Processore PC

- Faremo riferimento al cosiddetto processore PC, che rappresenta una schematizzazione dei processori a 32 bit presenti nei Personal Computer.
- Il processore è costituito da due unità fondamentali:
  - la ALU (Arithmetic and Logic Unit)
  - la FPU (Floating Point Unit)
- Spazio di indirizzamento
  - gli indirizzi di memoria sono espressi su 32 bit, quindi lo spazio di memoria è costituito da 2^32 locazioni di un byte
    - gli indirizzi validi vanno da 0x00000000 a 0xFFFFFFFF
  - gli indirizzi di I/O sono espressi su 16 bit, lo spazio di I/O è costituito da 2^16 *porte* di un byte
    - gli indirizzi validi vanno da 0x0000 a 0xFFFF

Registri interni

- La ALU si presenta al programmatore come un insieme di 10 registri
  - 8 registri generali (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
  - 2 registri di stato (EIP, EFLAG)
- Gli 8 registri generali sono a 32 bit e possono essere usati per memorizzare operandi o per contenere indirizzi
- Alcuni svolgono, spesso, specifiche funzioni:
  - EAX funge principalmente da accumulatore, e EDX da estensione dell'accumulatore
  - i registri ESI e EDI fungono principalmente da registri indice
  - il registro EBP funge da registro base per una zona della pila
  - il registro ESP funge da puntatore alla cima della pila
- Il registro di stato EIP contiene l'indirizzo dell'istruzione successiva
- Il registro di stato EFLAG contiene (fra l'altro):
  - i flag aritmetici CF (carry), ZF (zero), SF (sign), e OF (overflow)
  - il flag PF (parity)

Operandi e modalità di indirizzamento

- Gli operandi manipolati dalle istruzioni della ALU possono essere lunghi un byte, due byte (parola), quattro byte (parola lunga)
  - in ogni caso l'indirizzo dell'operando coincide con quello del byte di indirizzo più piccolo, che contiene la parte meno significativa dell'operando stesso
- Modalità di indirizzamento: una istruzione può avere un operando nell'istruzione stessa, in un registro della ALU, oppure in memoria
  - tre modi di indirizzamento: *immediato*, di *registro*, di *memoria*
- Un indirizzo viene calcolato attraverso la seguente regola generale:

  indirizzo = | base + indice*scala + spiazzamento | $_{\text{modulo } 2^{32}}$

  - *base* è il contenuto di uno dei registri generali della ALU
  - *indice* è il contenuto di uno dei registri generali
  - *scala* è un fattore che può valere 1, 2, 4, o 8
  - *spiazzamento* è un numero naturale a 8 o 32 bit
- Uno o due degli addendi precedenti possono anche mancare

Modalità di indirizzamento

- *Indirizzamento diretto*: è presente solo lo spiazzamento che rappresenta l'indirizzo
- *Indirizzamento indiretto*: è indicato solo il registro base (che funge da registro puntatore), il cui contenuto rappresenta l'indirizzo
- *Indirizzamento modificato con registro base*: sono indicati il registro base e lo spiazzamento (il registro base tipicamente contiene l'indirizzo di partenza di una struttura e lo spiazzamento consente di riferire la singola componente);
- *Indirizzamento modificato con registro indice (indirizzamento indiciato)*: sono indicati il registro indice e lo spiazzamento (lo spiazzamento tipicamente rappresenta l'indirizzo di partenza di un vettore, il registro indice consente di riferire la singola componente, e scala rappresenta il numero di byte di ogni componente)
- *Indirizzamento bimodificato senza spiazzamento*: sono indicati un registro base e un registro indice (con scala)
- *Indirizzamento bimodificato con spiazzamento*: sono indicati un registro base, un registro indice (con scala) e lo spiazzamento
- EBP non può essere usato come registro puntatore e ESP non può essere usato come registro indice

Modalità di indirizzamento

- Per le istruzioni di controllo, l'indirizzo di salto può essere costituito dall'operando (a 32 bit) il cui indirizzo viene determinato con le regole precedenti, oppure può essere ottenuto come

  $$\text{indirizzo} = | \text{EIP} + \text{spiazzamento} |_{\text{modulo } 2^{32}}$$

  dove spiazzamento è un numero intero a 8 o 32 bit (indirizzamento relativo rispetto a EIP)

- Per le istruzioni di ingresso/uscita l'indirizzo della porta può essere
  - specificato direttamente (indirizzamento diretto) per indirizzi minori di 256
  - contenuto in DX (indirizzamento indiretto)

Istruzioni macchina

- *src* e *dst* indicano il registro o le locazioni di memoria in cui possono trovarsi gli operandi sorgente e destinazione, secondo le regole di indirizzamento (*src* può anche essere immediato)
  - in caso di restrizioni *accumulatore* denota che l'operando può risiedere solo in EAX, AX, o AL; *memoria* denota che l'operando può risiedere solo in memoria; *registro* denota che l'operando può risiedere solo in un registro; *porta* denota che l'operando può risiedere solo in una porta di I/O
- Ogni istruzione ha un codice operativo (CO), e come parte del CO il termine *lun* specifica la lunghezza degli operandi
  - B = byte, W = parola, L = parola lunga
- Le istruzioni possono essere raggruppate in
  - istruzioni operative
    - istruzioni per il trasferimento dei dati
    - istruzioni aritmetiche
    - istruzioni logiche
    - istruzioni di traslazione/rotazione
    - istruzioni di ingresso/uscita

  istruzioni di controllo
    istruzioni di salto
    istruzioni per i sottoprogrammi
    istruzioni per il meccanismo di interruzione
    istruzioni per il controllo del processore

Istruzioni per il trasferimento dei dati

- Trasferiscono una copia dell'operando sorgente nell'operando destinazione
- ***MOVlun src, dst***
  - (MOVe) trasferisce una copia dell'operando sorgente nella destinazione
- ***PUSHlun src***
  - (PUSH) salva in pila una copia dell'operando sorgente (che deve essere a 16 o 32 bit); più in dettaglio: decrementa il contenuto di ESP di 2 o di 4, memorizza una copia dell'operando src nella doppia o quadrupla locazione il cui indirizzo è contenuto in ESP
- ***PUSHAL***
  - (PUSH All Long) salva in pila una copia del contenuto degli 8 registri generali
- ***PUSHFL***
  - (PUSH Flag Long) salva in pila una copia del registro EFLAG
- ***POPlun dst***
  - (POP) estrae dalla pila una parola o una parola lunga e la trasferisce nella destinazione; più in particolare: trasferisce nella destinazione una copia del contenuto della doppia o quadrupla locazione il cui indirizzo è contenuto in ESP, e incrementa di 2 o di 4 il contenuto di ESP

Istruzioni per il trasferimento dei dati

- ***POPAL***
  - (POP All Long) estrae dalla pila 8 parole lunghe e le trasferisce nei registri generali
- ***POPFL***
  - (POP Flag Long) estrae dalla pila una parola lunga e la trasferisce in EFLAG
- ***XCHGlun src, dst***
  - (eXCHanGe) scambia i contenuti della sorgente e della destinazione
- ***LEAL memoria, registro***
  - (Load Effective Address) calcola l'indirizzo di memoria dell'operando sorgente e lo carica nel registro destinatario (sempre a 32 bit)

Istruzioni aritmetiche

- ***ADDlun src, dst***
  - (ADD) sostituisce nella dst la somma fra l'operando src e quello dst (risultato significativo si per numeri interi che naturali)
- ***ADClun src, dst***
  - (ADD with Carry) sostituisce nella dst la somma fra l'operando src e quello dst e il contenuto del flag CF (risultato significativo si per numeri interi che naturali)
- ***INClun dst***
  - (INCrement) incrementa di 1 il contenuto della dst (significativo sia per i numeri naturali che per gli interi)
- ***SUBlun src, dst***
  - (SUBtract) sostituisce nella destinazione la differenza tra l'operando dst e quello src (significativo sia per i naturali che per gli interi)
- ***SBBlun src, dst***
  - (SUBtract with Borrow) sostituisce nella destinazione la differenza tra l'operando dst e l'insieme dell'operando src e del contenuto del flag CF (significativo sia per i naturali che per gli interi)
- ***DEClun dst***
  - (DECrement) decrementa di 1 il contenuto di dst (significativo sia per i naturali che per gli interi)

Istruzioni aritmetiche

- ***NEGlun dst***
  - (NEGate) sostituisce il contenuto della destinazione con il suo opposto (significativo per gli interi)
- ***CMPlun src, dst***
  - (CoMPare) confronta (in base al risultato della differenza dst – src) l'operando dst con l'operando src, aggiornando conseguentemente i flag (significativo sia per i numeri naturali che per gli interi)
- ***MULlun src***
  - (MULtiply) effettua il prodotto tra l'operando destinatario implicito, che funge da moltiplicando, e l'operando sorgente che funge da moltiplicatore (significativo per i naturali)
    - se il moltiplicatore è a 8 bit, allora il moltiplicando è contenuto in AL e il prodotto viene posto in AX
    - se il moltiplicatore è a 16 bit, allora il moltiplicando è contenuto in AX e il prodotto viene posto in DX_AX
    - se il moltiplicatore è a 32 bit, allora il moltiplicando è contenuto in EAX e il prodotto viene posto in EDX_EAX
    - se la metà più significativa del prodotto vale 0, il flag CF viene posto a 0 altrimenti a 1

Istruzioni aritmetiche

- ***IMULlun src***
  - (Integer MULtiply) effettua il prodotto tra l'operando destinatario implicito, che funge da moltiplicando, e l'operando sorgente che funge da moltiplicatore (significativo per interi)
    - se il moltiplicatore è a 8 bit, allora il moltiplicando è contenuto in AL e il prodotto viene posto in AX
    - se il moltiplicatore è a 16 bit, allora il moltiplicando è contenuto in AX e il prodotto viene posto in DX_AX
    - se il moltiplicatore è a 32 bit, allora il moltiplicando è contenuto in EAX e il prodotto viene posto in EDX_EAX
    - se la metà più significativa del prodotto è una estensione di segno della metà meno significativa, il flag OF viene posto a 0 altrimenti a 1
- ***DIVlun src***
  - (DIVide) effettua la divisione tra l'operando destinatario implicito, che funge da dividendo, e quello sorgente che funge da divisore (significativo per i naturali)
    - se il divisore è a 8 bit, allora il dividendo è contenuto in AX, il quoziente viene posto in AL e il resto in AH
    - se il divisore è a 16 bit, allora il dividendo è contenuto in DX_AX, il quoziente viene posto in AX e il resto in DX
    - se il divisore è a 32 bit, allora il dividendo è contenuto in EDX_EAX, il quoziente viene posto in EAX e il resto in EDX
- ***IDIVlun src***
  - (Integer DIVide) come DIVlun, ma opera su interi

Istruzioni logiche

- ***NOTlun dst***
  - (NOT) sostituisce ciascun bit dell'operando destinatario con il suo complemento
- ***ANDlun src, dst***
  - (AND) sostituisce ciascun bit dell'operando dst con il risultato dell'operazione AND tra il bit stesso e il bit corrispondente dell'opernando src
- ***ORlun src, dst***
  - (OR) sostituisce ciascun bit dell'operando dst con il risultato dell'operazione OR tra il bit stesso e il bit corrispondente dell'operando src
- ***XORlun src, dst***
  - (eXclusive OR) sostituisce ciascun bit dell'operando dst con il risultato dell'operazione XOR tra il bit stesso e il bit corrispondente dell'operando src
- ***TESTlun src, dst***
  - (TEST) esegue l'operazione AND tra ciascun bit dell'operando src e il corrispondente bit dell'operando dst, aggiornando i flag

Istruzioni di traslazione/rotazione

- L'operando sorgente è un numero naturale minore di 32, che può essere un operando immediato o il contenuto di CL; se il campo src viene omesso vale implicitamente 1
- ***SHLlun src, dst***
  - (SHift logical Left) l'operando sorgente indica il numero di traslazioni; una traslazione trasporta ciascun bit dell'operando dst nella posizione immediatamente a sinistra, trasferendo in CF il contenuto del bit più significativo e inserendo 0 nel bit meno significativo
- ***SALlun src, dst***
  - (Shift Arithmetical Left) src indica il numero di traslazioni; ogni traslazione trasporta ciascun bit dell'operando dst nella posizione immediatamente a sinistra, trasferendo in CF il bit più significativo e inserendo 0 nel bit meno significativo (il flag OF diviene 1 se il bit più significativo ha cambiato valore)
- ***SHRlun src, dst***
  - (SHift Logical Right) l'operando sorgente indica il numero di traslazioni; una traslazione trasporta ciascun bit dell'operando dst nella posizione immediatamente a destra, trasferendo in CF il contenuto del bit meno significativo e inserendo 0 nel bit più significativo
- ***SARlun src, dst***
  - (SHift Arithmetic Right) l'operando sorgente indica il numero di traslazioni; una traslazione trasporta ciascun bit dell'operando dst nella posizione immediatamente a destr, trasferendo in CF il contenuto del bit meno significativo e replicando il bit più significativo

Istruzioni di traslazione/rotazione

- ***ROLlun src, dst***
  - (ROtate Left) l'operando sorgente indica il numero di traslazioni; una traslazione trasporta ciascun bit dell'operando dst nella posizione immediatamente a sinistra, il bit più significativo in quello meno significativo e in CF il bit più significativo
- ***RORlun src, dst***
  - (ROtate Right) src indica il numero di traslazioni; ogni traslazione trasporta ciascun bit dell'operando dst nella posizione immediatamente a destra, il bit meno significativo in quello più sigificativo e in CF il bit meno significativo
- ***RCLlun src, dst***
  - (ROtate with Carry Left) l'operando sorgente indica il numero di traslazioni; una traslazione trasporta ciascun bit dell'operando dst nella posizione immediatamente a sinistra, il flag CF nel bit meno significativo e in CF il bit più significativo
- ***RCRlun src, dst***
  - (ROtate with Carry Right) l'operando sorgente indica il numero di traslazioni; una traslazione trasporta ciascun bit dell'operando dst nella posizione immediatamente a destra, il flag CF nel bit più significativo e in CF il bit meno significativo

Istruzioni di ingresso/uscita e per i sottoprogrammi

- Istruzioni di ingresso/uscita
- ***INlun porta, accumulatore***
  - (INput) trasferisce una copia del contenuto della porta (o della doppia porta o quadrupla porta) nell accumulatore
- ***OUTlun accumulatore, porta***
  - (OUTput) trasferisce una copia del contenuto dell'accumulatore nella porta (o nella doppia porta o quadrupla porta)
- Istruzioni per i sottoprogrammi
- ***CALL EIP + spiazzamento / CALL destinazione***
  - (CALL) chiama un sottoprogramma; più precisamente: salva nella pila il contenuto di EIP e trasferisce in EIP l'indirizzo di salto
- ***RET n***
  - (RETurn) ritorna da un sottoprogramma; più precisamente: estrae dalla pila una parola lunga e la trasferisce in EIP, quindi incrementa ESP di n (se n vale 0 può essere omesso)
- ***LEAVE***
  - (LEAVE) usata nei linguaggi ad alto livello per ritornare da un sottoprogramma; più precisamente: ricopia in ESP il contenuto di EBP, quindi estrae una parola lunga dalla pila e la trasferisce in EBP

Istruzioni di salto

- ***JMP EIP + spiazzamento* / *JMP destinazione***
  - (JuMP) determina l'indirizzo di salto (EIP + spiazzamento o il contenuto di destinazione) e lo pone in EIP
- ***J*cond *EIP + spiazzamento***
  - (Jump if CONDition) se la condizione COND è soddisfatta effettua il salto, altrimenti non compie nessuna azione
  - JE (Jump if Equal) ZF=1
  - JNE (Jump if Not Equal) ZF=0
  - JA (Jump if Above, per numeri natirali) CF=0 e ZF=0
  - JAE (Jump if Above or Equal, per numeri naturali) CF=0
  - JB (Jump if Below, per numeri naturali) CF=1
  - JBE (Jump if Below or Equal, per numeri naturali) CF=1 o ZF=1
  - JG (Jump if Greater, per numeri interi) ZF=0 e SF=OF
  - JGE (Jump if Greater or Equal, per numeri interi) SF=OF
  - JL (Jump if Less, per numeri interi) SF !=OF
  - JLE (Jump if Less or Equal) ZF=1 e SF!=OF
  - JZ (Jump if Zero) ZF=1
  - JNZ (Jump if Not Zero) ZF=0

Istruzioni di salto

- JC (Jump if Carry) CF=1
- JNC (Jump if Not Carry) CF=0
- JO (Jump if Overflow) OF=1
- JNO (Jump if Not Overflow) OF=0
- JS (Jump if Sign) SF=1
- JNS (Jump if Not Sign) SF=0
- JPE (Jump if Parity Even) PF=1
- JPO (Jump if Parity Odd) PF=0
- JCXZ (Jump if CX is Zero) il contenuto di CX è zero
- JECXZ (Jump if ECX is Zero) il contenuto di ECX è zero

- ***LOOP*cond *EIP + spiazzamento***
  - (LOOP until condition) decrementa il contenuto di ECX senza modificare il contenuto dei flag: se, dopo il decremento, il contenuto di ECX è diverso da zero e se la condizione cond è soddisfatta, allora effettua il salto, altrimenti non compie alcuna azione; lo spiazzamento deve essere compreso tra −128 e +127
  - LOOP nessuna ulteriore condizione
  - LOOPZ/LOOPE (LOOP if Zero/Equal) ZF=1
  - LOOPNZ/LOOPNE (LOOP if Not Zero/Equal) ZF=0

Istruzioni per le interruzioni e il controllo del processore

- Istruzioni per il meccanismo di interruzione
- ***INT n***
  - (INTerrupt) produce una interruzione di tipo n
- ***IRET***
  - (INTerrupt RETurn) ritorna da una interruzione

- Istruzioni per il controllo del processore
- ***NOP***
  - (No OPeration) non compie alcuna azione
- ***HLT***
  - (HaLT) arresta il processore, fino a quando non arriva una richiesta di interruzione o fino a quando il segnale di reset non diviene attivo

Assembler

- Linguaggio in corrispondenza uno a uno con il linguaggio macchina
- Il file contenente un programma in linguaggio Assembler è detto file sorgente e l'estensione del file è comunemente s
- L'assemblatore/compilatore elabora tale file e genera un file intermedio contenente una versione provvisoria del programma in linguaggio macchina (file oggetto, comunemente con estensione o)
- Il file oggetto è elaborato dal collegatore, che inserisce anche eventuali programmi di libreria e genera il file finale (file eseguibile)
-
- Un programma in linguaggio assembler è costituito da una serie di comandi che possono essere suddivisi in istruzioni, pseudo-istruzioni (dichiarazioni di variabili), e direttive (per l'assemblatore)
- Le istruzioni assembler permettono di specificare le istruzioni macchina in forma simbolica, e servono a definire il contenuto delle locazioni di memoria che costituiscono il testo del programma
- Le pseudo-istruzioni consentono di definire il contenuto delle locazioni di memoria che costituiscono i dati del programma (la pila ha dimensione standard e viene definita implicitamente)
- Le direttive forniscono all'assemblatore informazioni utili ai fini della traduzione del programma

Assembler

- Un'istruzione o una pseudo-istruzione occupa una riga di testo ed è composta dai seguenti 4 campi:

  `nome: codice operandi #commento`

  - il primo campo è opzionale e rappresenta l'indirizzo simbolico della prima locazione di memoria corrispondente all'istruzione o alla pseudo istruzione, ed è costituito da un identificatore scelto dal programmatore seguito da ':' (identificatori case sensitive, lettere, cifre, _, .)
  - il campo codice è costituito da una parola chiave del linguaggio e indica l'istruzione o la pseudo-istruzione
  - il campo operandi ha una struttura che dipende dal campo codice
  - l'ultimo campo è opzionale e se presente introduce un commento (fino a fine riga)
- Una riga di testo può contenere solo un nome (seguito da ':'): in tal caso rappresenta l'indirizzo della prima locazione di memoria associata alla istruzione o pseudo-istruzione successiva (una istruzione può avere tanti nomi e tutti corrispondono allo stesso indirizzo)
- Una direttiva ha la forma

  `.codice operandi`

  il campo operandi ha una struttura che dipende da quanto specificato nel campo codice
- Il codice dei vari comandi e i nomi dei registri possono essere espressi usando sia lettere maiuscole che minuscole

File sorgente

- Nel caso più semplice un programma Assembler è contenuto in un unico file
- Il programma comprende una sezione per i dati (che inizia con la direttiva *.data*) e una sezione per il testo (che inizia con la direttiva *.text*); le due seziono possono essere costituite da più parti
- La sezione per il testo contiene generalmente uno o più sottoprogrammi e il (sotto)programma principale
- Programmi per il compilatore
  - il sottoprogramma principale ha nome *_main* e deve essere dichiarato globale
  - il sottoprogramma principale deve lasciare in EAX una informazione sull'esito dell'esecuzione del programma, che vale 0 se non si sono avuti errori
  - la traduzione ed il collegamento vengono ottenute con un unico comando:
    - **`gcc nomefile.s -o nomefile.exe`**

```
#*************************
.data
    ...
    dati
    ...
#*************************
   *
.text
    ...
#-------------------------
.global _main
#-------------------------
_main: ...
    istruzioni
    ...
    xorl %eax, %eax
    ret
#*************************
```

Rappresentazione di costanti

- Nei comandi Assembler possono essere utilizzate costanti, sia per specificare operandi nelle istruzioni che per definire valori iniziali nelle dichiarazioni di variabili
- Le costanti possono essere suddivise in numeri, letterali carattere e letterali stringa
- I numeri possono essere naturali, interi e reali
  - un numero naturale può essere espresso in diverse basi
    - base dieci: inizia con una cifra diversa da 0, per es. 245
    - base otto: inizia con la cifra 0, per es. 0322
    - base sedici: inizia con la sequenza 0x (oppure 0X), per es. 0xA232
    - base due: inizia con la sequenza 0b (oppure 0B), per es. 0b11001
  - espresso in una delle basi facendo precedere il primo carattere da '-' o '+' (opzionale)
- Letterali carattere
  - carattere racchiuso tra due apici oppure preceduto da un apice, per es 'a' oppure 'A
  - viene convertito dall'assemblatore nella codifica ASCII (estesa a 8 bit)
  - caratteri speciali indicati con sequenza di escape
    - \n avanzamento linea (ASCII 0x0A)
    - \r ritorno carrello (ASCII 0x0D)
    - \t tabulazione (ASCII 0x09)

\\ barra invertita (ASCII 0x5C)
\' apice (ASCII 0x27)

Rappresentazione di costanti e definizione di var.

- Letterale stringa
  - una sequenza di uno o più caratteri racchiusi tra virgolette, per esempio “errore” e “questa e’ una stringa”
  - all’interno di una stringa possono essere usate le sequenze di escape con l’aggiunta di
    - \” indica virgolette (ASCII 0x22)
  - vengono convertiti in sequenze di caratteri ASCII (estese a 8 bit)
- Alle costanti può essere assegnato un nome simbolico tramite la direttiva .set, tale nome può poi essere usato nel programma in sostituzione della costante

```
.set num_passi, 246
```

- Il corpo della sezione dati è costituito, oltre che da eventuali direttive, da dichiarazioni di variabili
  - ogni variabile è caratterizzata da un tipo
  - scalare/vettoriale
  - vengono riservate una o più locazioni di memoria contigue
  - le variabili vengono dichiarate con una pseudo-istruzione del tipo

```
nome: .codice espressione
```

il nome può essere usato nel campo operandi di un’altra istruzione, l’espressione stabilisce il valore iniziale di una componente (tante espressioni quante sono le componenti)

Definizione di variabili

- Pseudo-istruzioni per definire caratteri, numeri naturali, numeri interi e indirizzi:

```
.byte     #numero naturale o intero a 8 bit, o carattere
.word     #numero naturale o intero a 16 bit
.long     #numero naturale o intero a 32 bit, o indirizzo
.quad     #numero naturale o intero a 64 bit
```

- per definire letterali stringa:

```
.ascii    #letterale stringa, memorizzata un byte per carattere
.asciz    #letterale stringa, come sopra più carattere nullo in coda
```

- Per definire una variabile vettoriale con *num* componenti, tutte di lunghezza *lun* byte (al più *lun* vale 8) e di valore iniziale *val*:

```
nome: .fill num, lun, val
```

  il campo *val* è opzionale e se assente (deve essere assente anche la virgola) vale implicitamente 0

- Per riservare un certo numero totale di byte (*num_byte*) come nel caso di strutture o vettori con componenti con dimensione maggiore di 8:

```
nome: .space num_byte, val_byte
```

  val_byte è opzionale e se non specificato vale 0

Dichiarazioni di variabili

- Esempi

```
alfab:  .byte       0x2a #variabile di tipo byte, 1 componente, valore iniziale: 0x2a
gamb:   .byte       'm', 0x0a #variabile di tipo byte, 2 componenti, val. iniziali: 'm' e 0x0a
etab:   .fill   2, 1, 0x3b #variabile di tipo byte, 2 componenti, val. iniziale: 0x3b
iotab:  .fill   50, 1 #variabile di tipo byte, 50 componenti, val. iniziale: 0
alfaw:  .word       -0x6b33 #variabile di tipo word, 1 componente, val. iniziale: -0x6b33
betaw:  .word       0x33, 22 #variabile di tipo word, 2 componenti, val iniziale: 0x33 e 22
gamw:   .fill 50, 2  #variabile di tipo word a 50 componenti, val iniziale: 0
alfal:  .long 0x323233f3 #variabile di tipo long, 1 componente, val iniziale: 0x323233f3
betal:  .long alfal #variabile di tipo long, 1 componente, val.iniziale: inidirizzo di alfal
gaml:   .fill 4, 4    #variabile di tipo long, 4 componenti, val. iniziale: 0
alfaq:  .quad       10 #variabile di tipo quad, 1 componente, val. iniziale: 10
```

Sezione testo

- Il corpo della sezione testo è costituito, oltre che dalle eventuali direttive, dalle istruzioni che costituiscono il programma.
- Poiché alle istruzioni e alle variabili può essere associato un nome simbolico, i campi src e dst possono contenere nomi di variabili o istruzioni
- Convenzioni:
  - i nomi dei registri devono essere preceduti dal carattere '%'
  - gli operandi immediati vengono preceduti dal carattere '$'
- Gli indirizzi in memoria vengono espressi secondo la seguente forma:

  spiazzamento(reg_base, reg_indice, scala)

  una o più componenti possono mancare (quando non è presente la componente scala viene assunta implicitamente uguale ad uno)
- Quando si riferiscono variabili scalari è spesso sufficiente l'indirizzamento diretto, mentre nel caso in cui si riferiscono variabili vettoriali spesso si usa l'indirizzamento indiretto o quello modificato con registro indice

Riferimento a variabili

```
.data
alfab: .byte 0
vettaw: .fill 50, 2
vettbw: .fill 50, 2
```

- La variabile scalare *alfab* può essere riferita così

```
movb alfab, %al  # indirizzamento diretto
```

- La componente i-ma della variabile vettoriale *vettaw* può essere così riferita

```
movl $vettaw, %ebx   #indirizzamento indiretto con reg. ebx puntatore
...
porzione di programma in cui il contenuto di ebx viene incrementato di 2*i
...
addw %ax, (%ebx)
```

- Le componenti dei vettori *vettaw* e *vettbw* possono essere così riferite

```
movl $0, %ebx    #indirizzamento con registro indice
...
porzione di programma in cui il contenuto di ebx viene incrementato di 2*i
...
movw vettaw(%ebx), %ax
addw %ax, vettbw(%ebx)
```

Riferimento ad altre istruzioni

- Il nome simbolico assegnato ad una istruzione può essere usato per riferire questa nelle istruzioni di controllo: l'assemblatore traduce l'istruzione di salto o di chiamata di sottoprogramma nel formato che prevede l'indirizzamento relativo, calcolando il valore numerico dello spiazzamento

```
#**********************
.data
#**********************
.text
sub: pushl %eax
     ret
#----------------------
.global _main
#----------------------
_main: ...
ciclo: ...
     jnz ciclo
     jmp avanti
avanti:
     call sub
     xorl %eax, %eax
     ret
#**********************
```

Riferimento alle porte di I/O e inizializzazioni

```
.set porta, indirizzo_di_io
...
#se porta minore di 256
inb $porta, %al
..
movw $porta, %dx
inb %dx, %al
```

- Discorso analogo per l'uscita
- L'inizializzazione del registro ESP, che determina il punto di partenza della pila, viene eseguita dal caricatore, prima di mandare in esecuzione il programma
- Per mandare in esecuzione il programma il caricatore effettua l'inizializzazione del registro EIP, con l'indirizzo di partenza del programma
- Inclusione: la direttiva *.include* permette di copiare il contenuto di un file dentro un altro file; supponiamo di aver predisposto un file *servizio.s* e di scrivere in un punto del programma (in un altro file)

  `.include "servizio.s"`

  questo equivale a scrivere in tale punto l'intero contenuto del file *servizio.s*

  - se il file non si trova nella cartella corrente è possibile specificare un percorso

servizio.s

- Il file *servizio.s* contiene due sottoprogrammi di utilità
  - sottoprogramma *tastiera*: attende che la tastiera generi un carattere e, quando ciò accade, immette in AL la codifica ASCII del carattere stesso
  - sottoprogramma *video*: invia sul video il contenuto di AL; se il registro AL contiene la codifica ASCII di un carattere scrivibile, il carattere stesso appare sul video nella posizione attuale del cursore e il cursore avanza di una posizione
- Programma che invia un messaggio

```
#*********************
.data
mess:   .ascii “programma in esecuzione\n\r”
        .ascii “per terminare premi un tasto”
f_mess:
#*********************
.text
.include “servizio.s”
#-----------------------
.global _main
#-----------------------
_main:  movl $mess, %ebx
        movw $(f_mess – mess), %cx
ripeti: movb (%ebx), %al
        call video
        addl $1, %ebx
        subw $1, %cx
        jnz ripeti
        call tastiera
        xorl   %eax, %eax
        ret
#*********************
```

Uso dei sottoprogrammi

- Il programma precedente può essere modificato dotandolo di un sottoprogramma *invia* che stampa un messaggio a video; parametri di *invia*:
  - indirizzo del messaggio da stampare (in EBX)
  - numero di caratteri di cui è formato il messaggio (in CX)

```
#*********************
.data
mess:   .ascii “programma in esecuzione\n\r”
        .ascii “per terminare premi un tasto”
f_mess:
#*********************
.text
.include “servizio.s”
invia:  pushl   %ebx
        pushl   %ecx
ripeti: movb    (%ebx), %al
        call    video
        addl    $1, %ebx
        subw    $1, %cx
        jnz     ripeti
        popl    %ecx
        popl    %ebx
        ret
#-----------------------
```

```
.global _main
#-----------------------
_main:  movl    $mess, %ebx
        movw    $(f_mess – mess), %cx
        call    invia
        call    tastiera
        xorl    %eax, %eax
        ret
#*********************
```

Programma che esamina la codifica di un carattere

```
#-----------------------------
.data
kappa: .fill  8, 1
#-----------------------------
.text
.include “servizio.s”
esamina:  #parametri in AL codice del carattere da esaminare
          #in EBX indirizzo della variabile risultato
          pushl  %eax
          pushl  %esi
          movl   $0, %esi
ciclo:    testb  $0x80, %al
          je     zero
          movb   $’1, (%ebx, %esi)
          jmp    avanti
zero:     movb   $’0, (%ebx, %esi)
avanti:   shlb   $1, %al
          incl   %esi
          cmpl   $8, %esi
          jb     ciclo
          popl   %esi
          popl   %eax
          ret
```

Programma che esamina la codifica di un carattere

```
#-----------------------------
.global _main
#-----------------------------
_main:
ancora:   call   tastiera
          cmpb   $0x0d, %al
          je     fine
          movl   $kappa, %ebx
          call   esamina
          call   video
          movb   $0x20, %al
          call   video
          movl   $0, %ecx
ripeti:   movb   kappa(%ecx), %al
          call   video
          incl   %ecx
          cmpl   $8, %ecx
          jb     ripeti
          movb   $0x0d, %al
          call   video
          movb   $0x0a, %al
          call   video
          jmp    ancora
fine:     xorl   %eax, %eax
          ret
```