametri
- L'estrazione di `IP` avviene mediante `pop` su un registro (che non deve essere manipolato)
- Alla conclusione delle operazioni, il valore di `IP` deve essere ripristinato nello stack in modo da consentire la ripresa delle operazioni della procedura chiamante
- Si consideri inoltre che i parametri passati alla procedura devono essere estratti dallo stack, con la conseguenza che si impegnano dei registri general-purpose per ospitare i parametri
- Il modo migliore di operare consiste nel lasciare i parametri sullo stack e leggerli quando necessario

Procedure

Passaggio di parametri mediante stack

- Lo stack è una sequenza di locazioni di memoria
- Di conseguenza, $SP + 2$ punta all'elemento `number2`
- $SP + 4$ punta all'elemento `number1`

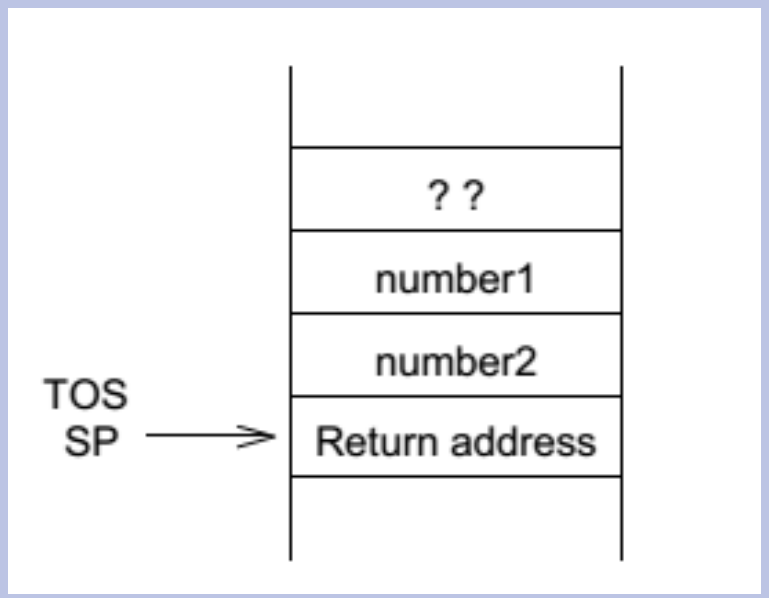


Procedure

Passaggio di parametri mediante stack

- Pertanto l'elemento `number2` si ottiene:

```
add SP, 2
mov BX, [SP]
```



Procedure

Passaggio di parametri mediante stack

- Un'alternativa migliore consiste nell'utilizzo del registro BP per specificare l'offset
- D'altra parte, BP viene utilizzato per l'accesso ai parametri, quindi il suo contenuto deve essere preservato
- Pertanto si può fare:

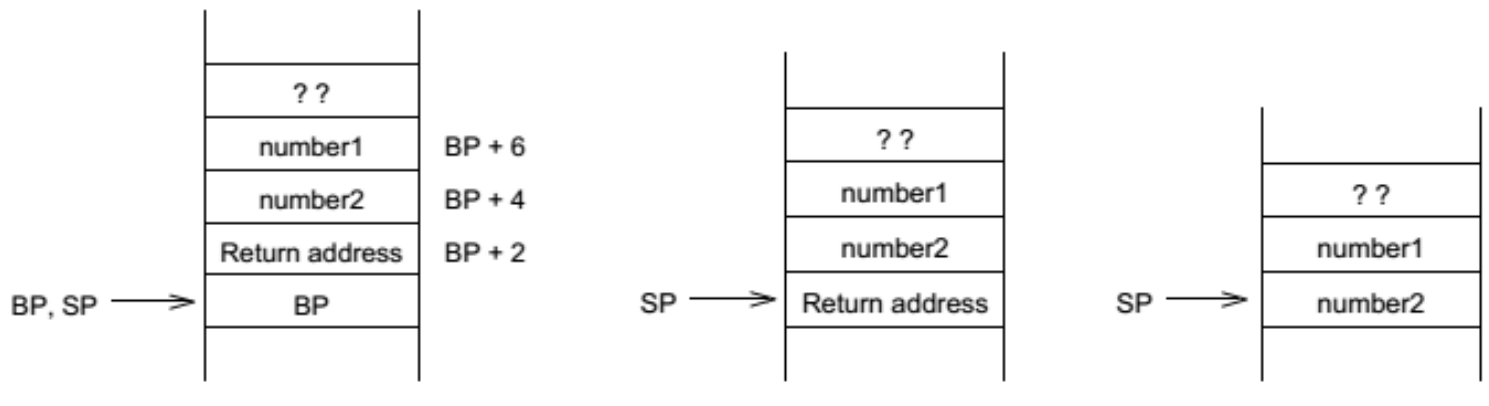
```
push BP
mov BP, SP
mov AX, [BP+2]
```

- Prima di completare l'esecuzione della procedura chiamata, si deve ripristinare il contenuto di BP mediante l'istruzione pop

Procedure

Passaggio di parametri mediante stack

- Le modifiche subite dallo stack sono in definitiva:



Procedure

Passaggio di parametri mediante stack: esempio

```
main PROC
    . . .
    push number1
    push number2
    call sum
    add SP, 4
    . . .
main ENDP
sum PROC
    pop AX          ; contiene IP procedura chiamante
    pop BX
    pop CX
    add BX, CX
    push AX         ; contiene IP procedura chiamante
    ret
sum ENDP
```

Procedure

Passaggio di parametri mediante stack: esempio

```
main PROC
    . . .
    push number1
    push number2
    call sum
    . . .
main ENDP
sum PROC
    pop AX          ; contiene IP procedura chiamante
    pop BX
    pop CX
    add BX, CX
    push AX         ; contiene IP procedura chiamante
    add SP, 4
    ret
sum ENDP
```

Procedure

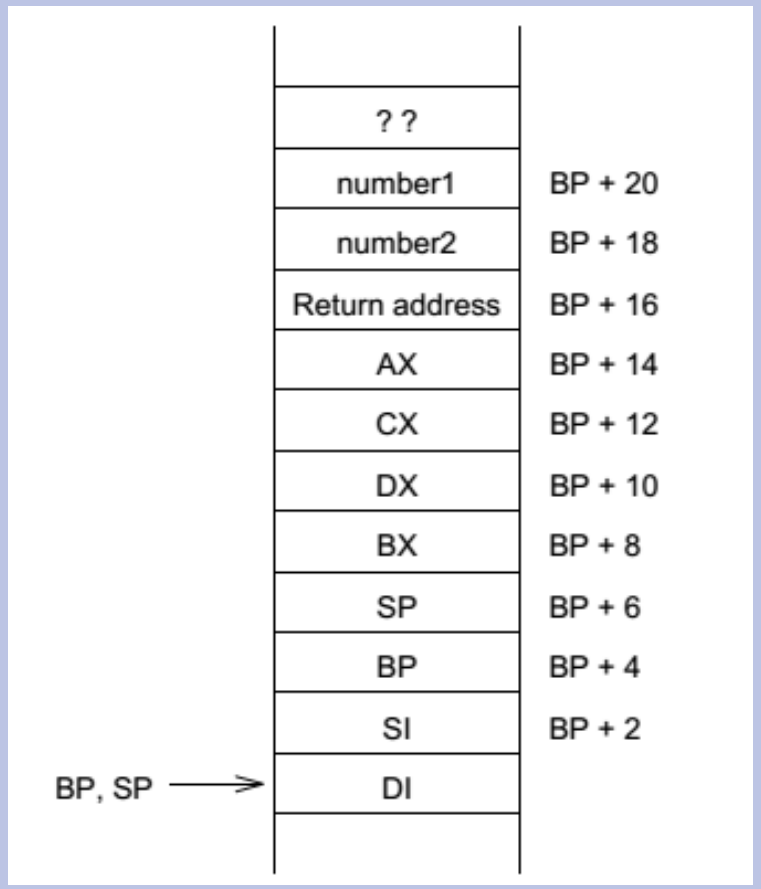
Preservare lo stato della procedura chiamante

- E' fondamentale preservare il contenuto dei registri nel corso di una invocazione a procedura
- Si consideri per esempio:

```
    . . .
    mov CX, count
L0: call sub01
    . . .
    . . .
    loop L0
    . . .
```

Procedure

Utilizzo di `pusha` per preservare lo stato dei registri



Procedure

Variabili locali

```
int sub01(int a, int b)
{
  int temp, N;
  . . .
  . . .
}
```

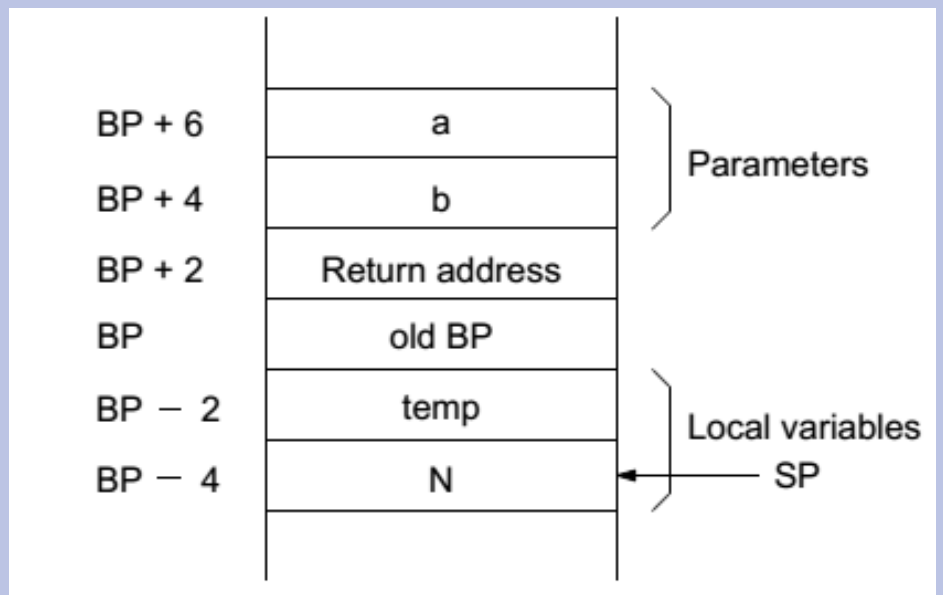
Procedure

Variabili locali

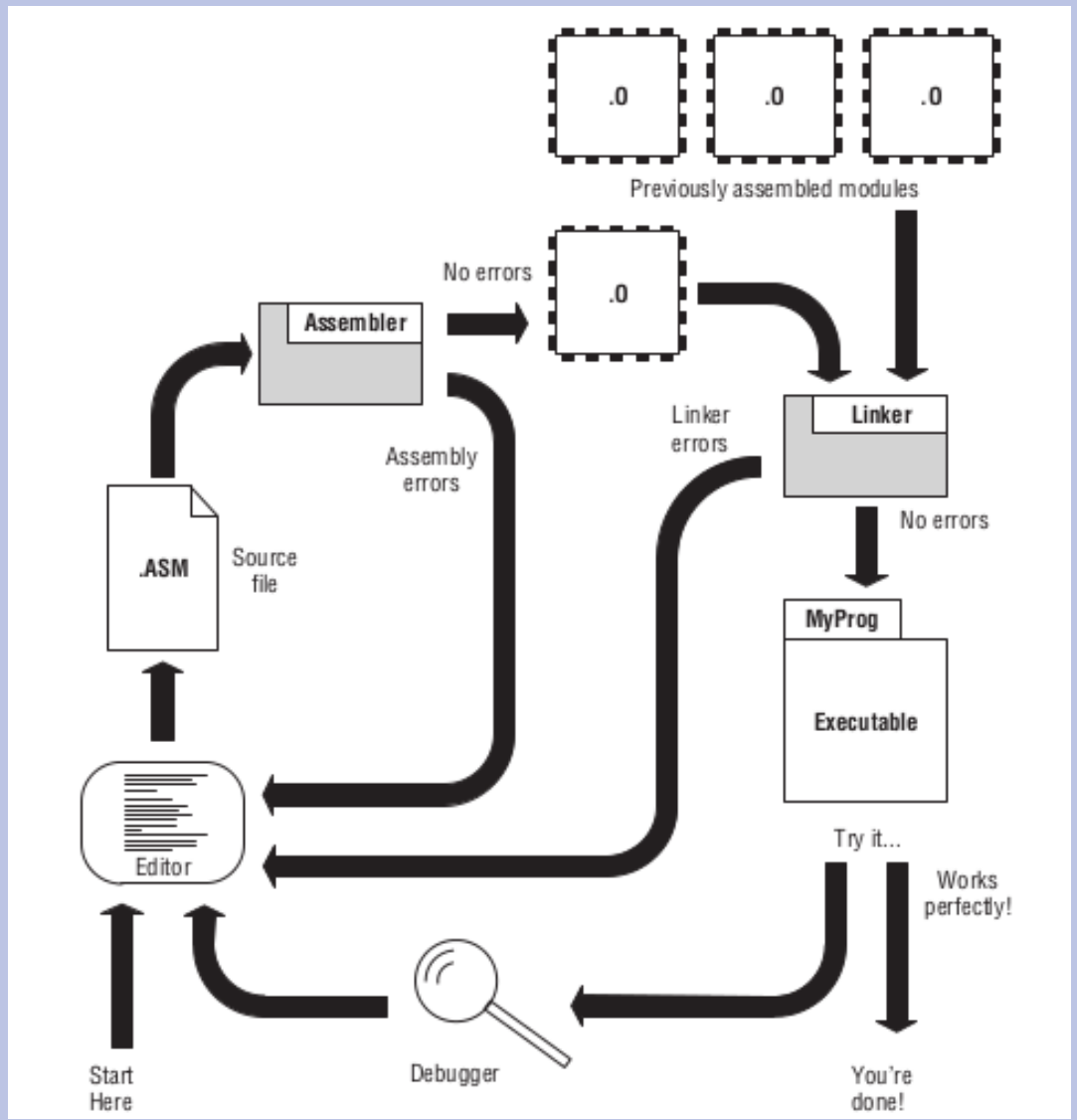
- Le variabili `temp` e `N` sono variabili locali che hanno senso fin tanto che la proceura `sub01` viene invocata
- Vengono distrutte dopo la terminazione della procedura
- Pertanto queste variabili sono dinamiche
- Non è il caso di riservare spazio nel segmento dati per queste variabili perché:
  - L'allocazione di spazio è permanente (anche dopo la terminazione della procedura chiamata)
  - Non funzionerebbe con le procedure ricorsive
- Per questi motivi, lo spazio per le variabili locali viene riservato nello stack

Procedure

Variabili locali



Sviluppo di un programma Assembly



Ordine delle operazioni

- Editing
- Traduzione (viene prodotto un codice oggetto)
- Linking (il codice oggetto viene "linkato" ad altri eventuali codici oggetto)
- Si produce un codice eseguibile che può essere mandato in esecuzione

Strumenti

- Editor di testo (per esempio `gedit`)
- Assemblatore (`nasm`)
- Linker (presente nelle installazioni Linux: `ld`)
- Terminale

Come procedere

Creazione del codice oggetto (assemblaggio)

```
nasm -f elf64 -g -F stabs ex01.asm
```

Linking

```
ld ex01.o -o ex01.x
```

Esecuzione

```
./ex01.x
```

Creazione del codice oggetto (assemblaggio)

```
nasm -f elf64 -g -F stabs ex01.asm
```

- `-f elf64`: formato del codice oggetto (`-f elf` nel caso di un'architettura a 32 bit)
- `-g`: le informazioni per il debug devono essere incluse nel codice oggetto
- `-F stabs`: formato per le informazioni del debug
- `ex01.asm`: nome del codice sorgente

Linking

```
ld ex01.o -o ex01.x
```

- `-o ex01.x`: permette di specificare il nome dell'eseguibile

Esecuzione

```
./ex01.x
```

- `./`: dice alla shell che il programma da eseguire si trova nella directory corrente

Formato ELF

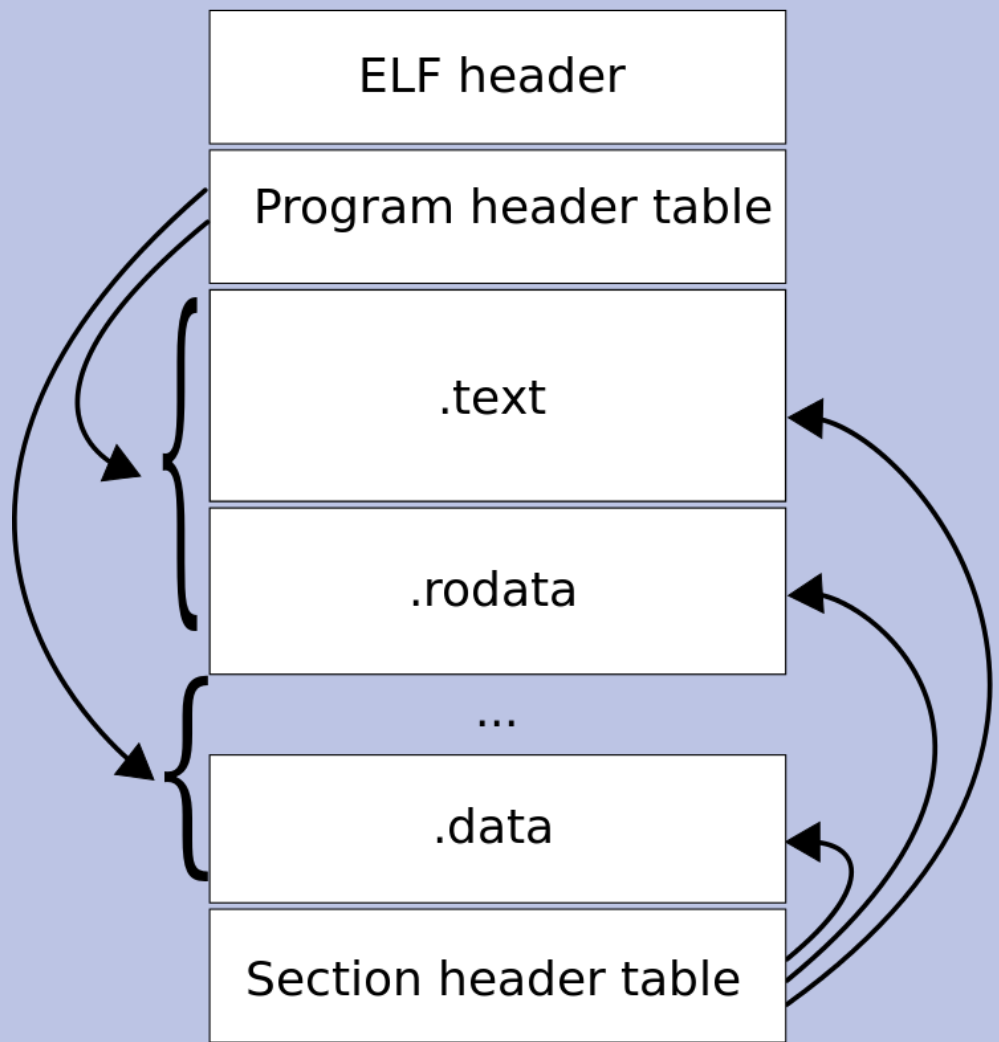
Executable and Linkable Format

Formato per eseguibili, codici oggetto, librerie e `core dump`

- Ogni file ELF si compone di un header seguito dai dati
  - Una program header table, che descrive i segmenti
  - Una section header table, che descrive le sezioni
  - I dati riferiti dalle due tabelle
- I segmenti contengono informazioni come stack, dati, codice
- Le sezioni contengono informazioni relative al linking e alla rilocazione

Formato ELF

Executable and Linkable Format



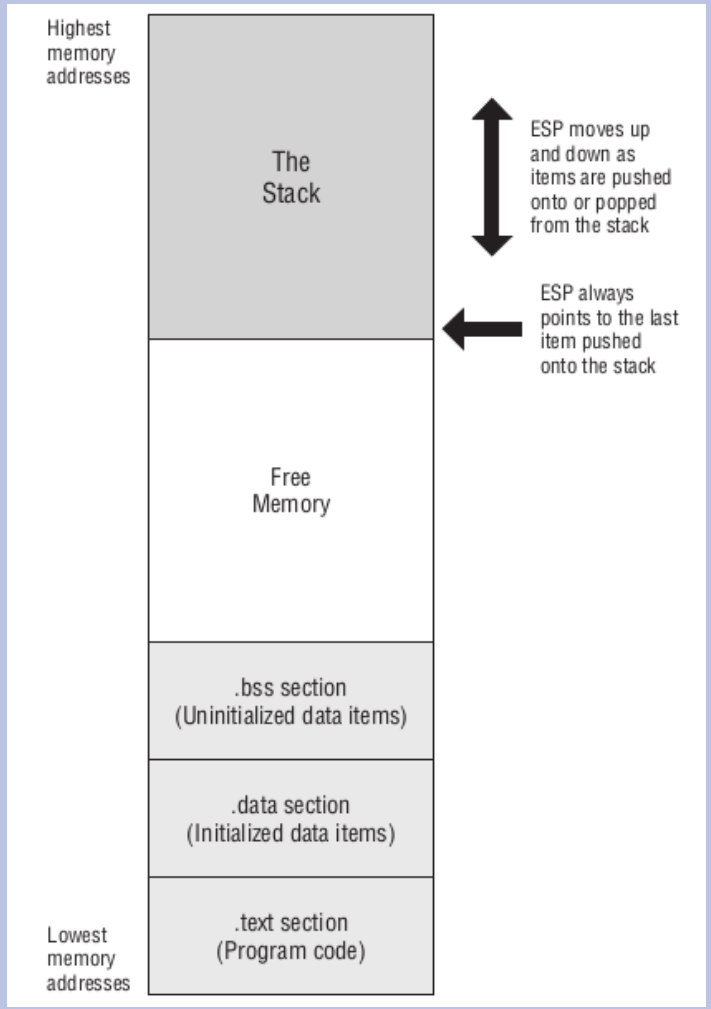
Formato ELF

Executable and Linkable Format

Sezione	Descrizione
.text	Codice
.data	Variabili
.bss	Variabili non inizializzate
.rodata	Costanti
.comment	Commenti inseriti dal linker
.stab	Informazioni per il debug

Formato ELF

Executable and Linkable Format

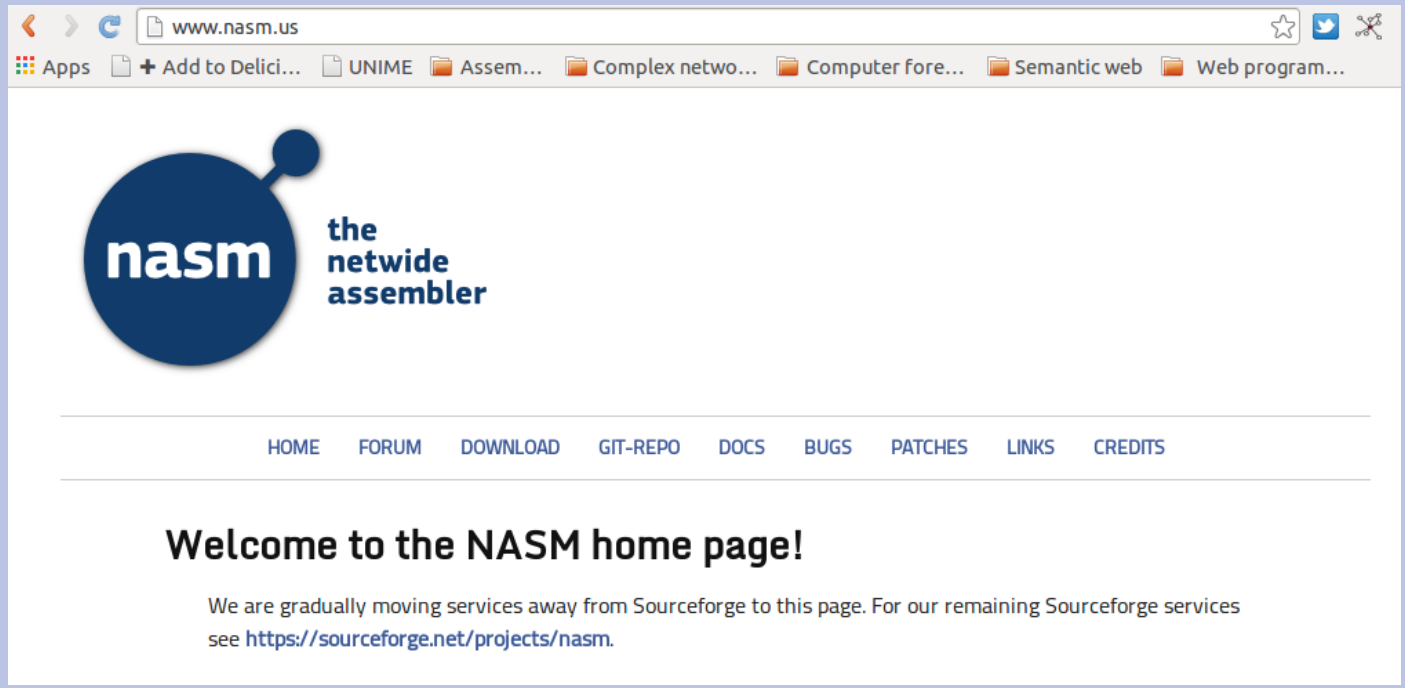


Struttura di un programma Assembly

```
SECTION .data
msg1: db "Hello World!", 10
len1: equ $-msg1

SECTION .bss

SECTION .text

global _start

_start:
  mov eax, 4
  mov ebx, 1
  mov ecx, msg1
  mov edx, len1
  int 80h

  mov eax, 1
  mov ebx, 0
  int 80h
```

NASM

The Netwide Assembler



Versioni disponibili

- DOS
- Linux
- MacOSX
- Win32

Documentazione

- Formato HTML
- Formato pdf

- Installazione
- Opzioni di compilazione
- Caratteristiche dell'assemblatore
- Preprocessore
- Formati di output

L'installazione dipende dal sistema operativo

- DOS
- Linux
- MacOSX
- Win32

Installazione in ambiente Unix/Linux

- `sh configure` oppure `./configure`
- `make`
- `sudo make install`

Installazione in ambiente MS-DOS

- Decomprimere il pacchetto
- Copiare il file `nasm.exe` in una directory opportuna
- Aggiungere il percorso alla variabile PATH

Installazione in ambiente MS-Windows

- Scaricare il file `nasm-2.11.02-win32.zip`
- Decomprimere
- Copiare il file `nasm.exe` in una directory opportuna
- Aggiungere il percorso alla variabile PATH

Installazione in ambiente MS-Windows (seconda possibilità)

- Scaricare il file `nasm-2.11.02-installer.exe`
- Mandare in esecuzione il programma di installazione
- Seguire le indicazioni

Interrupt 80h

- L'istruzione `int80h` viene usata per provocare una interruzione software e invocare i servizi di Linux
- In Linux esistono *system calls* che forniscono funzioni fondamentali per accedere ai servizi hardware (disco, video, porte, I/O)
- L'invocazione di una particolare funzione dell'istruzione `int80h` viene effettuata assegnando un particolare valore al registro `EAX`

Interrupt 80h

Servizio 1

- Uscita dal processo corrente e restituzione del controllo al processo che ha invocato il processo corrente
- Nel registro `EBX` si pone il valore 0 per indicare una terminazione senza anomalie

```
mov eax, 1
mov ebx, 0
int 80h
```

Interrupt 80h

Servizio 3

- Lettura
- Nel registro `EBX` si pone il valore 0 per indicare il terminale di input (tastiera)
- Nel registro `ECX` si pone un puntatore ad un'area di memoria destinata a contenere i caratteri inseriti
- Nel registro `EDX` si pone il numero massimo di caratteri da inserire

```
mov eax, 3
mov ebx, 0
mov ecx, stringa
mov edx, 100
int 80h
```

Interrupt 80h

Servizio 3

```
; ex-read.asm

SECTION .data

stringa: db 100

SECTION .bss

SECTION .text

global _start

_start:
  mov eax, 3
  mov ebx, 0
  mov ecx, stringa
  mov edx, 100
  int 80h
  mov eax, 4
  mov ebx, 1
  mov ecx, stringa
  mov edx, 100
  int 80h

  mov eax, 1
  mov ebx, 0
  int 80h
```

Interrupt 80h

Servizio 4

- Scrittura
- Nel registro `EBX` si pone il valore 1 per indicare il terminale di output (monitor)
- Nel registro `ECX` si pone un puntatore ad un'area di memoria che contiene i caratteri da stampare
- Nel registro `EDX` si pone il numero massimo di caratteri da stampare

```
mov eax, 4
mov ebx, 1
mov ecx, stringa
mov edx, 100
int 80h
```

Interrupt 80h

Servizio 5

- Open di un file
- Nel registro `EBX` si pone il filename (terminato da 0)
- Nel registro `ECX` si pone la modalità di accesso
- Nel registro `EDX` si pone il permesso sul file, nel caso in cui lo si crei

```
mov eax, 5
mov ebx, "pippo.txt",0
mov ecx, 1
mov edx, 0
int 80h
```

Interrupt 80h

Servizio 5

Modalità di accesso	Valore	Descrizione
O_RDONLY	0	Apertura in sola lettura
O_WRONLY	1	Apertura in sola scrittura
O_RDWR	2	Apertura in lettura e scrittura
O_CREAT	256	Il file viene creato se non esistente
O_APPEND	2000h	Il file viene aperto in modalità append

Interrupt 80h

Servizio 5

Permessi	Valore	Descrizione
S_ISUID	04000	Set user ID on execution
S_ISGID	02000	Set group ID on execution
S_ISVTX	01000	On directories, restricted deletion flag
S_IRWXU	0700	Owner has read, write and execute permission
S_IRUSR	0400	Owner has read permission
S_IWUSR	0200	Owner has write permission
S_IXUSR	0100	Owner has execute permission
S_IRWXG	070	Group has read, write and execute permission
S_IRGRP	040	Group has read permission
S_IWGRP	020	Group has write permission
S_IXGRP	010	Group has execute permission
S_IRWXO	07	Others have read, write and execute permission
S_IROTH	04	Others have read permission
S_IWOTH	02	Others have write permission
S_IXOTH	01	Others have execute permission

Interrupt 80h

Servizio 6

- Chiusura di un file

```
mov EAX, 6
int 80h
```

Interrupt 80h

Servizio 5

```
; ex-wf.asm

section .data

    filename db "./output.txt", 0
    text db "Ciao a tutti", 0
    textlen equ $ - text

section .text

global _start

_start:
        mov EAX, 5
        mov EBX, filename
        mov ECX, 1
        mov EDX, 400h
        int 80h
        mov EBX, EAX
        mov EAX, 4
        mov ECX, text
        mov EDX, textlen
        int 80h

        mov EAX, 6
        int 80h

        mov eax, 1
        int 80h
```

Modifica dei permessi di accesso ad un file (chmod)

- Nel registro `EAX` si pone il valore 15
- Nel registro `EBX` si pone il filename (terminato da 0)
- Nel registro `ECX` si pone il permesso sul file

```
mov eax, 15
mov ebx, "pippo.txt",0
mov ecx, 664o
int 80h
```

Interrupt 80h

Servizio 15

```
; ex-chmod.asm

section .data

    filename db "./prova.txt", 0

section .text

global _start

_start:
        mov EAX, 15
        mov EBX, filename
        mov ECX, 777o
        int 80h
                                    mov EAX, 6
                                    int 80h

                                    mov eax, 1
                                    int 80h
```

Creazione di una directory (mkdir)

- Nel registro `EAX` si pone il valore 39
- Nel registro `EBX` si pone il dirname (terminato da 0)
- Nel registro `ECX` si pone il permesso sul file

```
mov eax, 39
mov ebx, "dirname",0
mov ecx, 664o
int 80h
```

Interrupt 80h

Servizio 39

```
; ex-mkdir.asm

section .data

    filename db "./provadir", 0

section .text

global _start

_start:
        mov EAX, 39
        mov EBX, filename
        mov ECX, 660o
        int 80h
        mov EAX, 6
        int 80h

        mov eax, 1
        int 80h
```

Cambio della directory corrente (chdir)

- Nel registro `EAX` si pone il valore 12
- Nel registro `EBX` si pone il dirname (terminato da 0)

```
mov EAX, 12
mov EBX, dirname
int 80h
```

Creazione di un file

- Nel registro `EAX` si pone il valore 8
- Nel registro `EBX` si pone il filename (terminato da 0)
- Nel registro `ECX` si pongono i permessi del nuovo file

```
mov EAX, 8
mov EBX, filename
mov ECX, 660o
int 80h
```

Interrupt 80h

Servizi 12 e 8

```
; ex-chdir.asm

section .data

    dirname  db "./provadir", 0
    filename db "./pippo.txt", 0

section .text

global _start

_start:
        mov EAX, 12
        mov EBX, dirname
        int 80h

        mov EAX, 8
        mov EBX, filename
        mov ECX, 660o
        int 80h

        mov EAX, 6
        int 80h

        mov eax, 1
        int 80h
```

Link di un file

- Nel registro `EAX` si pone il valore 9
- Nel registro `EBX` si pone il filename originale (terminato da 0)
- Nel registro `ECX` si pone il filename copia (terminato da 0)

```
mov EAX, 9
mov EBX, orig
mov ECX, dest
int 80h
```

Interrupt 80h

Servizio 9

```
; ex-link.asm

section .data

    dirname  db "./provadir", 0
    sfile    db "orig.txt",0
    dfile    db "dest.txt",0

section .text

global _start

_start:
        mov EAX, 12
        mov EBX, dirname
        int 80h
```

```
mov EAX, 9
mov EBX, sfile
mov ECX, dfile
int 80h

mov eax, 1
int 80h
```

Linux System Call Reference

http://www.salvorosta.it/low/shared/syscalltable.html

Interrupt 80h

Conta dei caratteri inseriti da tastiera

```
; ex-read2.asm

SECTION .data

stringa db 100

SECTION .bss

SECTION .text

global _start

_start:
      mov eax, 3
      mov ebx, 0
      mov ecx, stringa
      mov edx, 100
      int 80h
      mov esi, eax

      mov eax, 4
      mov ebx, 1
      mov ecx, stringa
      mov edx, esi
      int 80h

      mov eax, 1
      mov ebx, 0
      int 80h
```

Dati non inizializzati

- Come già visto, la sezione dati è suddivisa in due parti
- Nella sezione `.data` si trovano i dati inizializzati
- Nella sezione `.bss` si definiscono invece i dati **non** inizializzati
- Per i dati inizializzati si utilizzano le direttive `define` (`db`, `dw`, `dd`, `dq`, `dt`)
- Per i dati non inizializzati si usano invece le seguenti direttive:

```
resb Reserve byte
resw Reserve word
resd Reserve double
resq Reserve quad
rest Reserve ten bytes
```

Dati non inizializzati

- Per esempio:

```
section .data
num1 db 100

section .bss
vecb resb 10
vecw resw 10
vecd resd 10
vecq resq 10
vect rest 10
```

Indirizzi

Sintassi di NASM

- NASM utilizza una speciale sintassi per specificare gli indirizzi
- Non supporta la direttiva `offset`
- Il nome della variabile viene considerato come rappresentante l'indirizzo di memoria al quale si trova il dato
- Per esempio

```
mov ebx, offset num1 (valido in MASM)
mov ebx, num1        (valido in NASM)

mov ebx, num1        (valido in MASM)
mov ebx, [num1]      (valido in NASM)
```

Interrupt 80h

Conta dei caratteri inseriti da tastiera

```
; ex-assign.asm

SECTION .data
msgb db 'Valore:   '
lmb equ $-msgb
strcr db 0dh,0ah
lcr equ $-strcr

SECTION .bss
      valb resb 5

SECTION .text
global _start
_start:
      mov eax, 4
      add eax, 30h
      mov [valb], eax
      mov eax, 4
      mov ebx, 1
      mov ecx, msgb
      mov edx, lmb
      int 80h

      mov eax, 4
      mov ebx, 1
      mov ecx, valb
      mov edx, 5
      int 80h

      mov eax, 1
      mov ebx, 0
      int 80h
```

Interrupt 80h

Conta dei caratteri inseriti da tastiera

```
SECTION  .data
    msg1  db 'Inserire un numero: '
    lmsg1 equ $-msg1
    msg2  db 'Numero inserito: '
    lmsg2 equ $-msg2

SECTION .bss
    num resb 5

SECTION .text
       global _start
_start:
       mov eax, 4
       mov ebx, 1
       mov ecx, msg1
       mov edx, lmsg1
       int 80h
```

```
mov eax, 3
mov ebx, 2
mov ecx, num
mov edx, 5
int 80h

mov eax, 4
mov ebx, 1
mov ecx, msg2
mov edx, lmsg2
int 80h

mov eax, 4
mov ebx, 1
mov ecx, num
mov edx, 5
int 80h

mov eax, 1
mov ebx, 0
int 80h
```

Interrupt 80h

Subroutine

```
; strlen.asm

SECTION .data
msg     db      'Ciao a tutti!', 0Ah
len     db      0
SECTION .text
global  _start

_start:

    mov     eax, msg
    call    strlen

    mov     [len], eax

    mov     eax, 4
    mov     ebx, 1
    mov     ecx, msg
    mov     edx, [len]
    int     80h

    mov     ebx, 0
    mov     eax, 1
    int     80h
; ---------------------------
strlen:
    push    ebx
    mov     ebx, eax

Ln: cmp     byte [eax], 0
    jz      Lf
    inc     eax
    jmp     Ln

Lf: sub     eax, ebx
    pop     ebx
    ret
```

Interrupt 80h

Include di file esterni

```
; strlen-inc.asm

%include 'string.asm'
SECTION .data
msg     db      'Ciao a tutti!', 0Ah
len     db      0
SECTION .text
global  _start

_start:

    mov     eax, msg
    call    strlen

    mov     [len], eax

    mov     eax, 4
    mov     ebx, 1
    mov     ecx, msg
    mov     edx, [len]
    int     80h

    mov     ebx, 0
    mov     eax, 1
    int     80h
; --------------------------
; string.asm
; --------------------------

strlen:
    push    ebx
    mov     ebx, eax

Ln: cmp     byte [eax], 0
    jz      Lf
    inc     eax
    jmp     Ln

Lf: sub     eax, ebx
    pop     ebx
    ret
```

Interrupt 80h

Include di file esterni /2

```
; conta.asm

%include 'funzioni.asm'

SECTION .text
global _start

_start:

    mov ecx, 0

L0: inc  ecx
    mov eax, ecx
    add eax, 30h
    push eax
    mov eax, esp
    call crprint

    pop eax
    cmp ecx, 10
    jne L0

    call theend
```

Interrupt 80h

Include di file esterni /2

```
strlen:
    push    ebx
    mov     ebx, eax

Ln: cmp     byte [eax], 0
    jz      Lf
    inc     eax
    jmp     Ln

Lf: sub     eax, ebx
    pop     ebx
    ret
```

Interrupt 80h

Include di file esterni /2

```
; --------------------------
print:
    push    edx
    push    ecx
    push    ebx
    push    eax
    call    strlen

    mov     edx, eax
    pop     eax

    mov     ecx, eax
    mov     ebx, 1
    mov     eax, 4
    int     80h

    pop     ebx
    pop     ecx
    pop     edx
    ret
```

Interrupt 80h

Include di file esterni /2

```
; --------------------------
crprint:
    call print

    push eax
    mov eax, 0Ah
    push eax
    mov eax, esp
    call print
    pop eax
    pop eax
    ret
```

Interrupt 80h

Include di file esterni /2

```
; ---------------------------
theend:
    mov eax, 1
    mov ebx, 0
    int 80h
    ret
```

Interrupt 80h

Stampa a video di numeri

```
%include        'funzioni.asm'

SECTION .text
global  _start

_start:

    mov ecx, 0

L0: inc     ecx
    mov     eax, ecx
    add     eax, 30h
    push    eax
    mov     eax, esp
    call    crprint

    pop     eax
    cmp     ecx, 10
    jne     L0

    call    theend
```