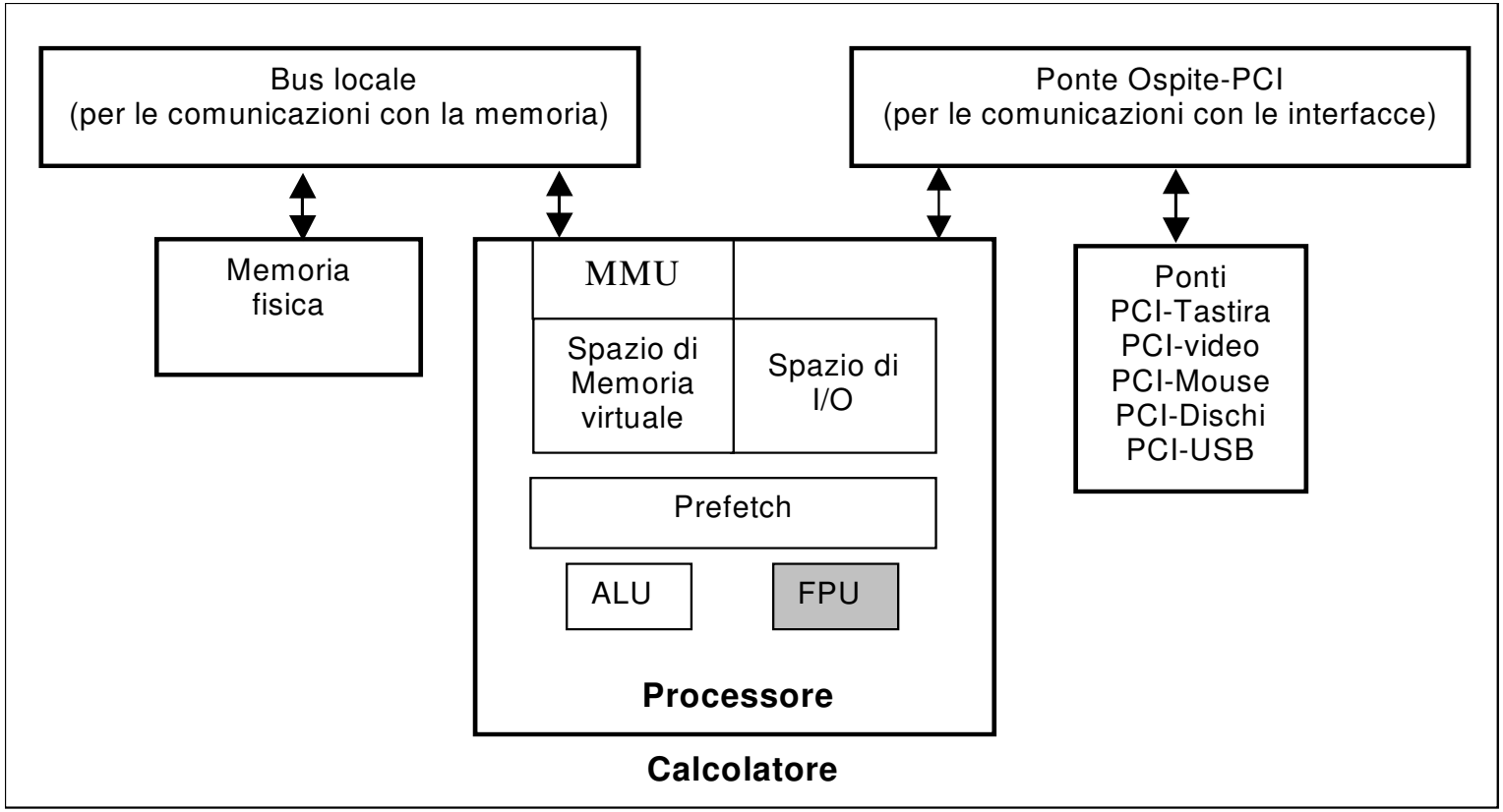
ELABORATORE x86-64
E PRINCIPI DI PROGRAMMAZIONE

Caratteristiche del processore x86-64 (1)

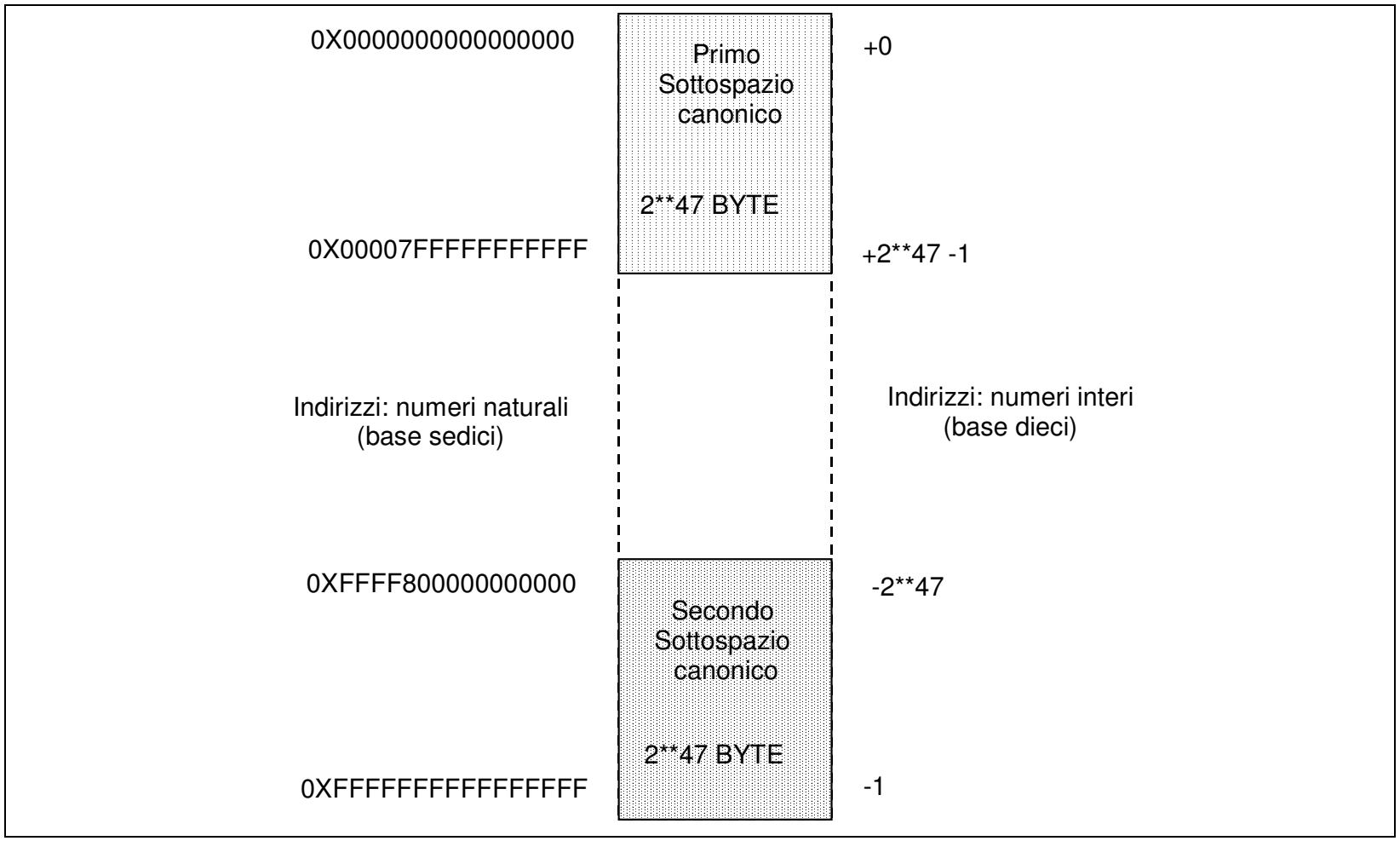
- **Processore Intel/AMD x86-64:**
  - **comprende 2 unità di elaborazione visibili al programmatore:**
    - **la ALU (*Aritmetic and Logic Unit*), che esegue le istruzioni generali e quelle sui numeri naturali e interi;**
    - **la FPU (*Floating Point Unit*), che esegue le istruzioni sui numeri reali.**
- **Operandi manipolati dalle istruzioni della ALU:**
  - **lunghi 8 bit (byte), 16 bit (parola), 32 bit (parola intera o lunga), 64 bit (parola quadrupla o doppia parola lunga).**
- **Operandi manipolati dalle istruzioni della FPU:**
  - **lunghi 80 bit.**
- **Processore schematizzato x86-64 (Processore con la P maiuscola):**
  - **costituito solo la ALU (esegue solo le elaborazioni sui numeri naturali e interi).**

Caratteristiche di un calcolatore x86-64



Calcolatore schematizzato x86-64

Spazio di memoria (virtuale)



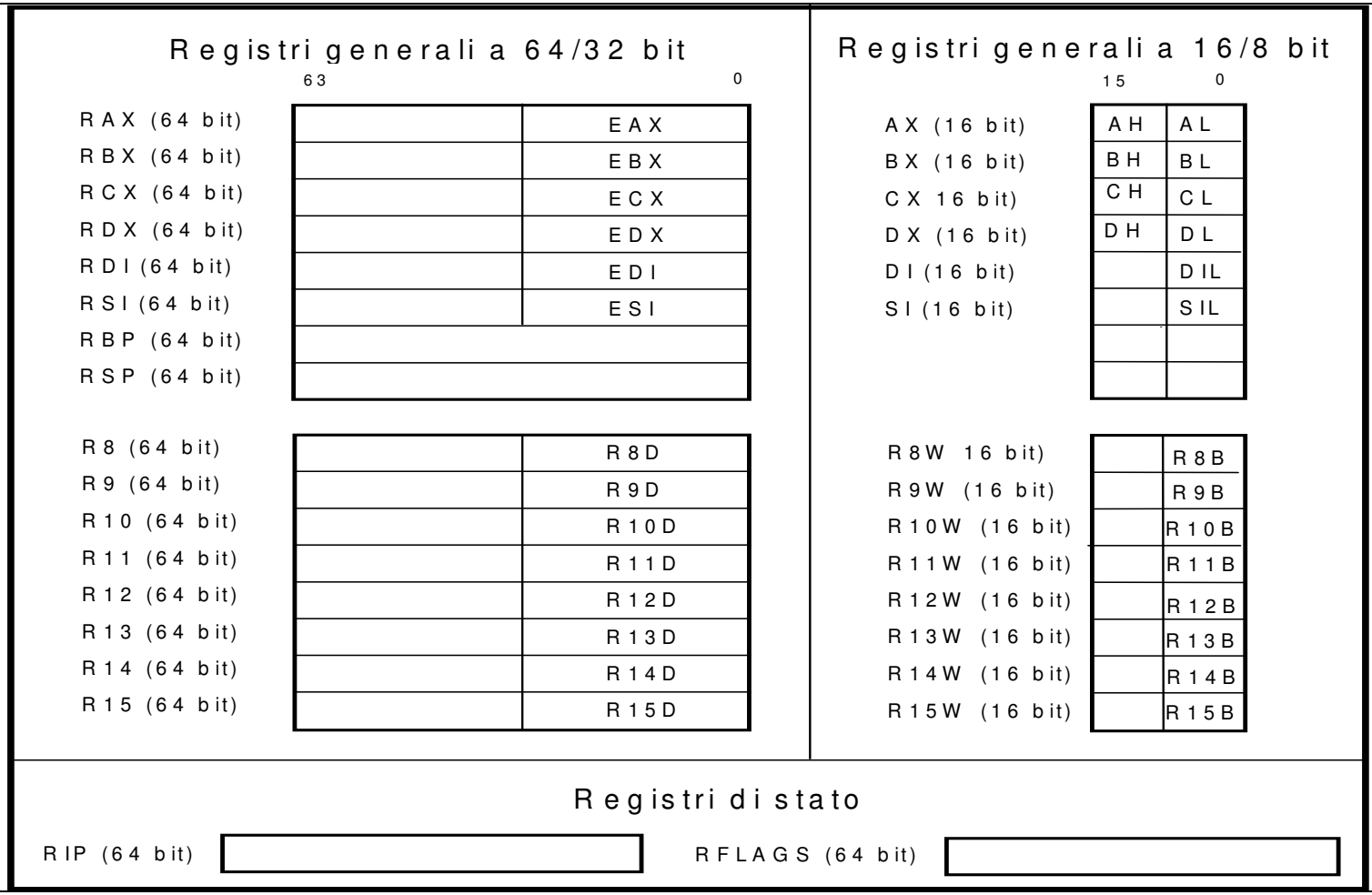
Spazi di indirizzamento

- **Funzionamento del Processore:**
  - **alla partenza, in modalità x86-16 (come il processore 8086);**
  - **dopo un'apposita istruzione, in modalità x86-64, con memoria virtuale.**
- **Spazio di memoria (virtuale, quello che usa il programmatore) in modalità x86-64:**
  - **indirizzo lineare e costituito da 64 bit, ciascuno dei quali individua una cella di un byte;**
    - **gli indirizzi sono numeri naturali, che possono anche essere considerati codifiche in complemento a 2 di numeri interi;**
  - **più celle possono essere associate consecutivamente in locazioni di 2, 4, 8 byte.**
  - **2**48 dei 2**64 possibili indirizzi sono detti *canonici* (hanno i 16 bit più significativi, tutti di valore 0 o tutti di valore 1, uguali al valore del bit alla loro destra):**
    - **individuano 2 sottospazi di 2**47 bit ciascuno, all'inizio e alla fine di tutto lo spazio di indirizzamento, con indirizzi:**

      0x0000000000000000-0x00007FFFFFFFFFFF

      0xFFFF800000000000-0xFFFFFFFFFFFFFFFF
  - **gli indirizzi utilizzati dal programmatore (per istruzioni e dati) devono essere canonici.**
- **Gli indirizzi vengono tradotti da virtuali a fisici (quelli delle locazioni di memoria) dalla MMU.**
- **Spazio di I/O (reale) in modalità x86-64**
  - **lineare, e costituito da 2**16 porte di un byte, il cui indirizzo va da 0x0000 a 0xFFFF;**
  - **le porte di un byte possono essere associate consecutivamente 2 a 2 (porte di 16 bit), 4 a 4 (porte di 32 bit), 8 a 8 (porte a 64 bit).**

Registri della ALU (1)

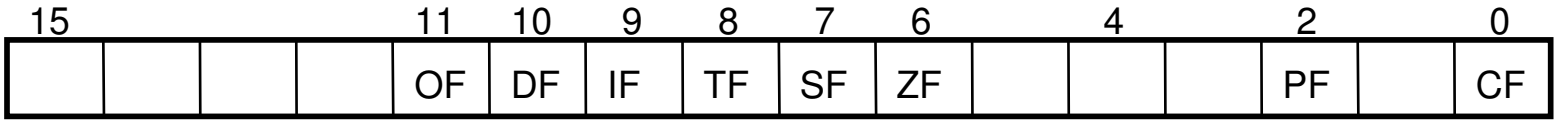


Registri della ALU(2)

- **Registri generali:**
  - **registri generali a 64 bit:**
    - **sono 16;**
    - **possono venir indifferentemente utilizzati per memorizzare operandi e per contenere indirizzi virtuali di memoria;**
    - **alcuni di essi svolgono specifiche funzioni:**
      - **registri RSP, RBP: servono a indirizzare la pila (RSP funge da puntatore alla cima della pila, ed RBP funge da registro base per una zona della pila);**
      - **registri RAX, RDX: spesso RAX funge da accumulatore, ed RDX da estensione dell'accumulatore;**
  - **registri generali a 32 bit e a 16 bit: parti meno significative dei corrispondenti registri a 64 bit (tranne RSP ed RBP) e a 32 bit;**
  - **registri generali a 8 bit: parti meno significative dei corrispondenti registri a 16 bit, e parti più significative dei registri AX, BX, CX e DX.**
- **Registri di stato:**
  - **Registro RIP: contatore di istruzioni (64 bit);**
  - **Registro RFLAGS: registro delle condizioni (64 bit).**

Registri della ALU (3)

- **Registro di stato RIP (o registro contatore):**
  - **contiene l'indirizzo virtuale della prossima istruzione.**
- **Registro di stato RFLAGS (o registro delle condizioni):**
  - **i due byte meno significativi contengono:**
    - **i flag aritmetici PF (Parity), ZF (zero), CF (Carry), SF (Sign), OF (Overflow);**
    - **i flag IF e TF relativi al meccanismo di interruzione;**
    - **il flag DF utilizzato dalle istruzioni sulle stringhe.**



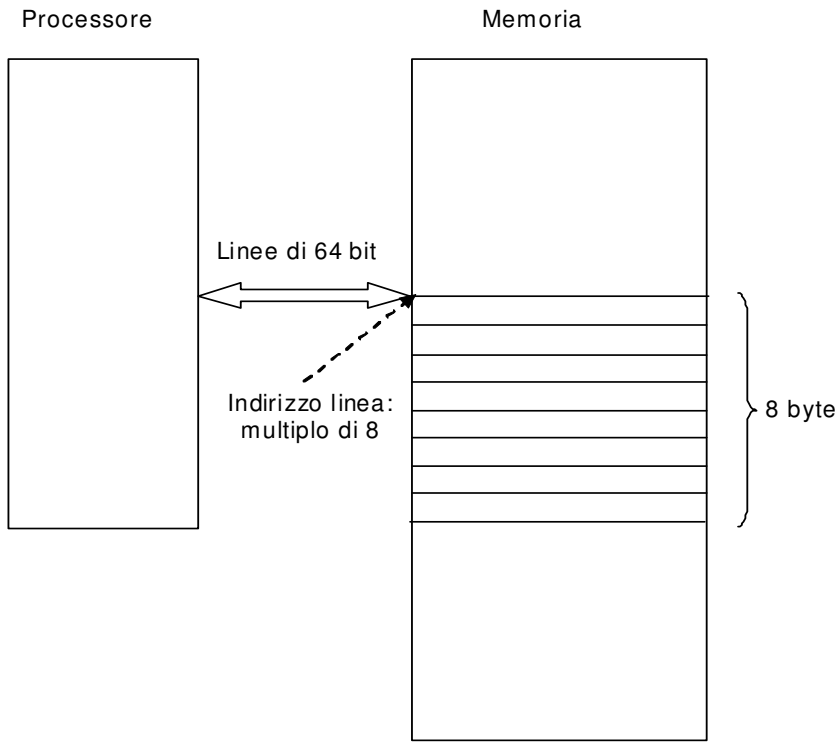
Codifica delle istruzioni

Codice operativo OC (1 o 2 byte) Operazione e lunghezza operandi)	Eventuale secondo byte Registri espliciti coinvolti e modi indirizzamento	Spiazzamento DISP 0/4 byte	Operando immediato IMM 0/8 byte

- **Codice operativo (OC):**
  - **il primo byte specifica il tipo di operazione e la lunghezza degli operandi (B (8 bit), W (16 bit), L (32 bit), Q (64 bit);**
  - **l'eventuale secondo byte contiene informazioni aggiuntive.**
- **Spiazzamento (DISPlacement):**
  - **assente;**
  - **numero intero di 1, 2 oppure 4 byte.**
- **Operando immediato (IMMediate):**
  - **assente;**
  - **numero intero di 1, 2 oppure 4 byte;**
  - **solo per l'istruzione MOVABSQ, numero intero di 8 bye.**

Organizzazione della memoria

- **Organizzazione hardware della memoria (virtuale e fisica):**
  - **a linee, di 8 byte ciascuna;**
  - **si possono leggere o scrivere in un solo ciclo di memoria uno o più byte della stessa linea;**
  - **un ciclo di memoria o tempo di accesso alla memoria è fatto da un numero di cicli del processore;**
  - **esso riguarda l'intera linea (è lo stesso qualunque sia il numero di byte della linea letto o scritto).**



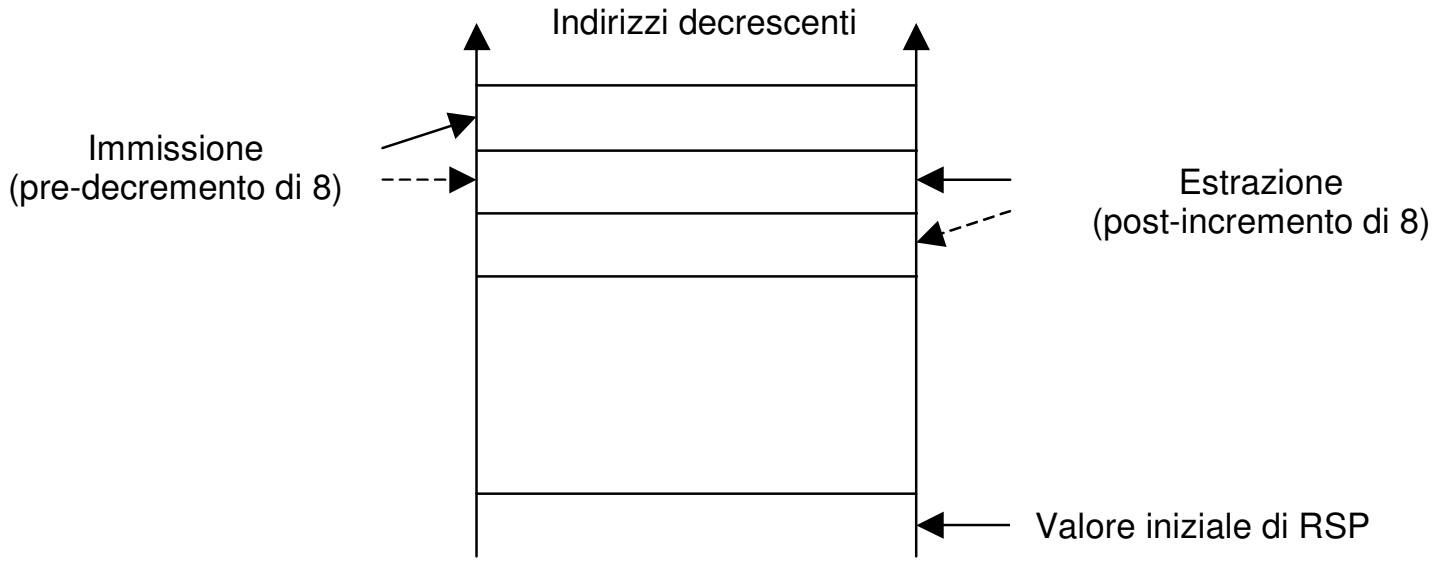
Indirizzi di istruzioni e di operandi

- **Istruzioni:**
  - **l' indirizzo di un'istruzione è quello del primo byte del Codice Operativo.**
    - **le istruzioni non possono venir singolarmente allineate (sono memorizzate in sequenza);**
  - **vengono prelevate a linee, e poi separate all'interno del processore, e viene effettuata la gestione dei salti (vedi "Struttura interna del processore").**
- **Operandi:**
  - **lunghi 1, 2, 4 o 8 byte;**
  - **i byte di un operando sono memorizzati in ordine inverso (*little-endian order* dei byte, non dei bit di un byte) (per primo, il byte meno significativo);**
  - **l'indirizzo dell'intero operando è quello del byte meno significativo dell'operando stesso.**
- **Campi DISP e IMM di un'istruzione:**
  - **anche i byte di questi campi sono memorizzati in ordine inverso.**
- **Allineamento degli operandi:**
  - **un operando può essere memorizzato a partire da qualunque indirizzo di memoria;**
  - **viene in genere allineato a indirizzi multipli della sua lunghezza, in modo che sia contenuto in un sola linea, così da minimizzare il tempo richiesto per l' accesso all'operando stesso.**

La Pila (1)

- **Pila (LIFO: *Last In First Out*):**
  - **organizzata a locazioni di 64 bit, utilizzando come puntatore il registro RSP;**
  - ***immissione* (PUSHQ (Q può essere omesso), CALL, …): decremento di 8 di RSP, quindi scrittura nella parola quadrupla indirizzata da RSP;**
  - ***estrazione* (POPQ (Q può essere omesso), RET, …): lettura della parola quadrupla indirizzata da RSP, quindi incremento di 8 di RSP.**
  - ***inizializzazione*: caricamento in RSP dell'indirizzo della locazione successiva a quella iniziale della pila.**
- **Indirizzamento di operandi (o di parte di essi) in pila (di dimensioni 1, 2, 4 o 8 byte):**
  - **registro base RBP opportunamente inizializzato;**
  - **tipicamente, utilizzo dell'istruzione MOV.**

Pila (2)


Indirizzi decrescenti
Immissione
(pre-decremento di 8)
Estrazione
(post-incremento di 8)
Valore iniziale di RSP

Classi di istruzioni

- **Operative:**
  - **operandi sorgente e destinatario (si trovano nel luogo sorgente e destinazione, rispettivamente);**
  - **rimpiazzano la destinazione con il risultato;**
  - **il luogo sorgente e destinazione possono essere impliciti o mancare.**
- **Di controllo:**
  - **stabiliscono l'indirizzo della prossima istruzione (eventualmente in modo condizionato da bit di RFLAGS).**
- **Di alt.**

Modi di Indirizzamento per le istruzioni operative

- **Indirizzamento di registro: operando in un registro.**
- **Indirizzamento immediato: operando nell'istruzione (campo IMM).**
- **Indirizzamento di memoria con espressioni canoniche (una o due componenti dell'indirizzo possono mancare):**

  INDIRIZZO = | BASE + INDICE*SCALA + DISP | modulo 264**

  - ***BASE* è il contenuto di uno dei registri generali di 64 bit;**
  - ***INDICE* è il contenuto di uno dei registri generali di 64 bit;**
  - ***SCALA* è un fattore che può valere 1, 2, 4 oppure 8;**
  - ***DISP* è un numero intero, di 8, 16 o 32 bit, esteso con segno a 64 bit.**
- **Indirizzamento di memoria con espressioni relative (rispetto a RIP):**

  INDIRIZZO = | RIP + DISP | modulo 264**

  - ***RIP*: contiene l'indirizzo dell'istruzione successiva a quella attualmente in esecuzione;**
  - ***DISP* è un numero intero, di 8 bit, di 16 bit o di 32 bit, esteso con segno a 64 bit.**
- **Nota:**
  - **l'operazione modulo 2**64 di un naturale (compresa la rappresentazione in complemento a 2 di un intero) produce come risultato un naturale (compresa la rappresentazione in complemento a 2 di un intero) espresso con 64 bit.**

Modi di indirizzamento per le istruzioni di controllo e di I/O

- **Indirizzamento per le istruzioni di controllo:**
  - **relativo:**

    INDIRIZZO = | RIP + DISP | modulo 264**
    - ***RIP*: contiene l'indirizzo dell'istruzione successiva a quella attualmente in esecuzione;**
    - ***DISP* è un numero intero di 8, 16 o 32 bit, esteso con segno a 64 bit.**
  - **indiretto (possibile solo per le istruzioni di salto incondizionato):**

    INDIRIZZO = CONTENUTO DI UN REGISTRO (di 64 bit);

    INDIRIZZO = CONTENUTO DI UNA LOCAZIONE DI MEMORIA (di 64 bit) individuata con un indirizzo canonico o relativo.
- **Indirizzamento di una porta di I/O:**
  - **l'indirizzo può essere specificato nell'istruzione;**
  - **l'indirizzo può essere contenuto nel registro DX;**
  - **prima forma:**
    - **possibile solo per indirizzi minori di 256.**

Zone di memoria indirizzabili

- **Spiazzamento:**
  - **è un intero, rappresentato al più su 32 bit.**
- **Indirizzamento con espressioni canoniche:**
  - **a partire dal contenuto di un registro base RBASE (da 0 se il registro base è assente), con il solo spiazzamento si può indirizzare una zona di memoria che va da:**

    | RBASE – 231| modulo 2**64 a | RBASE + 2**31 -1 | modulo 2**64 ;**
    - **il registro indice viene tipicamente utilizzato per gli array, i cui elementi possono essere lunghi 1, 2, 4 oppure 8 byte.**
- **Indirizzamento con espressioni relative:**
  - **a partire dal contenuto del registro RIP, consente di indirizzare una zona di memoria che va da:**

    | RIP - 231 | modulo 2**64 a | RIP + 2**31-1| modulo 2**64.**
- **Indirizzamento con espressioni indirette (salto incondizionato)**
  - **in un registro o in una locazione di memoria di 64 bit può essere contenuto qualunque indirizzo.**

Caricamento di un indirizzo di memoria

- **Caricamento di un indirizzo di memoria in un registro:**
  - **si può effettuare utilizzando l'istruzione MOVABSQ/MOVQ con un operando immediato, che viene calcolato a *tempo di traduzione*;**
    - **utilizzando MOVABSQ, l'operando immediato è di 64 bit;**
    - **nel registro può essere caricato qualunque indirizzo di memoria.**
    - **utilizzando MOVQ, l'operando immediato è di 32 bit, e viene esteso con segno a 64 bit;**
      - **nel registro può essere caricato un indirizzo di memoria compreso fra 2**31-1 e -2**31.**
  - **si può effettuare utilizzando l'istruzione LEAQ:**
    - **espressione relativa: trasferisce nel registro specificato la quantità DISP+RIP, che viene calcolata a *tempo di esecuzione*;**
    - **espressione canonica: trasferisce nel registro specificato la quantità BASE + INDICE*SCALA + DISP, che viene calcolata a *tempo di esecuzione*.**

UNIX e ambiente di programmazione GCC

- **Ambiente di programmazione GCC (GNU Compiler Collection), che rientra nel software libero:**
  - **collezione di software di sviluppo del sistema GNU;**
  - **comprende l'Assemblatore, il Compilatore C, il Compilatore C++, il Collegatore, il Caricatore;**
  - **può essere utilizzato in un calcolatore con x86-64 con sistema operativo UNIX, che rientra anch'esso nel software libero.**
    - **software libero: significa che può essere liberamente utilizzato, alle condizioni specificate nella Licenza Generale GNU (che ne esclude la vendita e che può anche essere anche ritirata in qualunque momento).**
- **Programma Assembler:**
  - **costituito, oltre che da commenti, da istruzioni, pseudo-istruzioni (per definire operandi), direttive (comandi per l'Assemblatore);**
  - **può utilizzare i servizi UNIX (*primitive*), in particolare quelli per leggere da tastiera e per scrivere su video;**
  - **può utilizzare letterali, che si scrivono come in C++;**
  - **può essere scritto su uno o più file, come in C++.**

Assembler GCC (1)

- **Forma di un commento:**
  - **va dal carattere # alla fine della linea corrente**
- **Forma di un'istruzione Assembler:**

  identificatore: codice-operativo sorgente, destinatario
  - **identificatore, sorgente, destinatario: possono anche mancare;**
  - **identificatore:**
    - **rappresenta un indirizzo simbolico;**
    - **lettere minuscole o lettere maiuscole sono differenti.**
  - **codice operativo:**
    - **specifica l'operazione e la lunghezza degli operandi (b: byte; w: parola; l: parola lunga; q: parola quadrupla);**
    - **lettere minuscole o lettere maiuscole sono equivalenti.**
  - **operandi (sorgente e destinatario):**
    - **%: specifica il nome di un registro;**
    - **$: specifica un operando immediato;**
    - ***: specifica un'indirezione (istruzioni *jmp* e *call*);**
    - **nel nome di un registro, lettere minuscole o lettere maiuscole sono equivalenti.**

Assembler GCC (2)

- **Forma di una pseudo-istruzione Assembler:**
  identificatore: codice operandi
  - **l'identificatore può anche mancare;**
    - **rappresenta un indirizzo simbolico;**
    - **lettere minuscole o lettere maiuscole sono differenti;**
  - **il codice inizia con . (punto) ;**
    - **lettere minuscole o lettere maiuscole sono equivalenti.**
- **Esempi di pseudo-istruzioni più utilizzate :**

```
.byte     valore          # numero naturale o intero di 8 bit, o letterale carattere
.word     valore          # numero naturale o intero di 16 bit
.long     valore          # numero naturale o intero di 32 bit
.quad     valore          # numero naturale o intero di 64 bit
.space    numero-byte
.fill     numero-componenti, numero-byte-per-componente
.ascii    letterale stringa
```

- **Osservazione:**
  - **le prime 4 pseudo-istruzioni consentono di memorizzare o un numero naturale o la codifica di un numero intero in complemento a 2, rispettando gli intervalli di rappresentazione;**
  - **ricordare che nella codifica dei numeri interi in complemento a 2, le operazioni di somma e sottrazione sui numeri interi si effettuano con le stesse operazioni sulle loro codifiche.**

Assembler GCC (3)

- **Forma di una direttiva Assembler:**
  codice ***operandi***
  - **il codice inizia con . (punto);**
    - **lettere minuscole o lettere maiuscole sono equivalenti.**
- **Esempi di direttive più utilizzate:**

```
.text                          # sezione testo
.data                          # sezione dati
.include    <id_file>          # identificatore del file di libreria da includere
.include    "id_file"          # identificatore del file utente da includere,
                               # con eventuale percorso
.global     identificatore, ...  # identificatore globale (vedi slide successive)
.globl      identificatore, ...  # come sopra
.extern     identificatore, ...  # identificatore esterno (vedi slide successive)
.align      n                  # allineamento a un indirizzo multiplo di n, potenza di 2
.balign     n                  # come sopra
```

Assembler GCC (4)

- **Espressione Assembler:**
  - **contiene valori numerici e identificatori (simbolici);**
    - **espressione-indirizzo: contiene almeno un identificatore simbolico.**
- **Fase di traduzione del programma (traduzione vera e propria e collegamento):**
  - **agli identificatori simbolici viene fatto corrispondere un valore numerico (indirizzo di memoria);**
  - **un'espressione produce un risultato numerico.**

Assembler GCC (5)

- **Indirizzamenti in Assembler:**
  - **Operando in un registro: nome del registro preceduto da *%*;**
  - **Operando nell'istruzione (immediato): preceduto dal simbolo $;**
- **Operando in memoria: indirizzo di memoria canonico (solo nelle istruzioni operative):**

  ***spiazzamento* (*base, indice, scala*)**
  - ***spiazzamento:* espressione che viene calcolata in fase di traduzione, il cui risultato (di 32 bit) viene memorizzato nel campo DISP dell'istruzione macchina;**
  - ***base* e *indice:* registri generali a 64 bit;**
  - ***scala*: numero naturale che può valere 1, 2, 4, 8, e se viene omesso vale 1;**
  - ***intero indirizzo*: calcolato a tempo di esecuzione.**
- **Esempi:**

mov*l* spiazzamento, %*l*reg	**ind. diretto**
mov*l* (%rbase), % *l*reg	**ind. con registro puntatore**
mov*l* spiazzamento(%rbase), % *l*reg	**ind. modificato con registro base**
mov*l* spiazzamento(, %rind, 4), % *l*reg	**ind. modificato con registro indice**
mov*l* (%rbase, %rind, 4), % *l*reg	**ind. bimodificato senza displacement**
mov*l* spiazzamento(%rbase, %rind, 4), % *l*reg	**ind. bimodificato con displacement**

- **La presenza di registro indice è significativa solo per gli array-**

Assembler GCC (6)

- **Operando in memoria: indirizzo di memoria relativo nelle istruzioni operative:**
  - **in Assembler, indicazione dell'indirizzo dell'operando e del simbolo RIP:**

    mov*l* ind(%rip), %reg

    - ***ind* è un'espressione che individua (nel caso più semplice coincide con) l'indirizzo simbolico dell'operando;**
    - **l'indicazione RIP è obbligatoria, altrimenti potrebbe essere un'espressione canonica con il solo DISP;**
      - **fase di traduzione: viene calcolato il valore del campo DISP dell'istruzione macchina (che non è *ind*);**
      - **programmatore Assembler: non deve specificare DISP, ma solo *ind (indirizzo simbolico dell'operando);***
      - **fase di esecuzione: l' indirizzo dell'operando viene calcolato come DISP + RIP*;***
    - **si può indirizzare un operando solo in una zona limitata a cavallo di RIP.**
- **Istruzione LEA:**

  leaq ind(%rip), %*q*reg
  - **stesse caratteristiche delle istruzioni operative con indirizzamento di memoria relativo.**

Assembler GCC (7)

- **Operando in memoria: indirizzo di memoria relativo nelle istruzioni di controllo:**
  - **indicazione Assembler dell'indirizzo di salto (non è prevista l'indirizzo di salto utilizzando un'espressione canonica):**

    jmp ind (l'indicazione di RIP è implicita)
    - ***ind* è un'espressione che individua (nel caso più semplice coincide con) l'identificatore dell'istruzione a cui avviene il salto;**
      - **in fase di traduzione, viene calcolato il valore del campo DISP dell'istruzione macchina;**
      - **il programmatore Assembler non deve specificare DISP, ma solo *ind (indirizzo simbolico di salto)*;**
      - **in fase di esecuzione, il salto avviene all'indirizzp DISP+RIP*;***
    - **si può saltare solo in una zona limitata a cavallo di RIP.**

Proprietà PIC

- **Programmi Assembler sviluppati nel seguito:**
  - **utilizzazione della proprietà PIC (*P*osition *I*ndependent *C*ode).**
- **Istruzioni operative:**
  - **operando di memoria individuato da un identificatore *ind*:**
    - **utilizzo dell'indirizzamento relativo rispetto a RIP, con indicazione esplicita di RIP, e quindi, in, calcolo da parte del traduttore (fase di traduzione) di DISP = ind – RIP;**
    - **in fase di esecuzione, calcolo dell'indirizzo dell'operando come DISP + RIP, ottenendo un indirizzo di operando traslato come RIP.**
- **Istruzioni di controllo:**
  - **indirizzo di salto individuato da un identificatore *ind*:**
    - **utilizzo dell'indirizzamento relativo rispetto a RIP (con RIP implicito), quindi, in fase di traduzione, calcolo di DISP = ind – RIP;**
    - **in fase di esecuzione, calcolo dell'indirizzo dell'operando come DISP + RIP, ottenendo un indirizzo di salto traslato come RIP.**
- **Programmi indipendenti dalla posizione:**
  - **la proprietà PIC vale sia per le istruzioni operative che per quelle di controllo, e quindi per tutto il programma.**

Semplice esempio numerico (1)

Programma PIC

```
 Indirizzo-Target     RIP

alfa:   5             6       .byte   'a'




ciclo:  10
        12            14      movb    alfa(%rip), %ah
        14            15      jpl     ciclo                   # RIP imlicito
        15            16
```

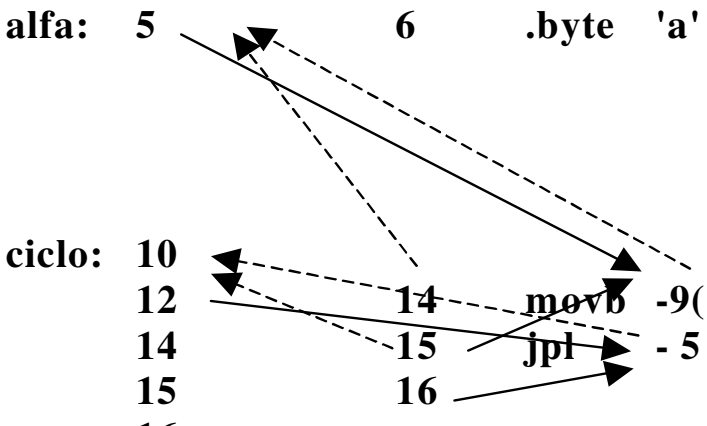
Semplice esempio numerico (2)

Posizione di partenza: 0

Indirizzo-Target **RIP**

Indirizzo oerando/salto

In traduzione **(linea intera):**
DISP = indirizzo_target – RIP.
In Esecuzione **(linea tratteggiata):**
Indirizzo Operando/Salto = DISP + RIP



```
alfa:   5          6        .byte   'a'

ciclo:  10
        12         14       movb    -9(%rip), %ah
        14         15       jpl     - 5
        15         16
        16
```

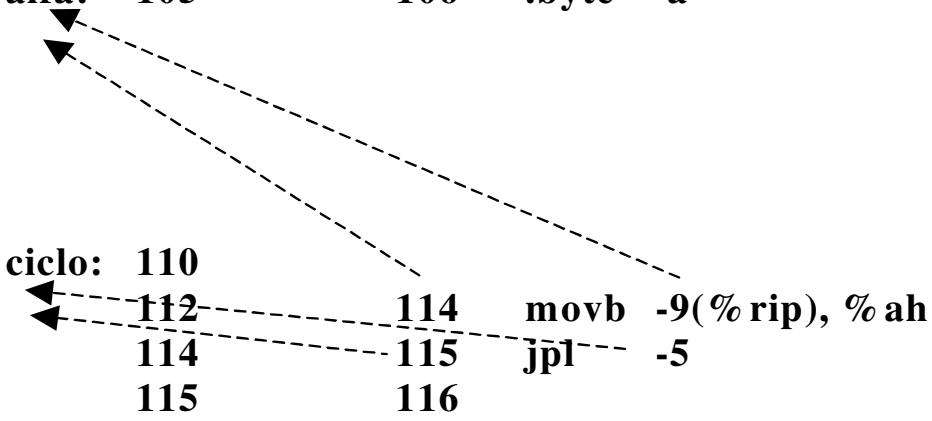
In traduzione:
DISP = 5 – 14 = -9
DISP = 10 – 15 = -5

In esecuzione**:**
All'istruzione di indirizzo 12: Indirizzo operando = -9+14 = 5 (*alfa*)
All'istruzione di indirizzo 14: Indirizzo salto = -5 + 15 = 10 (*ciclo*)

Semplice esempio numerico (3)

Stesso programma, posizione di partenza: 100

***In esecuzione*:**
Calcolo dell'indirizzo operando/salto:
DISP + RIP



***Esecuzione*:**
All'istruzione di indirizzo 112, indirizzo operando = - 9 + 114 = 105 (*alfa*)
All'istruzione di indirizzo 114, calcolo indirizzo salto = – 5 + 115= 110 (*ciclo*)

Struttura di un semplice programma Assembler (1)

- **Programma Assembler:**
  - **costituito da due sezioni, una sezione dati (statici) e una sezione testo, oltre che dalla pila e dalla memoria dinamica (*heap*), contenenti dati temporanei (tipicamente, creati e distrutti dinamicamente);**
  - **memorizzato su uno o più file, ciascuno detto *file sorgente*.**
    - **in ogni file, sono presenti nessuna, una o più parti della sezione dati (ciascuna inizia con la pseudo-istruzione *.data*), e nessuna, una o più parti della sezione testo (ciascuna inizia con la pseudo-istruzione *.text*).**
- **Inizio del programma:**
  - **identificatore *_start,* che deve essere dichiarato globale (direttiva *.global*).**
- **Fine del programma:**
  - **risultato per il Sistema Operativo UNIX: va lasciato nel registro EBX;**
    - **per convenzione, quando vale 0 tutto si è svolto correttamente, mentre quando è diverso da 0, il valore costituisce la codifica di un errore.**
  - **il valore di ritorno precedente è visibile (utilizzando il linguaggio di script *bash*) attraverso la variabile *$?* (comando per vederne il valore: echo $?).**
- **Istruzioni di ritorno a UNIX:**

```
movl    …, %ebx
movl    $1, %eax        # numero d'ordine della primitiva UNIX exit
int     $0x80           # servizio UNIX 0x80
```

Struttura di un semplice programma Assembler (2)

```
.data
                ...
                dati
                ...
.text
# sottoprogramma
sottop:         ...
                ret

# programma principale
.global     _start
_start:         …
                call        sottop
                …

uscita:         movl        …, %ebx
                movl        $1, %eax
                int         $0x80
```

Sottoprogrammi e registri

- **File tutti in Assembler:**
  - **un sottoprogramma Assembler (programma chiamato) comunemente salva e ripristina (utilizzando la pila) il contenuto di tutti i registri utilizzati, tranne quello/quelli in cui lascia il risultato;**
  - **un programma chiamante (programma principale o sottoprogramma) può lasciare informazioni che non devono essere modificate in qualunque registro (tranne in quello/quelli in cui il chiamato lascia il risultato).**
- **File in linguaggi non in Assembler (possibilità esaminata per il C++ successivamente):**
  - **un sottoprogramma non in Assembler salva e ripristina (utilizzando la pila) solo il contenuto di ben precisi registri (registri *invarianti*) fra quelli utilizzati;**
  - **un programma chiamante di un sottoprogramma non in Assembler può lasciare informazioni che non devono essere modificate solo in registri *invarianti*.**

Sviluppo di un programma Assembler (1)

- **Un file sorgente:**
  - **deve avere estensione *s;***
  - **viene tradotto separatamente dagli altri, dando luogo a un nuovo file, il file *oggetto*.**
- **Traduzione:**
  - **avviene attraverso il programma *Assemblatore;***
  - **comando (l'opzione *-o* non può essere omessa e specifica il nome del file oggetto, che deve avere estensione *o*):**

    as id_file.s –o id_file.o
- **File listato (contiene anche la traduzione provvisoria):**
  - **viene generato, in fase di traduzione, con l'opzione *–a,* ed una eventuale ridirezione:**

    as id_file.s -o id_file.o –a > id_file.l
  - **la ridirezione *> id_file.l* fa sì che l'uscita standard del comando *as* non sia *cout* (comunemente associato al terminale), ma il file specificato *id_file.l*.**

Sviluppo di un programma Assembler (2)

- **File oggetto ottenuti dalle traduzioni:**
  - **devono essere collegati insieme per formare un unico file *eseguibile* (senza estensione);**
  - **programma eseguibile (contenuto nel file eseguibile):**
    - **viene composto a partire da determinati indirizzi della sezione dati e della sezione testo;**
    - **se tutti gli indirizzi di memoria sono relativi a *rip*, il programma eseguibile (nella sua interezza) è *indipendente dalla posizione.***
- **Collegamento:**
  - **avviene attraverso il programma *Collegatore* (*Linker*);**
  - **comando (l'opzione *–o* specifica il nome del file eseguibile, che non ha estensione):**

    ld id_file1.o …. id_fileN.o –o id_file
  - **l'opzione *–o* può essere omessa, ed in questo caso l'identificatore del file eseguibile è per default *a.out* (non esistendo l'estensione, tutti i caratteri presenti costituiscono l'identificatore del file).**
- **File mappa (contiene informazioni di collegamento):**
  - **viene generato in fase di collegamento con l'opzione *-M*:**

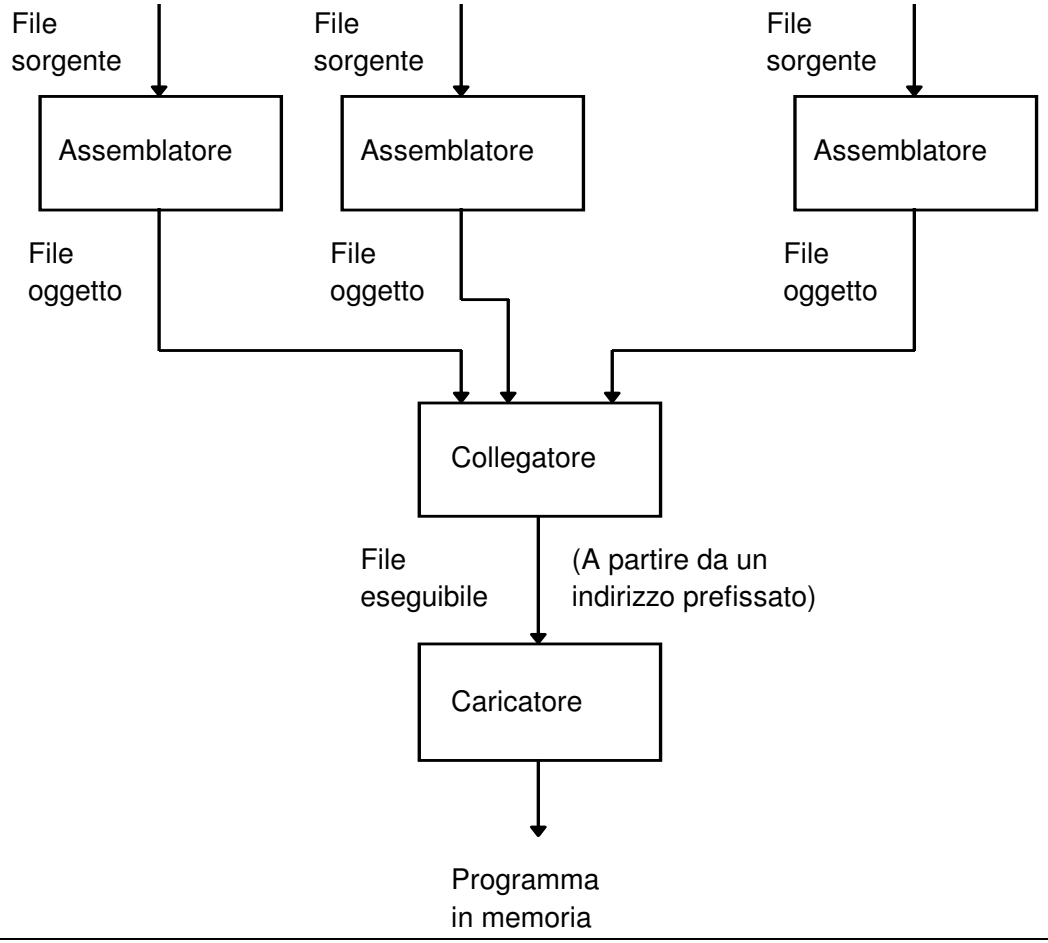
    ld id_file1.o -o … id_fileN.o -o id_file –M > id_file.m
  - **l'eventuale ridirezione *> id_file.m* fa sì che l'uscita standard del comando *ld* non sia *cout* (comunemente associato al terminale), ma il file specificato.**

Sviluppo di un programma Assembler (3)

- **Rilocazione e Caricamento:**
  - **il programma collegato, eventualmente rilocato nel caso in cui le sezioni *.text* e *.data* occupino zone indipendenti della memoria (la proprietà PIC vale solo se le due zone *.text* e *.data* vengono allocate mantenendo la loro eventuale distanza), viene poi caricato nella memoria stessa, e quindi eseguito;**
  - **comando:**

    ./id_file
- **Esecuzione del programma:**
  - **il Caricatore:**
    - **inizializza il puntatore di pila;**
    - **pone in pila il valore del nuovo contatore di programma (corrispondente all'identificatore *_start*), e di altri registri (CPL ed EFLAGS trattati in seguito);**
    - **esegue l' istruzione IRETQ (ritorno da interruzione).**
- **Programma costituito da un solo file:**
  - **viene sviluppato secondo le fasi sopra elencate;**
  - **il collegamento viene comunque effettuato, eventualmente con alcuni programmi di libreria, già tradotti una volta per tutte.**
- **Fase di traduzione (in senso lato):**
  - **costituita dalla traduzione vera e propria, dal collegamento, ed eventualmente dalla rilocazione.**

Rappresentazione grafica



I/O di singoli caratteri (utilizzando UNIX) e il file *ser.s*

- **Il file s*er.s*, scritto in Assembler, contiene:**

  due sottoprogrammi *tastiera* e *video*:

  - ***tastiera*: legge il successivo carattere battuto a tastiera e pone il suo codice ASCII nel registro AL;**
  - ***video*: scrive su video il carattere il cui codice ASCII è contenuto in BL;**
  - ***tastiera* e *video* utilizzano il servizio UNIX 0x80 (con opportuni parametri in RAX, RBX, RCX, RDX).**

  una routine *uscita*:

  - **restituisce il controllo al Sistema Operativo (registro EBX: se contiene 0 non ci sono stati errori), utilizzando anch'essa il servizio UNIX 0x80:**

```
uscita:  movl    $0, %ebx    # risultato
         movl    $1, %eax    # primitiva exit
         int     $0x80
```

- **Sottoprogrammi *tastiera* e *video*: effettuano ingresso/uscita a linee (il servizio UNIX 0x80 gestisce l'I/O in modo bufferizzato):**
  - **i caratteri battuti a tastiera, che compaiono in eco su video, vengono effettivamente letti quando da tastiera viene premuto il tasto *Enter*;**
    - **in lettura, *Enter* viene riconosciuto come carattere '\n' (nuova linea);**
  - **i caratteri inviati su video vengono effettivamente visualizzati quando viene inviato su video il carattere '\n' (nuova linea);**
    - **viene inserito dal driver del video anche il carattere '\r' (ritorno carrello).**

File *ser.s*

- **File *ser.s* (da includere nei file Assembler che effettuano I/O):**

```
# file ser.s
.data
 buff:        .byte 0
.text
 tastiera:
                ...
                ret
 video:
                 ...
                ret
 uscita:        movl   $0, %ebx       # risultato per UNIX
                movl   $1, %eax       # primitiva UNIX exit
                int    $0x80
```

- **Sottoprogrammi *tastiera* e *video*:**
  - **salvano e ripristinano tutti i registri utilizzati (tranne RAX per il sottoprogramma *tastiera,* che lascia il risultato in AL).**

Lettura di un carattere

- **Lettura di un carattere da tastiera e suo trasferimento in AL:**
  - **utilizzo del servizio UNIX 0x80, con parametri in RAX, RBX, RCX, RDX.**

```
tastiera:
        pushq   %rbx
        pushq   %rcx
        pushq   %rdx
        movq    $3, %rax            # primitiva UNIX read
        movq    $0, %rbx            # ingresso standard
        leaq    buff(%rip), %rcx    # indirizzo buffer di ingresso
        movq    $1, %rdx            # numero di byte da leggere
        int     $0x80
        movb    buff(%rip), %al
        popq    %rdx
        popq    %rcx
        popq    %rbx
        ret
```

Scrittura di un carattere

- **Scrittura su video del carattere contenuto in BL:**
  - **utilizzo del servizio UNIX 0x80, con parametri RAX, RBX, RCX, RDX.**

```
video:
        pushq   %rax
        pushq   %rbx
        pushq   %rcx
        pushq   %rdx
        movb    %bl, buff(%rip)
        movq    $4, %rax            # primitiva UNIX write
        movq    $1, %rbx            # uscita standard
        leaq    buff(%rip), %rcx    # indirizzo buffer di uscita
        movq    $1, %rdx            # numero byte da scrivere
        int     $0x80
        popq    %rdx
        popq    %rcx
        popq    %rbx
        popq    %rax
        ret
```

Programma *codifica* (1)

- **Programma *codifica*:**
  - **legge in AL caratteri fino al fine linea;**
  - **per ogni carattere letto, lo stampa e quindi ricava e stampa gli otto caratteri (in codifica ASCII) corrispondenti agli 8 bit della codifica del carattere letto, seguito da un fine-linea.**

```
# programma codifica, file codifica.s
.include   "ser.s"
.text
.global     _ start
_start:
ancora:   call       tastiera
          cmpb       $'\n', %al          # carattere nuova linea
          je         fine
          movb       %al, %bl            # carattere letto
          call       video
          movb       $' ', %bl           # carattere spazio
          call       video
```

Programma *codifica* (2)

```
        movb    $0,  %cl        # cl funge da contatore
ciclo:  testb   $0x80, %al      # esame del bit più significativo di al
        jz      zero
        movb    $'1', %bl
        call    video
        jmp     avanti
zero:   movb    $'0', %bl
        call    video
avanti: shlb    $1, %al         # traslazione sinistra di al
        incb    %cl
        cmpb    $8, %cl
        jb      ciclo
        movb    $'\n', %bl      # carattere nuova riga
        call    video
        jmp     ancora
fine:   jmp     uscita
```

Programma *codifica* (3)

- **File precedente:**
  - **deve essere prima tradotto;**
  - **il file tradotto deve essere poi collegato.**
- **Questo si ottiene con i comandi:**

  as codifica.s -o codifica.o

  ld codifica.o -o codifica
- **Il file eseguibile (*codifica*) può essere caricato e mandato in esecuzione con il comando:**

  ./codifica

Identificatori esterni e globali Assembler (dichiarazioni esterne opzionali, globali obbligatorie)

- **In un file si possono utilizzare identificatori definiti in altri file:**
  - **in Assembler, tali identificatori, per chiarezza, *possono* essere dichiarati esplicitamente *esterni;***
    - **se non vengono esplicitamente dichiarati esterni, vengono implicitamente considerati tali;**
  - **identificatori dichiarati esterni:**
    - **si utilizza la direttiva *.extern*:**

      .extern identificatore, ...
- **Alcuni identificatori definiti in un file possono essere utilizzati da altri file:**
  - **in Assembler, tali identificatori, *obbligatoriamente*, vanno dichiarati globali;**
  - **identificatori dichiarati globali:**
    - **si utilizza la direttiva *.global* (o *.globl*):**

      .global identificatore, ...
  - **gli identificatori non dichiarati globali sono propri di quel file;**
  - **uno stesso identificatore non globale può essere utilizzato in file diversi, e si riferisce a entità diverse.**

Punto di inizio

- **File sorgente Assembler:**
  - **può prevedere il punto di inizio (*entry_point*) dell'intero programma (identificatore *_start*).**
- **File principale:**
  - **contiene *l'entry_point*;**
  - **ne deve esistere uno e uno solo.**
- **File secondari:**
  - **altri file.**
- **Situazione comune:**
  - **il file principale contiene alcuni dati e il programma principale, con alcuni sottoprogrammi, mentre un file secondario contiene solo altri dati e altri sottoprogrammi.**
- **Indirizzo iniziale (identificatore *_start*):**
  - **occorre dichiararlo globale, per renderlo visibile al Caricatore (consentendo ad esso di trasferire tale indirizzo in pila, e quindi, tramite l'istruzione IRETQ, di inizializzare opportunamente il registro RIP).**

Programma *codifica1* (1)

- **Programma *codifica1* (prima nuova versione del programma *codifica)*:**
  - **due file: il primo contiene il programma principale, il secondo un sottoprogramma *esamina*, utilizzato dal primo file.**
- **Programma principale:**
  - **legge caratteri fino al fine linea;**
  - **per ogni carattere, oltre a stamparlo, richiama il sottoprogramma *esamina*, quindi stampa il risultato prodotto da quest'ultimo.**
- **Sottoprogramma *esamina*:**
  - **restituisce otto caratteri in codifica ASCII, corrispondenti agli 8 bit della codifica del carattere ricevuto.**
- **Trasmissione dei dati fra programma e sottoprogramma:**
  - **due variabili *alfa* e *beta* definite nel secondo file (esterne nel primo file e globali nel secondo);**
    - ***alfa*: contiene il codice del carattere, che il sottoprogramma deve esaminare;**
    - ***beta:* contiene l'indirizzo di una variabile array di 8 byte, dove il sottoprogramma deve porre il risultato.**
  - **il programma principale pone i dati in *alfa* e *beta*, quindi chiama *esamina*.**

Programma *codifica1* (2)

```
# Programma codifica1
# file principale codifica1a.s
# .global obbligatorio, .extern opzionale

.include    "ser.s"
# .extern   alfa, beta, esamina
.data
kappa:      .fill       8, 1

.text
.global     _start
_start:
ancora:     call        tastiera
            cmpb        $'\n', %al
            je          fine
            movb        %al, %bl
            call        video
            movb        $' ', %bl
            call        video
            movb        %al,  alfa(%rip)
            leaq        kappa(%rip),  %rax
            movq        %rax, beta(%rip)
            call        esamina
            leaq        kappa(%rip), %rax
            movq        $0, %rsi
ripeti:     movb         (%rax, %rsi), %bl
            call        video
            incq        %rsi
            cmpq        $8, %rsi
            jb          ripeti
            movb        $'\n', %bl
            call        video
            jmp         ancora
fine:       jmp         uscita
```

Programma *codifica1* (3)

```
# Programma codifica1
# file secondario codifica1b.s
# .global obbligatorio, .extern opzionale
.data
.global     alfa, beta
alfa:       .byte       0
beta:       .quad       0

.text
.global     esamina
esamina:    pushq       %rax
            pushq       %rbx
            pushq       %rsi
            movb        alfa(%rip), %al     # I dato
            movq        beta(%rip), %rbx    # II dato
            movq        $0, %rsi
ciclo:      testb       $0x80, %al
            jz          zero
            movb        $'1', (%rbx, %rsi)
            jmp         avanti
zero:       movb        $'0', (%rbx, %rsi)
avanti:     shlb        $1, %al
            incq        %rsi
            cmpq        $8, %rsi
            jb          ciclo
            popq        %rsi
            popq        %rbx
            popq        %rax
            ret
```

Programma *codifica1* (4)

- **File precedenti:**
  - **devono essere prima tradotti;**
  - **i file tradotti devono essere poi collegati.**
- **Questo si ottiene con i comandi:**

  as codifica1a.s -o codifica1a.o

  as codifica1b.s -o codifica1b.o

  ld codifica1a.o codifica1b.o -o codifica1
- **Il file eseguibile (*codifica1*) può essere caricato e mandato in esecuzione con il comando:**

  ./codifica1

Comando *g*++ (1)

- **Il comando g++:**

  g++ id_file1.s … id_fileN.s –o id_file

  – **richiama singolarmente l'Assemblatore per i file *id_file1.s*, …, *id_fileN.s*, producendo *N* file oggetto aventi uguale identificatore ed estensione *o*;**
  – **richiama il Collegatore per gli *N* file oggetto precedenti e alcuni file di libreria, producendo il file eseguibile *id_file*;**
  – **l'opzione *–o id_file* può essere omessa, ed in questo caso l'identificatore del file eseguibile è *a.out*.**

Comando *g++* (2)

- **Collega le librerie del C++.**
- **Un dei file dei file collegati funge da interfaccia con l'Assembler:**
  - **contiene l'entry_point *_start* (identificatore esplicitamente dichiarato globale);**
  - **richiama il sottoprogramma *main()* (identificatore esterno);**
  - **il sottoprogramma *main()* deve produrre un risultato intero che lascia in EAX;**
  - **al ritorno dal sottoprogramma *main(),* restituisce il controllo a UNIX con le istruzioni:**

```
movl    %eax, %ebx      # risultato di main() lasciato in %eax e trasferito in %ebx
movl    $1, %eax        # primitiva UNIX exit
int     0x80
```

  - **utilizzando il comando g++, occorre pertanto organizzare il programma principale come un sottoprogramma, avente identificatore *main* e che lascia il risultato in EAX.**
- **Organizzazione di un programma Assembler sviluppato con g++:**

```
.data
 …
.text
.include    "ser.s"
 .global     main
 main:       …
             movl  $0, %eax
             ret
```

I/O in Assembler come in C++ (1)

- **Comando g++:**
  - **collega anche la libreria di I/O del C++.**
- **Modo utilizzato per fare I/O (orientato al carattere o formattato) in Assembler (come in C++):**
  - **definire un file *servi.cpp* con funzioni C++ per fare I/O, come il seguente, e farlo tradurre in Assembler dal Compilatore;**
  - **la dichiarazione *extern "C"* lascia inalterato in Assembler l'identificatore di una funzione C++ (il compilatore C++ si comporta, a questo fine, come il compilatore C). Se si vuole questo, la dichiarazione *è obbligatoria*.**

```
// file servi.cpp
#include<iostream>
using namespace std;
extern "C" char leggisucc()
{     char c; cin.get(c); return c;     }
extern "C" char leggichar()
{     char c; cin >> c; return c;       }
extern "C" int leggiint()
{     int i; cin >> i; return i;                }
extern "C" unsigned leggiex()
{     unsigned u; cin>>hex >>u>>dec; return u;  }
```

```
extern "C" void scrivisucc(char c)
{    cout.put(c);          }
extern "C" void scrivichar(char c)
{     cout << c << ' ';    }
extern "C" void scriviint(int i)
{     cout << i << ' ';    }
extern "C" void scriviex(unsigned int u)
{     cout << hex << u << dec << ' ';            }
extern "C" void nuovalinea()
{      cout << endl;                    }
```

I/O in Assembler come in C++ (2)

- **Traduzione in Assembler di un file C++:**
  - **il comando g++, con l'opzione –S, effettua la traduzione in Assembler di un file con estensione *cc* o *cpp*, producendo un nuovo file con lo stesso nome ed estensione *s*;**
  - **nel caso precedente, il comando è così fatto: *g++ -S servi.cpp;***
  - **il file *servi.s* ottenuto contiene sottoprogrammi Assembler con lo stesso identificatore e funzionalmente equivalenti a quelli C++, che saranno illustrati in dettaglio nella slide seguente;**
  - **il file *servi.s* può essere incluso nei file Assembler che effettuano I/O;**
  - **il programma deve essere sviluppato utilizzando il comando g++.**
- **Sottoprogrammi Assembler ottenuti da funzioni C++;**
  - **come detto (slide 31) le funzioni C++ salvano e ripristinano solo il contenuto di alcuni fra i registri utilizzati (i cosiddetti registri *invarianti*) (e non di tutti quelli utilizzati),**

    Registri invarianti: RSP, RBP, RBX, R12-R15
  - **pertanto un programma Assembler, se utilizza un sottoprogramma contenuto nel file *servi.s*, non deve lasciare informazioni che non devono essere modificati in registri non invarianti.**

I/O in Assembler come in C++ (3)

```
# file servi.s
leggisucc:    # legge da tastiera il carattere successivo (anche uno spazio bianco)
              # e pone la sua codifica ASCII in AL;
leggichar:    # salta eventuali spazi bianchi, legge da tastiera il successivo
              # carattere diverso da spazio bianco e pone la sua codifica ASCII in AL;
leggiex:      # salta eventuali spazi bianchi, legge da tastiera una sequenza di caratteri
              # congruente con la rappresentazione in base 16 di un naturale
              # e la converte in un naturale in base due che pone in EAX;
leggiint:     # salta eventuali spazi bianchi, legge da tastiera una sequenza di
              # caratteri congruente con la rappresentazione in base dieci di un
              # intero, la converte in un intero base due che pone in EAX;
scrivisucc:   # scrive su video il carattere la cui codifica ASCII si trova in DIL;
scrivichar:   # scrive su video il carattere contenuto in DIL seguito dal carattere spazio;
nuovalinea: # scrive su video un fine linea (carattere '\n', con inserimento
              # implicito, come in C++, del carattere '\r').
scriviex:     # converte il naturale espresso in base due contenuto in EDI in una sequenza
              # di  caratteri che rappresentano cifre in base sedici, e scrive tale sequenza
              # su video;
scriviint:    # converte l' intero espresso in base due contenuto in EDI in una sequenza
              # di caratteri che rappresentano cifre in base dieci, e scrive tale sequenza
              # su video.
```

Programma *codifica2* (1)

Programma *codifica2*: seconda versione del programma *codifica:*

```
#  .global obbligatorio, .extern opzionale

# Programma codifica2
# file principale codifica2a.s
# utilizzo di registri invarianti

.include    "servi.s"
# .extern   alfa, beta, esamina
.data
kappa:      .fill       8, 1

.text
.global     main
main:
ancora:     call        leggisucc
            movb        %al, %r12b
            cmpb        $'\n', %r12b
            je          fine
            movb        %r12b, %dil
            call        scrivichar
            movb        %r12b, alfa(%rip)
            leaq        kappa(%rip), %rax
            movq        %rax, beta(%rip)
            call        esamina  # produce 8 char

            movq        $0, %r12
            leaq        kappa(%rip), %r13
ripeti:     movb        (%r13, %r12), %dil
            call        scrivisucc
            incq        %r12
            cmpq        $8, %r12
            jb          ripeti
            call        nuovalinea
            jmp         ancora
fine:       movl        $0, %eax
            ret
```

Programma *codifica2* (2)

```
# Programma codifica2, file secondario codifica2b.s
# uguale al file secondario codifica1b.s
# .global obbligatorio, .extern opzionale

.data
.global     alfa, beta
alfa:       .byte       0
beta:       .quad       0

.text
.global     esamina
esamina:    pushq       %rax
            pushq       %rbx
            pushq       %rsi
            movb        alfa(%rip), %al     # I dato
            movq        beta(%rip), %rbx    # II dato
            movq        $0, %rsi
ciclo:      testb       $0x80, %al
            jz          zero
            movb        $'1', (%rbx, %rsi)
            jmp         avanti
zero:       movb        $'0', (%rbx, %rsi)
avanti:     shlb        $1, %al
            incq        %rsi
            cmpq        $8, %rsi
            jb          ciclo
            popq        %rsi
            popq        %rbx
            popq        %rax
            ret
```

- **Sviluppo ed esecuzione del programma:**
- g++ codifica2a.s codifica2b.s –o codifica2
- ./codifica2

Programmi C++ su più file (dichiarazioni esterne obbligatorie, globali opzionali)

- **Regole di collegamento semplificate:**
  - **in un file C++, gli identificatori delle variabili definite al di fuori delle funzioni e gli identificatori delle funzioni:**
    - **sono *implicitamente* globali (la dicharazione *.global* può quindi essere omessa);**
  - **in un file si possono riferire identificatori definiti in altri file, purché:**
    - **siano *esplicitamente* dichiarati esterni: per le funzioni la parola chiave *extern* può essere omessa, ma resta obbligatoria la dichiarazione;**
    - **siano globali nei file dove sono definiti.**
- **Dichiarazione di identificatori esterni (simbolo *extern*):**
  - **variabile:**

    extern type1 var1, type2 var2 …;
  - **nella dichiarazione di una funzione:**

    extern type fun(type1,…, typeN);
  - **nella definizione di una funzione:**

    extern type fun(type1 par1,…, typeN parN) { }
- **Ingresso/uscita:**
  - **per ragioni di uniformità, per fare I/O in file Assemble si include *servi.s,* e per fare I/O in file C++ si include *servi.cpp*.**

Programma codifica3 (1)

- **Programma codifica3:**
  - **terza nuova versione del programma codifica.**
  - **// Programma codifica3,**
  - **/ file principale codifica3a.cpp (extern obbligatorio, global opzionale)**

```
#include "servi.cpp"
extern char alfa, char* beta;
extern void esamina();                          // paroola chiave extern può essere omesso
char kappa[8];
int main()
{     char al;
      for(;;)
       {        al =leggisucc();
                if (al == '\n') break;
                scrivichar(al);
                alfa = al; beta = &kappa[0];          // anche beta = kappa;
                esamina();
                for (int i=0; i<8; i++) scrivisucc(kappa[i]);
                nuovalinea();
      };
      return 0;
}
```

Programma codifica3 (2)

```
// Programma codifica3, file secondario codifica3b.cpp
// extern obbligatorio, global opzionale

char alfa; char* beta;                                  // implicitamente globali
void esamina()                                          // implicitamente globale
{     for(int i = 0; i < 8; i++)
      { if ((alfa & 0x80) == 0)
              *(beta+i) = '0';  else *(beta+i) = '1';          // anche beta[i]
        alfa = alfa<<1;
      }
}
```

Programmi misti (1)

- **Programma organizzato su più file in GCC:**
  - **può essere scritto utilizzando linguaggi *differenti* per i vari file (Assembler, C, C++);**
  - **ciascun file viene tradotto con il proprio traduttore:**
    - **i compilatori C e C++ producono file Assembler;**
    - **l'Assemblatore produce file oggetto, ciascuno avente lo stesso identificatore di quello tradotto.**
  - **il Collegatore (unico) produce il programma eseguibile.**
- **GCC, comando *g++*:**
  - **richiama i traduttori per i vari file;**
    - **singoli file con estensione *c*: richiama il Compilatore C;**
    - **singoli file con estensione *cc o cpp*: richiama il compilatore C++;**
  - **il compilatore richiamato produce un file con estensione *s;***
  - **per singoli file con estensione *s*, compresi quelli generati nel punto precedente, richiama l'Assemblatore, che produce un file avente lo stesso identificatore di quello tradotto ed estensione *o*;**
  - **richiama il Collegatore, inserendo anche eventuali file coinvolti appartenenti alla libreria C++.**

Programmi misti (2)

- **Compilatori C/C++ (traducono in Assembler):**
  - **inseriscono le variabili globali, non modificando i loro identificatori, nella sezione dati;**
  - **inseriscono le funzioni nella sezione testo: il compilatore C non modificando i loro identificatori, il compilatore C++ modificandoli;**
  - **utilizzano determinati standard per l'aggancio delle funzioni;**
    - **i parametri e le variabili locali delle funzioni coinvolgono registri e locazioni della pila.**
- **Intero programma:**
  - **consistente solo se i differenti linguaggi utilizzati:**
    - **fanno uso della stessa modalità di rappresentare i dati;**
    - **utilizzano gli stessi identificatori;**
    - **utilizzano lo stesso standard per l'aggancio delle funzioni (sottoprogrammi in Assembler,);**
    - **i contenuti dei registri generali obbediscono alle stesse regole di salvataggio/ripristino (stessi registri invarianti).**
- **Linguaggi C++ e Assembler:**
  - **i file Assembler sviluppati dal programmatore dovranno utilizzare le stesse regole (rigide) che utilizza il compilatore C++ per produrre file Assembler.**

Identificatori C e Identificatori C++

- **Identificatori di variabili e di funzioni C:**
  - **sono uguali agli identificatori Assembler.**
- **Identificatori di variabili C++:**
  - **sono uguali agli identificatori Assembler (come per il C).**
- **Identificatori di funzioni C++:**
  - **per assicurare la possibilità di avere *overloading,* vengono tradotti con identificatori Assembler ottenuti secondo regole che tengono conto del numero, del tipo e dell'ordine degli argomenti formali;**
  - **possono essere tradotti con gli stessi identificatori Assembler (avere la corrispondenza del compilatore C), purché vengano dichiarati (anche nelle dichiarazioni o nelle definizioni di funzioni) *extern "C"*.**
- **Ipotesi per gli identificatori di funzione:**
  - **al momento, verrà utilizzata la corrispondenza C (uguaglianza di identificatori), rinunciando ad avere overloading;**
    - **tale corrispondenza non può essere applicata alle funzioni membro di classi, non essendo le classi previste in C;**
  - **in un secondo momento, verranno illustrate le regole seguite dal compilatore C++.**

Identificatori di funzioni C++ uguali agli identificatori C (e Assembler)

- **Identificatori delle funzioni C++ tradotti come se fossero identificatori di funzioni C:**
  - **nelle dichiarazioni e nelle definizioni di funzioni occorre aggiungere la specifica *extern "C";***
  - **pertanto, in un file C++:**
    - **le funzioni esterne vanno dichiarate, prima di essere utilizzate, nel seguente modo:**

      extern "C" type fun(…);
    - **le funzioni globali vanno definite nel seguente modo:**

      extern "C" type fun(…)
      { // …
      }
    - **i corpi delle funzioni non vengono modificati dalla specifica *extern "C"*.**
- **La funzione C++ *main()* è implicitamente definita *extern "C"*.**

Dati discreti (non reali) in C++

- **Tipi di dato considerati:**
  - **tipo *short int*: 2 byte, rappresentati in complemento a 2;**
  - **tipo *int*: 4 byte, rappresentati in complemento a 2;**
  - **tipo *long int*: 8 byte, rappresentati in complemento a 2;**
  - **tipi *unsigned int*: stesso numero di byte dei tipi *int*, semplicemente espressi in base 2 (non ci sono negativi);**
  - **tipo *bool*: un naturale di un byte, con valore 0 equivalente a *false* e valore 1 equivalente a *true*;**
  - **tipo *char*: un naturale di un byte, in codifica ASCII (il bit più significativo vale 0), o un intero di un byte, in complemento a 2;**
  - **tipi *enumerati*: un tipo *unsigned int*, con il valore di ogni enumeratore dato dal suo numero d'ordine, a partire da 0;**
  - **tipi *puntatore*: 8 byte (indirizzo completo di memoria);**
  - **tipi *riferinento*: 8 byte (indirizzo completo di memoria);**
  - **tipi *array* (monodimensionale): codifica dei singoli elementi;**
  - **tipi *struttura* o *unione*: codifica dei singoli campi;**
  - **tipi *classe*: codifica dei singoli campi dato (come i tipi struttura).**

Allineamento delle variabili (semplificato)

- **Variabili singole:**
  - **allineate a indirizzi multipli della loro lunghezza (2, 4, 8);**
- **Variabile di un tipo array:**
  - **allineata come richiesto dal primo elemento;**
  - **all'interno dell'array, tutti gli elementi sono in sequenza.**
- **Variabile di un tipo struttura o unione:**
  - **allineata come il campo avente vincoli maggiori;**
    - **all'interno della struttura, ogni campo è allineato secondo i suoi vincoli.**
- **Variabile di un tipo classe:**
  - **allineata come richiesto dall'insieme dei campi dato, considerati come struttura;**
    - **all'interno dei campi dato della classe, ogni campo è allineato secondo i suoi vincoli.**
- **Allineamento in pila:**
  - **variabili lunghe 2 o 4 byte, allineate a indirizzi multipli della loro lunghezza;**
  - **variabili più lunghe di 4 byte, allineate a indirizzi multipli di 8.**

Programma misto *codifica4* (1)

```
// Programma codifica4, file principale codifica4a.cpp (extern obbligatorio,
// global opzionale)
#include "servi.cpp"
extern char alfa, beta;
extern "C" void esamina();       // differenza rispetto al file codifica3a.cpp: (extern " C ")
char kappa[8];
int main()
{     char al
      for(;;)
       {        al =leggisucc();
                if (al == '\n') break;
                scrivichar(al);
                alfa = al; beta = &kappa[0];      // anche beta = kappa;
                esamina();
                for (int i=0; i<8; i++) scrivisucc(kappa[i]);
                nuovalinea();
      };
      return 0;
}
```

Programma misto *codifica4* (2)

```
# Programma codifica4, file secondario codifica4b.s (.extern optionale, .global obbligatorio)

.data
.global     alfa, beta
alfa:       .byte       0
beta:       .quad       0
.text
.global     esamina     # nessun salvataggio e ripristino, non essendoci registri utilizzati
                        # il cui valore va preservato da un chiamante C++
esamina:    movb        alfa(%rip), %al         # I parametro
            movq        beta(%rip), %rdx        # II parametro
            movq        $0, %rsi
ciclo:      testb       $0x80, %al
            jz          zero
            movb        $'1', (%rdx, %rsi)
            jmp         avanti
zero:       movb        $'0', (%rdx, %rsi)
avanti:     shlb        $1, %al
            incq        %rsi
            cmpq        $8, %rsi
            jb          ciclo
            ret
```

Programma misto *codifica5* (1)

```
# Programma codifica5
# file principale codifica5a.s,
# uguale al file codifica2a.s
# .extern opzionale, .global obbligatorio


.include    "servi.s"
# .extern   alfa, beta, esamina
.data
kappa:      .fill       8, 1

.text
.global     main
main:
ancora:     call        leggisucc
            movb        %al, %r12b
            cmpb        $'\n', %r12b
            je          fine
            movb        %r12b, %dil
            call        scrivichar
            movb        %r12b, alfa(%rip)
            leaq        kappa(%rip), %rax
            movq        %rax, beta(%rip)
            call        esamina

            movq        $0, %r12
            leaq        kappa(%rip), %r13
ripeti:     movb        (%r13, %r12), %dil
            call        scrivisucc
            incq        %r12
            cmpq        $8, %r12
            jb          ripeti
            call        nuovalinea
            jmp         ancora
fine:       movl        $0, %eax
            ret
```

Programma misto *codifica5* (2)

```
// Programma codifica5, file secondario codifica5b.cpp (extern obbligatorio, global opzionale)

char alfa; char* beta;
extern "C" void  esamina()    // differenza rispetto al file codifica3b.cpp: extern " C "
{      for(int i = 0; i < 8; i++)
       { if ((alfa & 0x80) == 0)
                *(beta+i) = '0';  else *(beta+i) = '1';  // anche beta[i]
         alfa = alfa<<1;
       }
}
```

Sviluppo della versioni 3 e delle versioni miste 4 e 5

- **Sviluppo della versione C++ numero 3**

  g++ codifica3a.cpp codifica3b.cpp –o codifica3

  ./codifica3

- **Sviluppo della versione mista numero 4:**

  g++ codifica4a.s codifica4b.cpp –o codifica4

  ./codifica4

- **Sviluppo della versione mista numero 5:**

  g++ codifica5a.cpp codifica5b.s –o codifica5

  ./codifica5

Opzioni comuni per il comando g++

- **Opzione per il non collegamento di file di libreria:**
  - ***nostdlib***
  - **non collega implicitamente nessun file di libreria, ma solo quelli esplicitamente elencati;**
  - **in questo caso, se il programma principale è in Assembler, deve avere la struttura vista per il comando *as* (entry point *_start*).**
- **Opzione per la sola traduzione in Assembler di un file C++:**
  - ***-S***
  - **serve a esaminare come il compilatore traduce un file C++;**
  - **può essere utile agli studenti, ai fini della preparazione dell'esame scritto, anche se il file tradotto non è di semplice comprensione, in quanto il Compilatore applica regole generali;**
  - **non va comunque utilizzata in fase di esame scritto, in quanto viene rilevata e lo scritto annullato**

Modelli di programma

- **Modello:**
  - **stabilisce come si deve effettuare la memorizzazione delle parti del programma per assicurare che tutti gli indirizzi prodotti dalle istruzioni elaborative possano correttamente indirizzare i dati, e quelli prodotti dalle istruzioni di controllo possano saltare nei punti previsti.**
- **Parti di un programma:**
  - **sono 4, ossia la sezione *.text*, la sezione *.data*, la *pila* e lo *heap* (gli indirizzi devono comunque essere canonici, altrimenti la MMU non li traduce);**
  - **la pila e lo heap possono essere memorizzati ovunque, poiché viene utilizzato un indirizzamento canonico con un registro base (la pila viene indirizzata tramite i registri RSP o RBP e lo heap tramite un puntatore);**
  - **le zone istruzioni *.text* e *.data* devono obbedire a determinate regole**
- **File in C++ e modello**
  - **il concetto di modello ha effetto quando nel programma si utilizza un file C++ da far tradurre al compilatore;**
  - **un fle scritto direttamente in Assembler deve rispettare il modello e non viene modificato dal Compilatore.**

Modello grande

- **Modello di programma grande:**
  - **la sezione *.text* e la sezione *.data* possono essere memorizzate ovunque;**
  - **gli indirizzi simbolici vengono tradotti, modificati per collegamento ed eventualmente rilocati:**
  - **se rappresentabili con 32 bit, possono essere posti singolarmente nei campi DISP o IMM delle istruzioni che prevedono tali campi.**
  - **se richiedono per la rappresentazione più di 32 bit (uno o più dei 32 bit più significativi sono diversi dal bit alla loro destra), tipicamente 64, viene utilizzata dal Compilatore l'istruzione MOVABSQ, l'unica che possiede un campo IMM di 64 bit.**
- **Indirizzamenti usati nel modello:**
  - **per indirizzare un dato, il compilatore usa tipicamente un'*espressione canonica*, con un registro base di 64 bit preventivamente caricato con una istruzione MOVABSQ, quest'ultima avente un operando immediato di 64 bit;**
  - **per effettuare un salto incondizionato, il compilatore usa tipicamente un'espressione indiretta, con un registro preventivamente caricato con una istruzione MOVABSQ, quest'ultima avente un operando immediato di 64 bit;**
  - **per effettuare un salto condizionato, con cicli manifestamente corti,, il compilatore utilizza un'*Espressione relativa (rispetto a RIP)* .**

Esempio: Programma C++

```
// un solo file C++
#include"servi.cpp"
using namespace std;
int i; int ar[10] = { 30,30,30,30,30,30,30,30,30,30 };
void fai()
{     i = 5;
}
int main()
{    fai();
     ar[i] = 8;
     for ( i = 0; i < 10; i++ ) scriviint(ar[i]);
     nuovalinea;
}
```

Esempio: Modello grande

```
# Modello grande: un solo file Ass.
.data
i:          .long  0
ar:         .fill  10, 4, 30

.text
.include    "servi.s"
fai:        movabsq    $i, %r14
            movl        $5, (%r14)
            ret

.global     main
main:       movabsq    $fai, %rbx
            call       *%rbx

            movabsq    $ar, %r12
            movabsq    $i, %r14
            movslq     (%r14), %r15
            movl       $8, (%r12, %r15, 4)
            # ar[5] = 8
            # stampa tutti gli elementi di ar
            movl       $0, (%r14)
ciclo:      movslq     (%r14), %r15
            movl       (r12, %r15, 4), %edi
            movabsq    $scriviint, %r13
            call       *%r13
            incl       (%r14)
            cmpl       $10, (%r14)
            jl         ciclo       # RIP implicito
            movabsq    $nuovalinea, %r13
            call       *%r13

            movl       $0, %eax
            ret
```

Modello Grande (versione con 2 file (1))

```
# Modello grande: File principale
.data
#.extern      ar, i

.text
.include "servi.s"
#.extern      fai
.global       main
main:
          movabsq        $fai, %rbx
          call           *%rbx

          movabsq        $i, %r14
          movslq         (%r14), %r15
          movabsq        $ar, %r12
          movl           $8, (%r12, %r15, 4)
#         ar[5] = 8
# stampa tutti gli elementi di ar
          movl           $0, (%r14)
ciclo:    movslq         (%r14), %r15
          movl           (r12, %r15, 4), %edi
          movabsq        $scriviint, %r13
          call           *%r13
```

```
# .global obbligatorio, .extern opzionale


          incl           (%r14)
          cmpl           $10, (%r14)
          jl             ciclo      # RIP implicito
          movabsq        nuovalinea, %r13
          call           *%r13

          movl           $0, %eax
          ret
```

Modello Grande (versione con 2 file (2))

 # Modello grande: File secondario

```
# .global obbligatorio, .extern opzionale

.data
.global     ar, i
i:   .long  0
ar:  .fill  10, 4, 30

.text
.global     fai
fai:        movabsq   $i, %r15
            movl      $5, (%r15)     // i = 5
            ret
```

Modello piccolo

- **Modello di programma piccolo:**
  - **la sezione *.text* e la sezione *.data* vengono complessivamente memorizzate nei primi 2 Gbyte di memoria;**
  - **gli indirizzi numerici sono rappresentabili con 32 bit, e possono essere posti nel campo IMM o nel campo DISP, delle istruzioni che prevedono tali campi.**
- **Indirizzamenti usati nel modello:**
  - **con il modello piccolo, il Compilatore, per esprimere un indirizzo di memoria, può utilizzare un'espressione canonica, anche senza registro base;**
  - **il valore numerico che può assumere il campo DISP va da (2**31-1) a -2**31 (lo stesso dicasi per il campo IMM);**
  - **senza l'utilizzo del registro base, il massimo valore di DISP eventualmente presente nella prima istruzione deve assicurare di raggiungere l'indirizzo dell'ultima, e il minimo valore di DISP eventualmente presente nell'ultima istruzione deve assicurare di raggiungere l'indirizzo della prima:**
- **Blocco di memoria da utilizzare per la memorizzazione delle zone *.text* e *.data*:**
  - **deve quindi avere dimensioni inferiori ai 2**31 (2 Gigabyte).**

Modello piccolo

```
# Modello piccolo: file unico
# .global obbligatorio, .extern opzionale

.data
i:          .long   0
ar:         .fill   10, 4, 30

.text
.include "servi.s"

fai:        movl    $5, i
            ret

.global     main
main:       call    fai             # RIP implicito

            movslq  i, %r15
            movl    $8, ar(, %r15, 4)
```

```
    # stampa tutti gli elementi
            movl    $0, i
ciclo:      movslq  i, %r15
            movl    ar(, %r15, 4), %edi
            call    scriviint   # RIP implicito
            incl    i
            cmpl    $10, i
            jl      ciclo       # RIP implicito
            call    nuovalinea # RIP implicito

            movl    $0, %eax
            ret
```

Proprietà PIC (*P*osition *I*ndependent *C*ode)

- **Proprietà PIC:**
  - **indipendenza dalla posizione.**
- **Compilatore:**
  - **può tradurre tutte le istruzioni operative e tutte quelle di salto (condizionato o incondizionato) utilizzano un indirizzamento relativo rispetto a RIP;**
  - **può anche tradurre le istruzioni operative utilizzando un indirizzamento con espressioni canoniche aventi registro base, previo caricamento del registro base stesso con l'istruzione LEAQ, avente a sua volta un indirizzamento relativo.**
  - **può effettuare chiamate di funzioni esterne, usando il salto indiretto.**
- **Modello PIC:**
  - **la sezione testo e Sezione dati non devono occupare complessivamente più di 2 Gbyte consecutivi di memoria, in qualunque zona.**

Modelli di programma e proprietà PIC

- **Modello piccolo + PIC**
  - **la proprietà PIC può combinarsi col modello piccolo, ottenendo un modello piccolo esteso a tutta la memoria;**
  - **per operandi che si trovano in pila (o nello heap) non ci sono restrizioni rispetto a quanto detto, potendo essere memorizzati ovunque.**
- **Modello grande + PIC**
  - **la proprietà PIC può combinarsi col modello grande, facendo predisporre al Compilatore una *G*lobal *O*ffset *T*able (GOT) :**
  - **la tabella viene utilizzata per memorizzarvi indirizzi di 64 bi, ed utilizzata per caricare registri su cui fare un'indirezione (operatore C++ '*') o da utilizzare in un'espressione canonica.**
- **Esempi riportati nel testo ed esercizio di esame:**
  - **nessuna opzione per il comando g++**
  - **file C++: traduzione secondo il modello piccolo;**
  - **file Assembler: possono essere usate istruzioni PIC.**

Programma C++ (slide 75) e Modello PIC

```
# File Ass. unico
# .global obbligatoria, .extern opzionale

.data
i:          .long       0
ar:         .fill       10, 4, 30

.text
.include "servi.s"

fai:        movl        $5, i(%rip)
            ret

.global     main
main:       call        fai    # PIC implicito

            leaq        ar(%rip), %r12
            movslq      i(%rip), %r15
            movl        $8, (%r12, %r15, 4)
```

```
            # stampa tutti gli elementi
            movl        0, i(%rip)
ciclo:      movslq      i(%rip), %r15
            movl        (%r12, %r15, 4), %edi
            call        scriviint    # PIC implicito
            incl        i(%rip)
            cmpl        $10, i(%rip)
            jl          ciclo        # PIC implicito
            call        nuovalinea # PIC implicito

            movl        $0, %eax
            ret
```

Per le chiamate di funzioni esterne, per esempio di libreria, può usare l'indirizzamento indiretto

Modelli e ambiente GNU/GCC

- **Specifica del modello nell'ambiente GNU/GCC (col comando g++):**

  -mcmodel=small

  -mcmodel=large

  - **senza specifiche, il modello utilizzato è quello piccolo;**
- **Specifica della proprietà PIC nell'ambiente GNU/GCC (col comando g++):**

  -f pic

  -f no-pic

  senza specifiche, non viene rispettata la proprietà PIC.
- **Programmi riportati nel seguito ed esercizi d'esame:**
  - **per i file C++, non vengono specificate opzioni né per il modello né per la proprietà PIC (il modello è quindi quello piccolo, e la regola PIC non viene rispettata);**
    - **sono di dimensione modesta, per cui la zona *.data* la zona *.text* sono sicuramente contenute nei primi 2 Gbyte di memoria;**
  - **per i file Assembler:**
    - **essendo modello piccolo, si può utilizzare la proprietà PIC per le istruzioni operative e per quelle di salto a indirizzi interni;**
    - **per richiamare funzioni esterne, usare il salto indiretto.**