ELABORATORE x86-64
E PRINCIPI DI PROGRAMMAZIONE

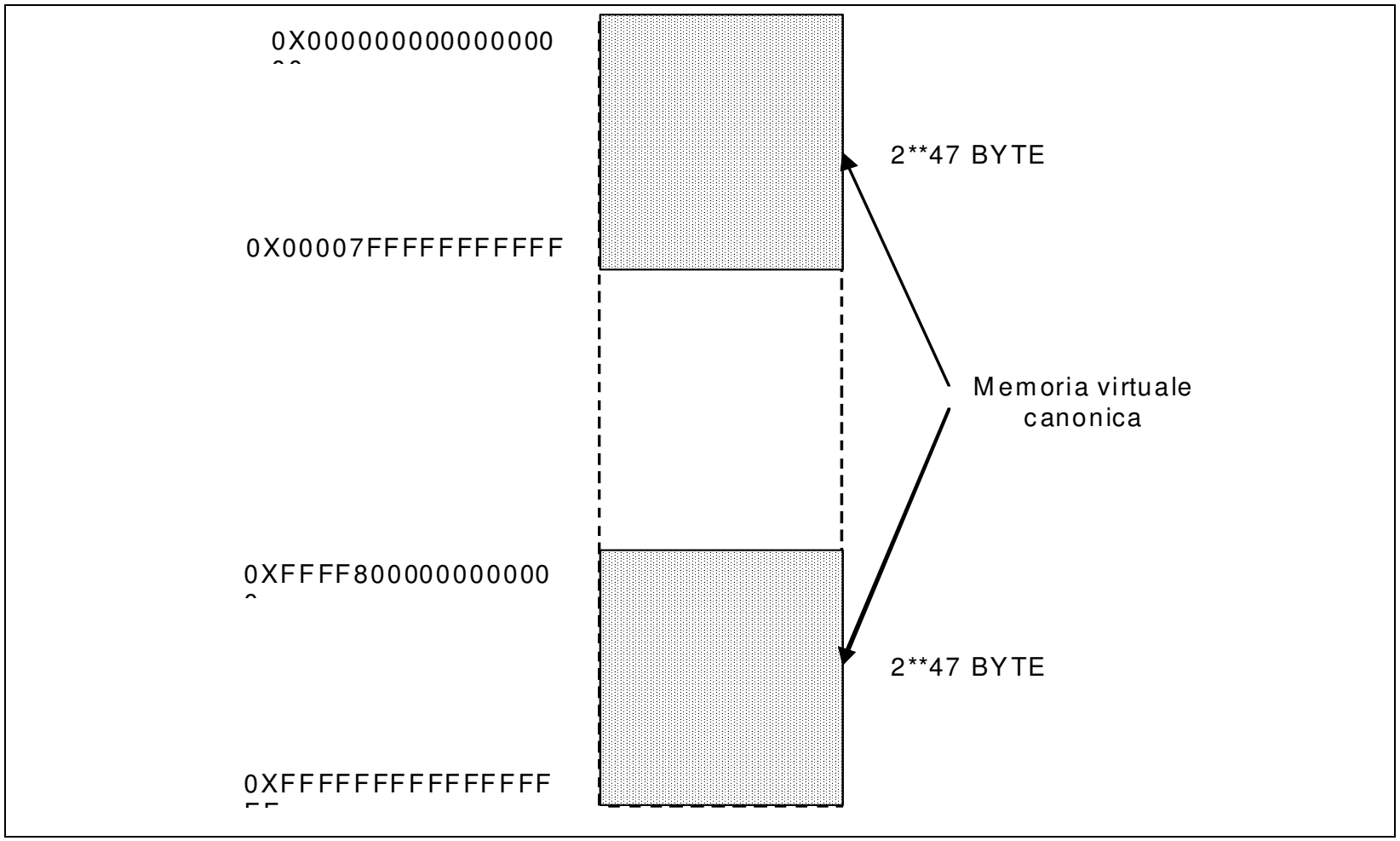
Caratteristiche del processore x86-64

- **Processore schematizzato x86-64 (Processore PC):**
  - **processore didattico;**
  - **schematizzazione (software compatibile) dei processori a 64 bit della famiglia INTEL, funzionanti in modo protetto (x86-64);**
  - **costituito da 2 tipi di 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).**
- **Operandi manipolati dalle istruzioni della FPU:**
  - **lunghi 80 bit.**
- **In questo corso esamineremo solo la ALU e non la FPU (e quindi le elaborazioni sui numeri reali).**

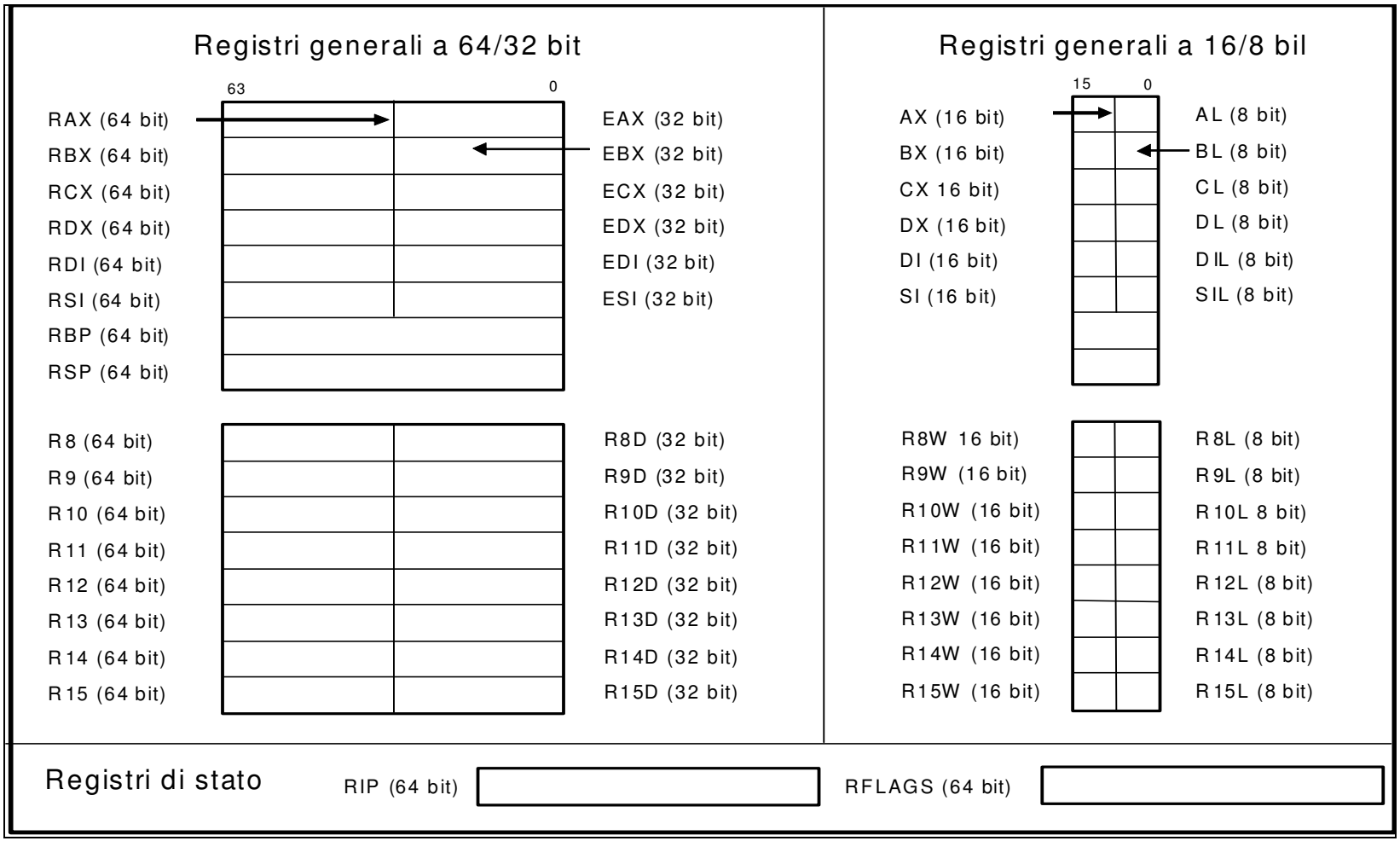
Spazi di indirizzamento

- **Funzionamento del Processore x86-64:**
  - **alla partenza, in modalità x86-16 (come il processore 8086);**
  - **dopo un'apposita istruzione, in modalità x86-64, con memoria virtuale.**
- **Spazio di memoria in modalità x86-64 (virtuale):**
  - **indirizzo lineare e costituito da 64 bit;**
  - **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.**
- **Spazio di I/O in modalità x86-64 (senza memoria virtuale):**
  - **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 due a due (porte di 16 bit) o quattro a quattro (porte di 32 bit).**

Spazio di memoria (virtuale)



Registri della ALU (1)

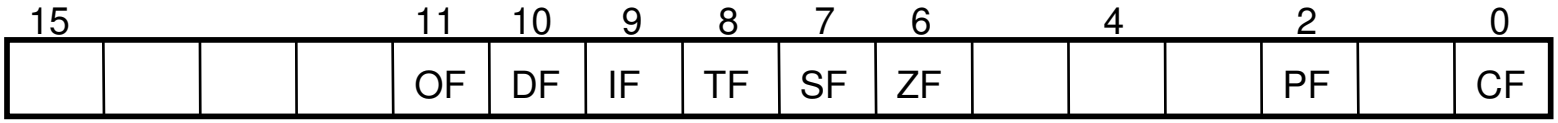


Registri della ALU(2)

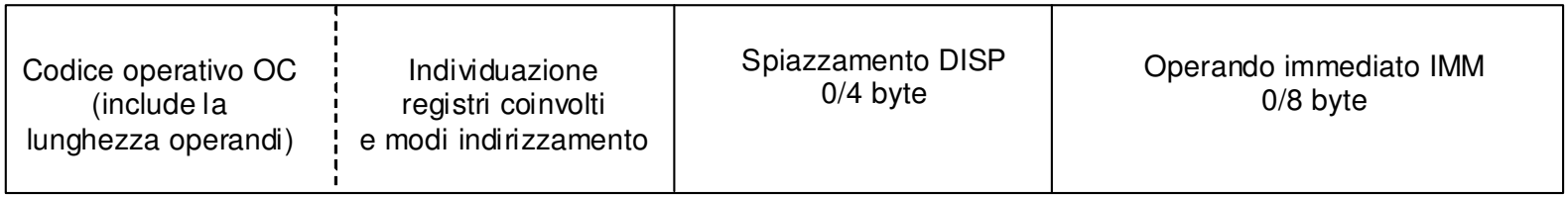
- **Registri generali:**
  - **registri generali a 64 bit:**
    - **sono 16;**
    - **possono venir indifferentemente utilizzati per memorizzare operandi e per contenere indirizzi di memoria;**
    - **alcuni di essi svolgono specifiche funzioni:**
      - **registri RSP, RBP,: servono a indirizzare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

Registri della ALU (3)

- **Registro di stato RIP (o registro contatore):**
  - **contiene l'indirizzo 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):**
  - **specifica il codice operativo e la lunghezza degli operandi;**
  - **il secondo campo contiene informazioni aggiuntive.**
- **Spiazzamento (DISPlacement):**
  - **assente**
  - **1, 2 oppure 4 byte.**
- **Operando immediato (IMMediate):**
  - **assente**
  - **da 1 a 4 byte**
  - **di 8 bye solo per l'istruzione MOVABSQ.**

Indirizzamenti per le istruzioni operative

- **Indirizzamento di registro:**
  - **operando in un registro del processore.**
- **Indirizzamento immediato:**
  - **operando nell'istruzione (campo IMM).**
- **Indirizzamento di memoria assoluto per le istruzioni operative (una o due componenti 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 bit, di 16 bit o di 32 bit, esteso a 64 bit.**
- **Indirizzamento di memoria relativo rispetto a RIP per le istruzioni operative:**

  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 a 64 bit.**

Indirizzamenti per 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 bit, 16 bit o di 32 bit, esteso a 64 bit.**
  - **indiretto:**

    INDIRIZZO = CONTENUTO DI UN REGISTRO;
    - **possibile solo per le istruzioni di salto incondizionato e di chiamata.**
- **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.**

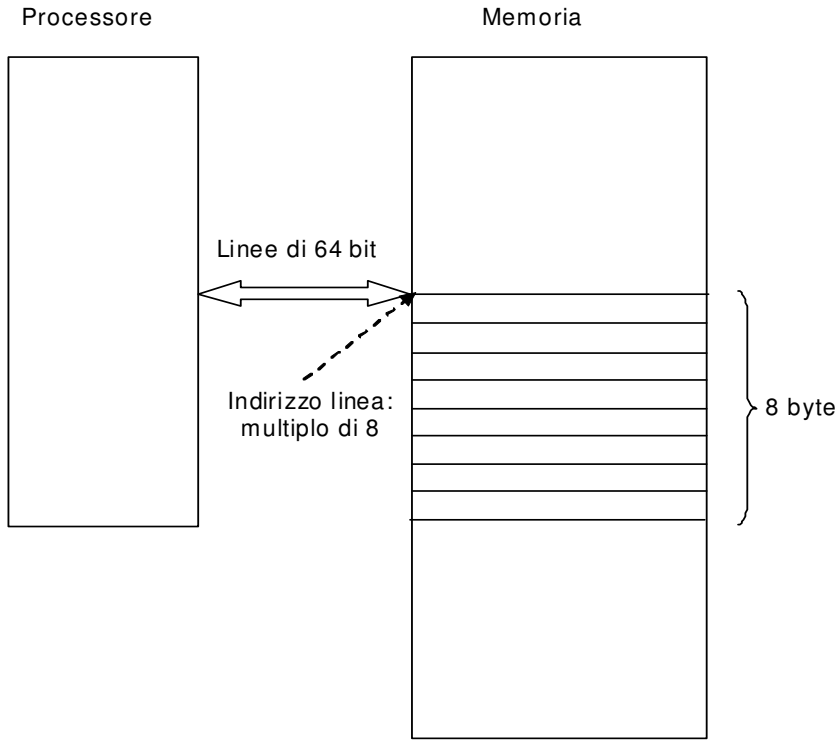
Indirizzo di memoria

- **Spiazzamento:**
  - **è un intero, rappresentato al più su 32 bit.**
- **Indirizzo assoluto:**
  - **a partire dal contenuto di un registro base RBASE (da 0 se il registro base è assente), consente di 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.**
  - **caricamento del registro base:**
    - **si effettua con l'istruzione MOVABSQ con un operando immediato di 64 bit, che può rappresentare qualunque indirizzo di memoria.**
- **Indirizzo relativo:**
  - **a partire dal contenuto del registro RIP, consente di indirizzare una zona di memoria che va da | RIP - 2**31 | modulo 2**64 a | RIP + 2**31-1| modulo 2**64.**

Organizzazione della memoria

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

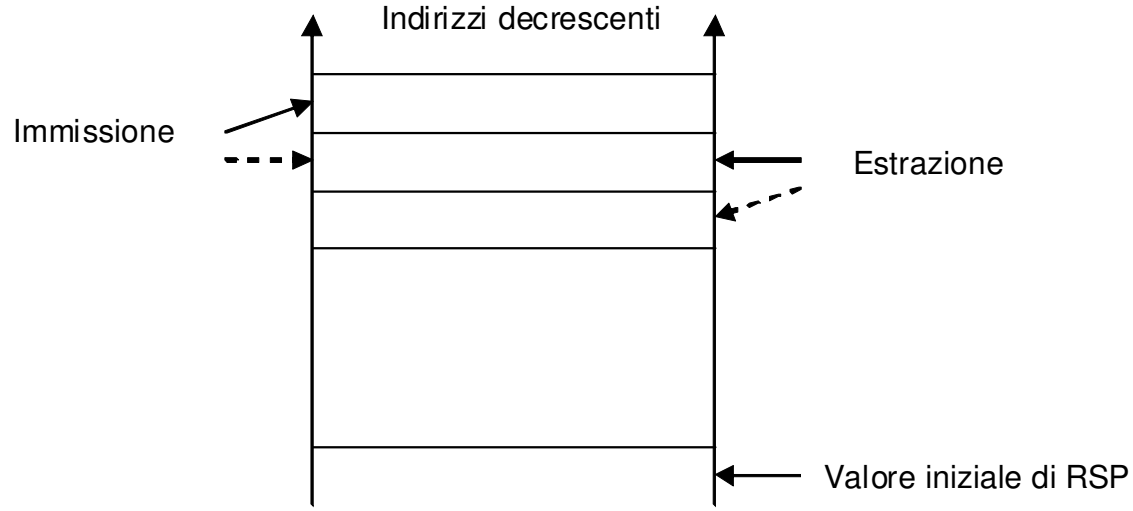


Indirizzo di istruzioni e di operandi

- **Istruzioni:**
  - **l' indirizzo di un'istruzione è quello del primo byte del Codice Operativo.**
    - **le istruzioni non possono venir 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 dell'operando sono memorizzati in ordine inverso;**
  - **l'indirizzo dell'intero operando è quello del byte meno significativo dell'operando stesso.**
- **Allineamento degli operandi:**
  - **un operando può stare a partire da qualunque indirizzo di memoria;**
  - **viene in genere allineato a indirizzi multipli della sua lunghezza, in modo che sia memorizzato in un sola linea, così da minimizzare il tempo richiesto per l' accesso all'operando stesso.**

La Pila

- **Pila:**
  - **organizzata a 64 bit, utilizzando come puntatore il registro RSP;**
  - **immissione (PUSHQ, CALL, …): decremento di 8 di RSP, quindi scrittura nella parola quadrupla indirizzata da RSP;**
  - **estrazione (POPQ, RET, …): lettura della parola quadrupla indirizzata da RSP, quindi incremento di 8 di RSP.**
- **Registro base per indirizzare locazioni della pila: RBP.**



UNIX e ambiente di programmazione GCC

- **Ambiente di programmazione GCC (GNU Compiler Collection):**
  - **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 processore INTEL x86-64 e sistema operativo UNIX.**
- **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.**
  - **i letterali si scrivono come in C++.**

Assembler GCC (1)

- **Forma di un commento:**
  - **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 viene fatto corrispondere un valore numerico (indirizzo di memoria);**
  - **un'espressione produce un risultato numerico.**

Assembler GCC (5)

- **Istruzioni operative con indirizzo di memoria assoluto:**
  - **indicazione Assembler dell'indirizzo dell'operando in memoria:**

    ***spiazzamento*(*base, indice, scala*)**
    - ***spiazzamento:*** **espressione che viene calcolata in fase di traduzione, il cui risultato 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.**
- **Esempi:**

mov*l* spiazzamento, %*l*reg	**ind. diretto**
mov*l* (%rbase), % *l*reg	**ind. indiretto**
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**

Assembler GCC (6)

- **Registri base e indice:**
  - **se il registro base e il registro indice sono assenti, il campo DISP rappresenta l'indirizzo dell'operando in memoria, che può spaziare in una zona limitata ;**
  - **il registro base può essere caricato con un operando immediato (istruzione *movabsq*), costituito da un qualunque espressione indirizzo, in particolare da un identificatore *ind*, consentendo di indirizzare qualunque operando di memoria:**

```
movabsq    $ind,     %rreg
mov/       (%rreg),  /reg
```

Assembler GCC (7)

- **Istruzioni operative con indirizzamento di memoria relativo:**
  - **indicazione Assembler dell'indirizzo dell'operando in memoria:**

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

    - ***ind*** **è un'espressione che individua (nel caso più semplice coincide con) l'identificatore dell'operando**
    - **in fase di traduzione, viene calcolato il valore del campo DISP dell'istruzione macchina come *ind – rip*;**
    - **si può indirizzare un operando solo in una zona limitata a cavallo di RIP**

Assembler GCC (8)

- **Istruzioni di salto con indirizzamento di memoria relativo:**
  - **indicazione Assembler dell'indirizzo di salto:**

    ```
    jmp   ind
    ```

    (RIP è implicito)
    - ***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 come *ind – rip***
    - **si può saltare solo in una zona limitata a cavallo di RIP.**
- **Istruzioni di salto indiretto (solo *jmp* e *call*):**

  ```
  movabsq    $ind,   %rax
  call       *%rax
  ```

  - **il registro può venir caricato con l'istruzione *movabsq*, per cui si può saltare a qualunque indirizzo.**

Modello PIC

- **Programmi Assembler sviluppati nel seguito:**
  - **utilizzazione del modello PIC (*P*osition *I*ndependent *C*ode).**
- **Istruzioni operative:**
  - **operando di memoria individuato da un identificatore:**
    - **utilizzo dell'indirizzamento relativo rispetto a RIP.**
- **Istruzioni di controllo:**
  - **indirizzo di salto individuato da un identificatore:**
    - **utilizzo dell'indirizzamento relativo rispetto a RIP.**
- **Programmi indipendenti dalla posizione:**
  - **tutte le espressioni indirizzo *ind* vengono tradotte come DISP = ind – RIP, per cui DISP rimane costante con lo spostamento di tutto il programma.**

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
```

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* (estensione *o*).**
- **Traduzione:**
  - **avviene attraverso il programma *Assemblatore;***
  - **comando (l'opzione *-o* 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 con l'opzione *–a*:**

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

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 e' *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 con l'opzione *-M*:**

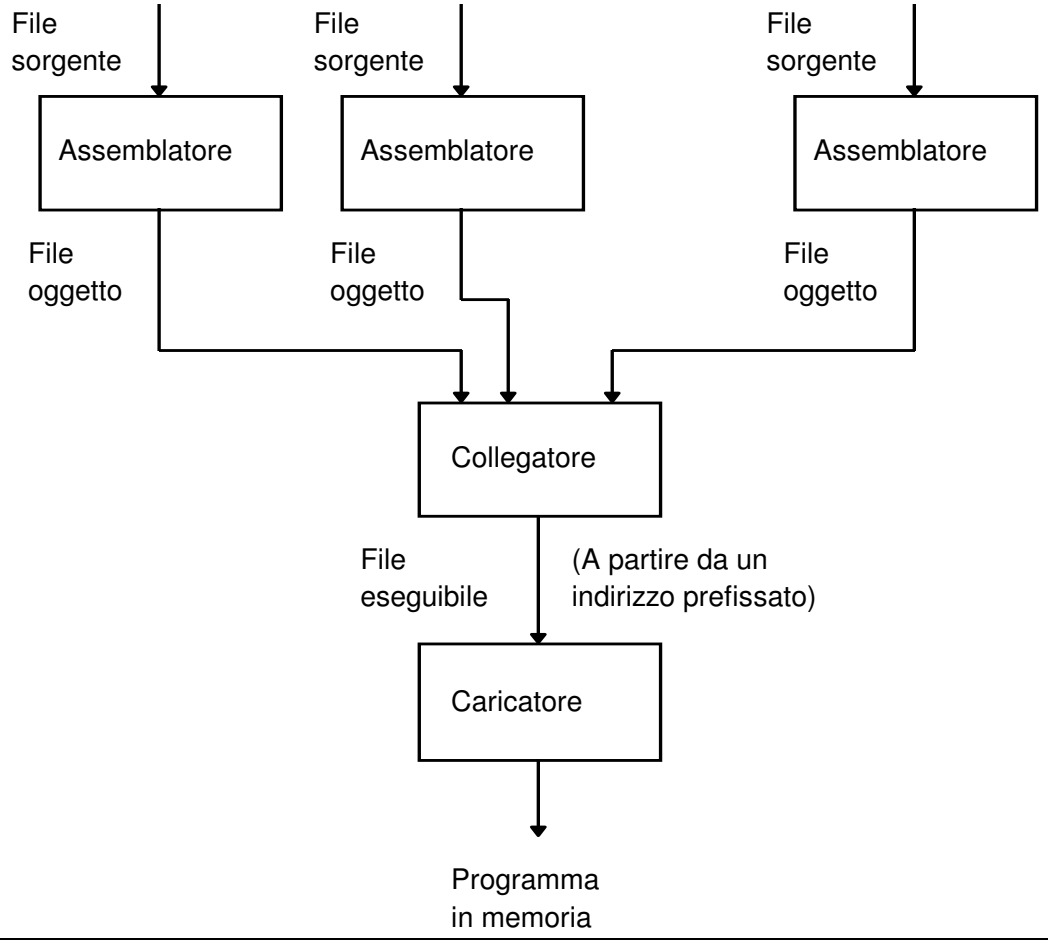
    ld id_file.o -o id_file –M > id_file.m
  - **l'eventuale ridirezione *> id_file.m* fa sì che l'uscita standard del comando *ld* non sia *cout* (comunemente associato al terminale), ma il file specificato.**

Sviluppo di un programma Assembler (3)

- **Rilocazione e Caricamento:**
  - **il programma collegato, eventualmente rilocato se le sezioni .text e .data vengono allocate in zone indipendenti della memoria, poi caricato in memoria, e quindi eseguito;**
  - **comando:**

    ./id_file
  - **la rilocazione non avviene se tutti gli indirizzi di memoria sono relativi a *rip*.**
- **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 IRET.**
- **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



File *ser.s* (1)

- **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.**
- **Contiene 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
         movl    $1, %eax
         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 automaticamente inserito anche il carattere '\r' (ritorno carrello).**

File *ser.s* (2)

- **File *ser.s* (da includere nei programmi Assembler):**

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

- **Sottoprogrammi tastiera e video:**
  - **salvano e ripristinano i registri utilizzati.**

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 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 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, 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 piu' 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 (o pubblici)

- **In un file si possono utilizzare identificatori definiti in altri file:**
  - **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:**
  - **tali identificatori, obbligatoriamente, vanno dichiarati globali;**
  - **identificatori dichiarati globali:**
    - **si utilizza la direttiva *.global* (o *.globl*):**

      .global identificatore, ...

Punto di inizio

- **File sorgente Assembler:**
  - **contiene uno spezzone di programma, che 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 IRET, di inizializzare opportunamente il registro RIP).**
- **In un file, gli identificatori non dichiarati globali sono propri di quel file:**
  - **uno stesso identificatore può essere utilizzato in file diversi e identifica entità diverse.**

Registri

- **Chiamante e registri:**
  - **può lasciarvi informazioni, purché non vengano alterate dal programma chiamato.**
- **Chiamato e registri:**
  - **può utilizzarli per memorizzarvi informazioni, purché non contengano dati utili al chiamante.**
- **Convenzioni:**
  - **in genere, nella realizzazione di programmi su più file, il sistema di sviluppo stabilisce convenzioni sui registri utilizzabili dal chiamante e dal chiamato.**
- **Convenzione *certamente sicura*:**
  - **il chiamante non lascia informazioni nei registri, ma solo in locazioni di memoria, presupponendo che il chiamato utilizzi qualunque registro;**
  - **il chiamato salva e poi ripristina il valore dei registri utilizzati.**
- **Programmi sviluppati nelle slide di questo capitolo:**
  - **utilizzano la convenzione certamente sicura.**

Programma *codifica1* (1)

- **Programma *codifica1*:**
  - **due file: il primo contiene il programma principale, il secondo il 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, 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 codifica1a.s
.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 codifica1b.s
.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++ 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)

- **Uno dei file di libreria collegati:**
  - **contiene l'entry_point *_start* (identificatore esplicitamente dichiarato globale);**
  - **richiama il sottoprogramma *main()* (identificatore esterno);**
    - **il sottoprogramma *main()* produce un risultato intero che lascia in EAX;**
  - **al ritorno dal sottoprogramma *main(),* restituisce il controllo a UNIX con le istruzioni:**

```
movl    %eax, %ebx
movl    $1, %eax
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 del programma principale:**

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

Programma *codifica2* (1)

```
# programma codifica2, file codifica2a.s
.include    "ser.s"
# .extern   alfa, beta, esamina
.data
kappa:      .fill       8, 1

.text
.global     main
main:
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
            movl        $0, %rsi
ripeti:     movb         (%rax , %rsi), %bl
            call        video
            incq        %rsi
            cmpq        $8, %rsi
            jb          ripeti
            movb        $'\n', %bl
            call        video
            jmp         ancora
fine:       movl        $0, %eax
            ret
```

Programma *codifica2* (2)

```
# programma codifica2, file codifica2b.s
# uguale al file secondario codifica1b.s
.data
.global     alfa, beta
alfa:       .byte       0
beta:       .quad       0

.text
.global     esamina
esamina:    pushl       %rax
            …
            ret
```

- **Sviluppo ed esecuzione del programma:**

  g++ codifica2a.s codifica2b.s –o codifica2

  ./codifica2

Programmi C++ su più file

- **Regole di collegamento semplificate:**
  - **in un file, gli identificatori delle variabili definite al di fuori delle funzioni e gli identificatori di funzioni:**
    - **sono implicitamente globali;**
  - **in un file si possono riferire identificatori definiti in altri file, purché:**
    - **siano esplicitamente dichiarati esterni;**
    - **siano globali nei file dove sono definiti.**
- **Dichiarazione di identificatori esterni (simbolo *extern*):**
  - **variabile:**

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

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

    extern type fun(type1 par1,…, typeN parN) { }
  - **per le funzioni (dichiarazione o definizione), il simbolo *extern* può essere omesso.**

Programmi misti (1)

- **Programma GCC organizzato su più file:**
  - **può essere scritto utilizzando linguaggi *differenti* per i vari file (Assembler, C, C++);**
  - **ciascun file viene tradotto con il proprio traduttore, e tutti i traduttori producono file oggetto della stessa forma;**
  - **viene utilizzato un unico Collegatore, che 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 Assembler, avente lo stesso identificatore di quello tradotto e estensione *s;***
  - **singoli file con estensione *s*, compresi quelli generati nei punti precedenti:**
    - **richiama l'Assemblatore, che produce un file avente lo stesso identificatore di quello tradotto e estensione *o*;**
  - **richiama il Collegatore, inserendo anche il file di libreria che contiene l'entry point *_start* e, se inclusi, i file della libreria di I/O.**

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 (in Assembler, sottoprogrammi).**
- **Linguaggio C++ e Assembler:**
  - **i file Assembler (che consentono di avere notevole elasticità) dovranno utilizzare le stesse regole del compilatore C++ (che invece possiede convenzioni rigide) .**

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 (in dichiarazioni o 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 occorre aggiungere la specifica *extern "C";***
  - **pertanto:**
    - **le funzioni esterne C++ vanno dichiarate, prima di essere utilizzate, nel seguente modo:**

      ```
      extern "C" type fun( …);
      ```

    - **le funzioni globali C++ vanno definite nel seguente modo:**

      ```
      extern "C" type fun( … )
      {   // …
      }
      ```

- **Corpi delle funzioni C++:**
  - **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 *int*: 4 byte, in complemento a 2;**
  - **tipo *long int*: 8 byte, in complemento a 2;**
  - **tipo *bool*: un byte, con l'intero 0 equivalente a *false* e l'intero 1 equivalente a *true*;**
  - **tipo *char*: un byte, in codifica ASCII (il bit più significativo vale 0);**
  - **tipi *enumerati*: un tipo *int*, con il valore di ogni enumeratore dato dal suo numero d'ordine, a partire da 0;**
  - **tipi *riferinento*: 8 byte (indirizzo completo di memoria);**
  - **tipi *puntatore*: 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 un tipo 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 più lunghe di 4 byte, allineate a indirizzi multipli di 8.**

Programma misto *codifica3* (1)

```
// programma codifica3, file codifica3a.cpp
using namespace std;
#include <iostream>
extern char alfa;
extern char* beta;
extern "C" void esamina();
char kappa[8];
int main()
{     char al;
      for(;;)
       {        cin.get(al);
                if (al == '\n') break;
                cout << al  << ' ';
                alfa = al; beta = &kappa[0];         // anche beta = kappa;
                esamina();
                for (int i=0; i<8; i++) cout << kappa[i];
                cout << endl;
      };
      return 0;
}
```

Programma misto *codifica3* (2)

```
# programma codifica3, file codifica3b.s, uguale al file codifica2b.s
.data
.global     alfa, beta
alfa:       .byte       0
beta:       .long       0

.text
.global esamina
esamina:    pushl       %eax
            …
            ret
```

Programma misto *codifica4* (1)

```
# programma codifica4, file codifica4a.s, uguale al file codifica2a.s
.include "ser.s"
# .extern   alfa, beta, esamina
.data
kappa:      .fill       8, 1
.text
.global     main
main:
ancora:     call        tastiera
            …
fine:       movl        $0, %eax
            ret
```

Programma misto *codifica4* (2)

```
// programma codifica4, file codifica4b.cpp
char alfa;
char* beta;
extern "C" void  esamina()
{     for(int i = 0; i < 8; i++)
      { if ((alfa & 0x80) == 0)
                *(beta+i) = '0';  else *(beta+i) = '1';  // anche beta[i]
                alfa = alfa<<1;
      }
}
```

Programma *codifica5*

```
// programma codifica5, file codifica5a.cpp: uguale al file codifica3a.cpp
using namespace std;
#include <iostream>
extern char alfa;
extern char* beta;
extern "C" void esamina();
char kappa[8];
int main()
{     …
}

Programma codifica5, file codifica5b.cpp: uguale al file codifica4b.cpp
char alfa;
char* beta;
extern "C" void  esamina()
{     …
}
```

Sviluppo delle versioni miste 3 e 4 e della versione 5

- **Sviluppo della versione 3**

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

  ./codifica3

- **Sviluppo della versione 4:**

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

  ./codifica4

- **Sviluppo della versione 5:**

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

  ./codifica5

Modelli di memoria

- **Parti di un programma:**
  - **come detto sono 4, ossia sezione *.text*, sezione *.data*, *pila*, *heap*;**
  - **la pila e lo heap possono essere memorizzati ovunque (a indirizzi canonici), poiché vengono indirizzati (con un indirizzo assoluto) utilizzando un registro base (la pila viene indirizzata tramite i registri RSP o RBP e la zona per la memoria dinamica tramite puntatori).**
- **Modello grande:**
  - **la sezione *.text* e la sezione *.data* possono essere memorizzate ovunque (ovviamente, a indirizzi canonici).**
- **Modello PIC (*P*osition *I*ndependent *C*ode):**
  - **la sezione *.text* e la sezione *.data* devono essere memorizzate in non più di 2 Gbyte consecutivi di memoria, posizionati ovunque (ovviamente, a indirizzi canonici).**
- **Modello piccolo:**
  - **la sezione *.text* e la sezione *.data* devono essere memorizzate nei primi 2 Gbyte di memoria.**
- **Istruzioni e modelli:**
  - **con il modello grande, viene utilizzata l'istruzioni *movabsq* per caricare un registro base oppure per effettuare un salto indiretto (non condizionato);**
  - **con il modello PIC, per le istruzioni operative con indirizzamento simbolico di operando e per le istruzioni di controllo viene utilizzato l' indirizzamento di memoria relativo;**
  - **con il modello piccolo, per le istruzioni operative può essere utilizzato anche il campo DISP, e per le istruzioni di controllo viene utilizzato l'indirizzamento relativo.**

Modelli di memoria: esempio (1)

- **Programma C++:**

```
int i; int ar[8];
extern "C" void fai()
{    }
int main
{       …
        i = 5;
        ar[i] = 10;
        fai();
        …
}
```

```
# modello grande
# variabili i e ar indirizzate con registro base
# funzione fai chiamata con una call indiretta
.data
i:          .long       0
ar:         .fill       8, 4
.text
.global fai
fai:        …
.global     main
main:       …
            movabsq     $i, %rax
            movl        $5, (%rax)
            movabsq     $ar, %rax
            movabsq     $i, %rbx
            movslq      (%rbx), %rcx
            movl        $10, (%rax, %rcx, 4)
            movabsq      $fai, %rax
            call        *%rax
            …
```

Modelli di memoria: esempio (2)

- **Programma C++:**

```
int i; int ar[8];
extern "C" void fai()
{    }
int main
{       …
        i = 5;
        ar[i] = 10;
        fai();
        …
}
```

```
# modello PIC
# variabili i e ar indirizzate tramite RIP
# funzione fai chiamata con una call tramite RIP
.data
i:          .long       0
ar:         .fill       8, 4
.text
.global fai
fai:        …
.global     main
main:       …
            movl        $5, i(%rip)
            leaq        ar(%rip), %rax
            movslq      i(%rip), %rcx
            movl        $10, (%rax, %rcx, 4)
            call        fai
            …
```

Modelli di memoria: esempio (3)

- **Programma C++:**

```
int i; int ar[8];
extern "C" void fai()
{    }
int main
{       …
        i = 5;
        ar[i] = 10;
        fai();
        …
}
```

```
# modello piccolo
# i e ar indirizzate col solo DISP
# funzione fai chiamata con una call tramite RIP

.data
i:          .long       0
ar:         .fill       8, 4
.text
.global fai
fai:        …
.global     main
main:       …
            movl        $5, i
            leaq        ar, %rax
            movslq      i, %rcx
            movl        $10, (%rax, %rcx, 4)
            call        fai
            …
```

Modello PIC e ambiente GNU

- **Sistema di sviluppo GNU:**
  - **si possono utilizzare i modelli grande e piccolo (senza specifiche, il modello è quello piccolo);**
  - **non viene supportato il modello PIC;**
  - **utilizzando il comando g++ si può specificare che il file eseguibile sia PIC (opzione *-fpic*):**
    - **in questo caso tale file risulta indipendente dalla posizione in cui verrà caricato, senza altri vincoli sulle istruzioni utilizzate.**
- **Programmi riportati nel seguito:**
  - **non vengono specificate opzioni per il modello (il modello e quindi piccolo);**
  - **sono di dimensione modesta, per cui la zona dati e la zona testo sono sicuramente contenute nei primi 2 Gbyte di memoria;**
  - **in Assembler, si utilizza codice PIC.**

Opzioni comuni per il comando g++

- **Comando *g++*:**
  - **Opzioni per il modello di memoria:**
    - ***–mcmodel=large***
    - ***–mcmodel=small***
  - **Opzione per il collegamento dei 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 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.**