Programmazione Assembly

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

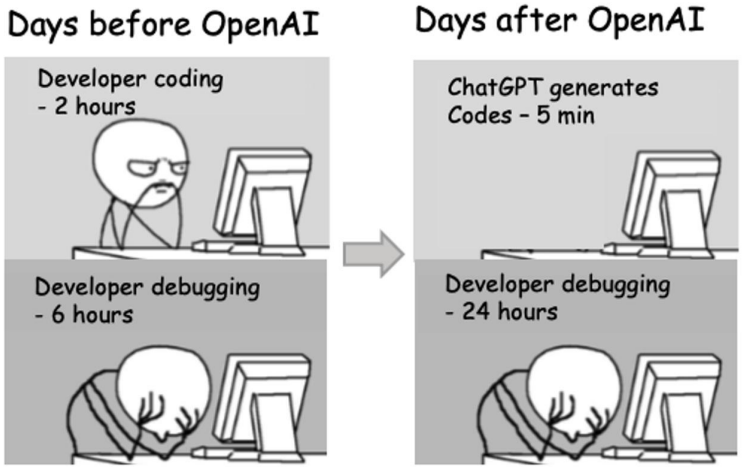
La giusta mentalità

- Avete già usato altri linguaggi in passato?
- Il tipico flusso di lavoro è:
  - Lancia un IDE
  - Butta giù un po' di idee e di istruzioni
  - Lancia qualche interprete e vedi subito i risultati
  - Smanetta sul codice, finché non ottieni quello che volevi (o quasi)

- È una buona mentalità se vogliamo accroccare un programma o sperimentare qualche idea
- Avrete forse già notato che con l'approccio "hack until it works" alla fine non funziona niente

La giusta mentalità

- Il flusso di lavoro “più moderno”:
  - Apri ChatGPT e chiedi di scrivere il codice per te
- Funziona benissimo, fin quando non va tutto malissimo
  - Per poter utilizzare strumenti generativi in maniera corretta, dovete essere in grado *voi stessi* di scrivere correttamente quello che chiedete



La giusta mentalità

- Programmare in assembly sarà più difficile all'inizio, perché richiede di pianificare in anticipo cosa si vuole creare
- È necessario progettare i componenti chiave del programma prima di iniziare ad implementarlo
  - Anche una piccola pianificazione può rendere tutto il processo più liscio
- Non si può essere sciatti nello scrivere, o non funzionerà niente

- Vi mostrerà esattamente cosa possono/non possono fare gli elaboratori
  - Capirete meglio come si programma in C e le relazioni tra il software e l'harwdare
  - Riuscirete a scrivere codice efficiente indipendentemente dal linguaggio

Lo stack software

Lo “stack software”

- Utilizzeremo il C come “assembly portabile”
- Il C è un linguaggio compilato: avete bisogno di un compilatore
- Su Linux:
  - È il sistema più semplice su cui configurare lo sviluppo (in C):
    - La vita è più semplice se usate la riga di comando
  - Su Debian/Ubuntu lanciate:
    - `$ sudo apt-get install build-essential`
  - Su Fedora:
    - `$ sudo yum groupinstall development-tools`
  - Su Arch:
    - `$ sudo pacman -Suy gcc`
  - Su altre distribuzioni:
    - Cercate su Internet “c development tools”
  - Dopo l’installazione, questo comando dovrebbe funzionare:
    - `$ cc --version`
    - Con alta probabilità vi ritroverete ad usare gcc, ma anche CLang va benissimo

Lo “stack software”

- Su MacOs:
  - L’installazione è semplice, anche se dovete tirarvi dentro tantissima roba pressoché inutile
  - Scaricate l’ultima versione di XCode
    - Il download è tipicamente molto pesante e richiede tempo
  - Per confermare che il vostro compilatore C funziona, scrivete in un terminale:
    - `$ cc --version`
  - Dovreste ritrovarvi ad utilizzare una qualche versione del compilatore CLang, ma se avete scaricato una versione più vecchia di XCode vi ritroverete ad usare gcc. Entrambi vanno bene.

- Se avete processori M1/M2, dovete abilitare la modalità di compatibilità con l’architettura x86:

```
arch -x86_64 /bin/bash
```

Lo “stack software”

- Su Windows:
  - Il compilatore Microsoft installato con Visual Studio è strapieno di sovrastrutture inutili per questo corso e non implementa una versione del C sensata
  - La soluzione più “semplice” è scaricare e installare Cygwin
    - https://www.cygwin.com/
    - Otterrete anche molti strumenti di sviluppo Posix
  - Un’alternativa è il sistema MinGW:
    - http://www.mingw.org/
    - Più minimalista, ma funzionerà comunque
  - Un’opzione più avanzata: scaricate Virtual Box e installate una Virtual Machine con Linux
  - Un’opzione ancora più avanzata: usate il Windows Subsystem for Linux (WSL) versione 2 su Windows 10/11 per lanciare Ubuntu in Windows
  - Un’opzione drastica: create una nuova partizione e iniziate a usare Linux
  - L’opzione più a “lungo termine”: formattate e installate Linux

L’Editor

ATTENZIONE: Evitate di usare qualsiasi Integrate Development Environment (IDE) mentre state imparando un linguaggio nuovo. Sono utili per velocizzare il lavoro, ma il loro aiuto in genere vi impedisce di imparare davvero il linguaggio.

- Varie opzioni possibili:
  - GEdit su Linux e MacOS
  - TextWrangler su MacOS
  - Notepad++ su Windows
  - Nano, che funziona nel terminale e si trova più o meno ovunque
  - Vim, che ha una certa curva di apprendimento
  - Emacs, che ha una curva di apprendimento più ripida
- Esiste l’editor perfetto per ciascuna persona nel mondo
- Non sprecate adesso troppo tempo a trovare ora quello perfetto per voi

Debugger

- Su Linux/Cygwin: possiamo usare GDB
  - Su Debian/Ubuntu:
    - `$ sudo apt-get install gdb`
  - Su Fedora:
    - `$ yum install gdb`
  - Su Arch:
    - `$ sudo pacman -Suy gdb`
- Su MacOs, GDB è rimpiazzato da LLDB
  - Se avete installato la toolchain di compilazione tramite XCode, lo troverete già installato

Ulteriori strumenti

- `objdump`: è un software che permette di disassemblare un eseguibile
  - permette di "ricostruire" il codice assembly a partire da programmi compilati (`objdump -d nomefile`)
  - molto utile per ispezionare cosa farà davvero un'applicazione
- `valgrind`: una macchina virtuale per individuare errori nell'uso della memoria
  - esegue il codice in un ambiente virtualizzato
  - effettua controlli sull'accesso a memoria
  - facilita l'individuazione di *undefined behaviour* legati all'uso della memoria
- `strings`: estrae da un eseguibile tutte le stringhe presenti e le mostra a schermo

Lo “stack software”

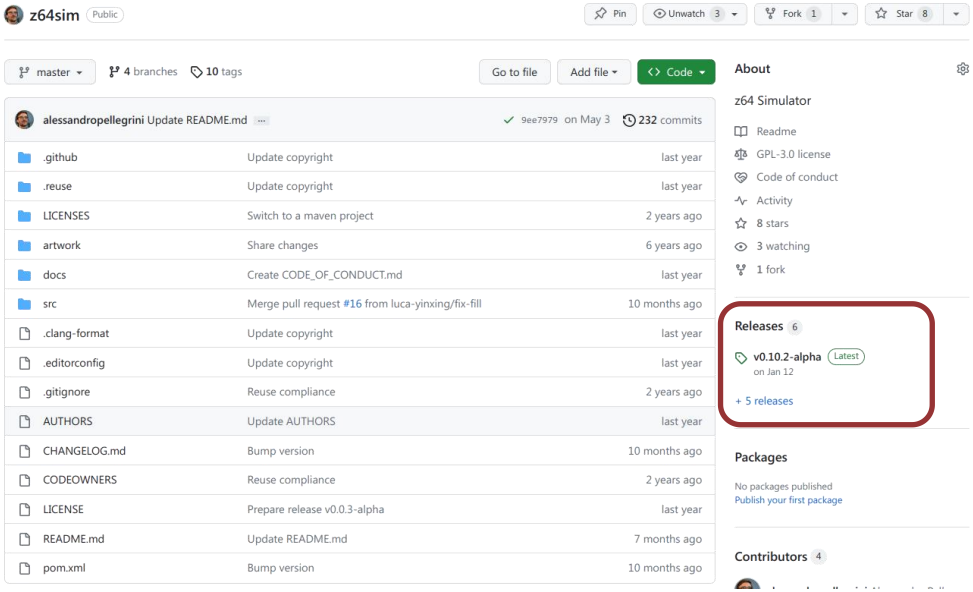
- z4sim: il simulatore del processore z64

  https://github.com/alessandropellegrini/z64sim

- Implementato in Java, funziona su tutti i sistemi

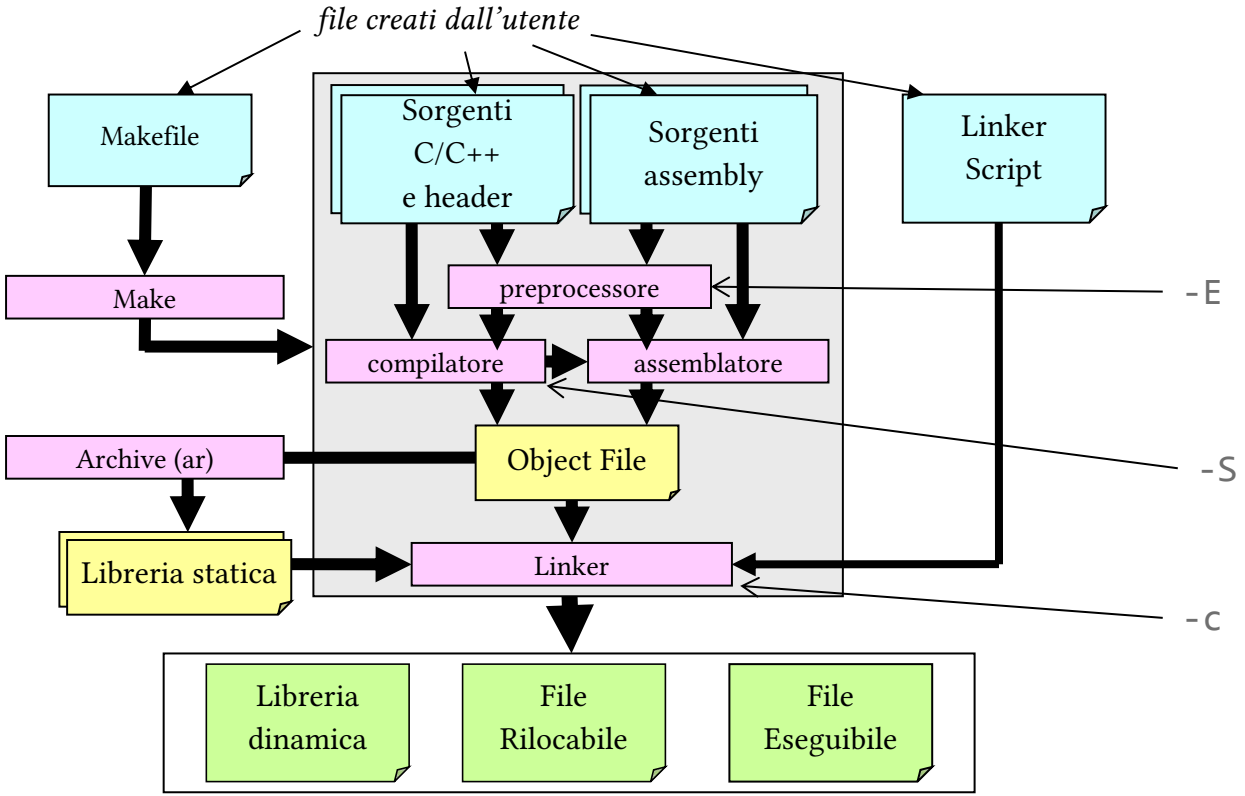
- Le istruzioni per l’uso (poche!) sono sulla pagina del progetto

- È un progetto in versione alpha!



Riscaldamento

Il processo di compilazione



Hello World!

```
1.  #include <stdio.h>
2.
3.  /* This is a comment. */
4.  int main(int argc, char *argv[])
5.  {
6.      int times = 100;
7.
8.      // this is also a comment
9.      printf("Hello World! I welcome you %d times.\n", times);
10.     return 0;
11. }
```

Chiediamo al preprocessore di importare un altro file

Un commento multilinea

I vostri programmi partono sempre dalla funzione `main`. Il sistema operativo carica il programma e lo lancia da qui. Per funzionare, deve restituire un `int` ed accettare due parametri: un `int` per il numero di parametri e un array di stringhe `char *` per i parametri

Inizio di un blocco

Dichiarazione e assegnazione di variabile locale

Un commento a linea singola

Chiamata a funzione. È una funzione strana, con un numero arbitrario di parametri

Valore di ritorno della funzione. In questo caso è il valore di ritorno al sistema operativo

La fine di un blocco

Notate che tutte le asserzioni terminano con un ';' in C!

Utilizzo di un debugger

- I debugger ci consentono di controllare l'esecuzione di un programma, interrompendo l'esecuzione in maniera selettiva
- Quando l'esecuzione è interrotta, ci consentono di osservare lo stato della memoria, lo stato dei registri, il codice assembly e, in taluni casi, anche il codice sorgente originale
- Possiamo anche modificare lo stato di un programma (cambiando, ad esempio, il contenuto di una variabile) prima di riprendere l'esecuzione
- Sono strumenti fondamentali, ma hanno una curva di apprendimento ripida
  - Iniziate ad utilizzare subito il debugger che avete installato in precedenza!

GDB Cheatsheet (LLDB è praticamente uguale)

- Compilate aggiungendo l'opzione `-g`
- Per lanciare il debugger: `gdb --args ./program [args]`
- Comandi principali:
  - `run [args]`: avvia il programma passando gli argomenti `[args]`.
  - `break [file:]function`: imposta un breakpoint.
  - `backtrace`: mostra un backtrace delle funzioni chiamate fin'ora.
  - `print: expr`: mostra il valore di `expr`.
  - `continue`: continua l'esecuzione del programma.
  - `next`: vai alla prossima riga nel sorgente, ma non entrare nelle funzioni.
  - `step`: vai alla prossima riga nel sorgente, entrando nelle funzioni.
  - `step instruction`: esegui una singola istruzione assembly
  - `quit`: termina l'esecuzione del debugger.
- Modalità interattiva: `CTRL-x, a` (o `--tui`)
  - `layout regs`: mostra una finestra con il contenuto dei registri
  - `layout asm`: mostra il sorgente assembly
  - `layout src`: mostra il sorgente C
  - `focus [what]`: sposta il focus su una particolare finestra

Hello World: Anatomia del programma

```
1. #include <stdio.h>
2.
3. /* This is a comment. */
4. int main(int argc, char *argv[])
5. {
6.     int times = 100;
7.
8.     // this is also a comment
9.     printf("Hello World! I
         welcome you %d times.\n",
         times);
10.    return 0;
11. }
```

```
.data
.LC0:
        .ascii  "Hello World! I welcome
                   you %d times.\n"

.text
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $8, %rsp
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %esi
        leaq    .LC0(%rip), %rdi
        xorl    %eax, %eax
        call    printf@PLT
        xorl    %eax, %eax
        addq    $8, %rsp
        leave
        ret
```

Hello World: Anatomia del programma

```
.data
.LC0:
        .ascii  "Hello World! I welcome you %d times.\n"

.text
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $8, %rsp
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %esi
        leaq    .LC0(%rip), %rdi
        xorl    %eax, %eax
        call    printf@PLT
        xorl    %eax, %eax
        addq    $8, %rsp
        leave
        ret
```

- direttive assembly
- etichette
- variabili globali
- opcode
- registri destinazione
- registri sorgente
- costanti
- operandi in memoria

Scheletro di un programma assembly z64

```
.org [INDIRIZZO CARICAMENTO]
.data
        # Dichiarazione costanti e variabili globali

.text
main:
        # Corpo del programma
         hlt # Per arrestare l'esecuzione
```

Direttive assembly

- *label*: mnemonico testuale definito dal programmatore ed associato all'indirizzo di ciò che la segue immediatamente
- *Location Counter*: identificato da `.`, viene valutato con il valore dell'indirizzo corrente.
  - Può essere impostato esplicitamente per far "saltare" la generazione di indirizzi.
  - Può essere usato per calcolare le dimensioni di strutture dati:

```
msg:
        .ascii "Hello, world!\\n"
        len = . - msg
```

- `.org address, fill`: metodo alternativo di impostare il location counter, impostando i byte a fill

Direttive assembly

- `.equ symbol, expression`: definisce una costante (non occupa memoria al momento della dichiarazione)
  - Metodo alternativo: `symbol = expression`
  - Lo stesso simbolo può essere ridefinito in più parti del codice
  - Non si può usare il simbolo prima della sua definizione (*one pass scan*)
- `.byte expressions`: riserva memoria (di dimensione byte) per `expressions`:

```
var: .byte 0
array: .byte 0, 1, 2, 3, 4, 5
```

- `.word expressions`: riserva memoria (di dimensione word) per `expressions`
- `.long expressions`: riserva memoria (di dimensione longword) per `expressions`
- `.quad expressions`: riserva memoria (di dimensione quadword) per `expressions`

Direttive assembly

- `.ascii` “`string`”: riserva memoria per un vettore di caratteri e imposta il valore a string
- `.fill repeat, size, value`: riserva una regione di memoria composta da `repeat` celle di dimensione `size` impostate a `value`
  - `size` e `value` sono opzionali (default: `size` = 1, `value` = 0).
- `.text:` tutto ciò che compare da qui in poi va nella sezione testo
- `.data`: tutto ciò che compare da qui in poi va nella sezione data
- `.comm symbol, length`: dichiara un’area di memoria con nome (`symbol`) di dimensione `length` nella sezione `.bss`
- `.driver ivn`: identifica l’inizio della routine di servizio associato al codice `ivn`

Strutture di controllo

Controllo di condizione

```
if(x == 1) {
    // CODE BLOCK A;
} else if(x == 2) {
    // CODE BLOCK B;
} else {
    // CODE BLOCK C;
}
```

```
    cmpb $1, %al
    jnz .elseif
    # CODE BLOCK A
    jmp .endif
.elseif:
    cmpb $2, %al
    jnz .else
    # CODE BLOCK B
    jmp .endif
.else:
    # CODE BLOCK C
.endif:
```

Aggiornamento del Carry Flag

- L'operazione $a - b$ richiede un prestito quando b è maggiore di a.
- Nel caso di sottrazione, la CU rileva la necessità di prestito verificando se l'addizione in complemento a 2 corrispondente determina un riporto
- Se nell'addizione non c'è un riporto, allora c'è un prestito nella sottrazione
- Se c'è un riporto nell'addizione, allora non c'è un prestito nella sottrazione

- Nel caso di esecuzione di `sub`, CF viene quindi negato

Confronti in aritmetica non segnata

- L'istruzione `cmp` effettua una sottrazione tra la destinazione e la sorgente, scartando il risultato
- I bit di FLAGS vengono però aggiornati
- Si possono usare CF e ZF flag per inferire relazioni tra gli operandi

Condizione	Primo controllo	Secondo controllo
`dest < source`	CF = 1	
`dest ≥ source`	CF = 0	
`dest > source`	CF = 0	ZF = 0
`dest = source`	ZF = 1	
`dest ≠ source`	ZF = 0	

Confronti in aritmetica segnata

- CF non ha alcun significato nel caso di operazioni in complemento a due: l'overflow si verifica confrontando gli ultimi due riporti
- Per effettuare confronti, effettuiamo una sottrazione (`cmp`): ci chiediamo se è vero che

```
dest - src < 0
```

- Il risultato di questa disequazione è dato dal flag di segno SF
- MA, se c'è stato un overflow, allora il segno è cambiato
- Il bit OF è calcolato come lo xor degli ultimi due riporti: determina se c'è stato un overflow
  - Se non si è verificato overflow, la disuguaglianza è verificata se SF=1.
  - Se si è verificato un overflow, la disuguaglianza è verificata se SF=0.

Confronti in aritmetica segnata

Condizione	Primo controllo	Secondo controllo
`dest < source`	SF ≠ OF	
`dest ≥ source`	SF = OF	
`dest > source`	ZF = 0	SF = OF
`dest = source`	ZF = 1	
`dest ≠ source`	ZF = 0	

Confronti in aritmetica segnata

```
.org 0x800
.data
        x: .word 3
        y: .word -2
.text
        # Imposta a 1 l'indirizzo 0x1280 solo se x > y
        # Assumo che x ed y possano assumere valori negativi
        movw x, %ax
        movw y, %bx
        cmpw %bx, %ax
        jz .nonImpostare
        js .SFset
        jo .nonImpostare # eseguita se SF = 0. Se OF = 1 allora SF != OF
        jmp .set
        .SFset:
        jno .nonImpostare # eseguita se SF = 1. Se OF = 0 allora Sf != OF
        .set:
        movb $1, 0x1280
        .nonImpostare:
        hlt
```

Pseudo-operazioni

- Alcuni costrutti assembly possono essere complessi o ripetitivi
- L'assemblatore può fornire *pseudo-operazioni* per semplificare il processo di sviluppo
- Sono operazioni non realmente implementate in hardware
- L'assemblatore le sostituisce con blocchi di codice equivalenti

- Lo z64 implementa come pseudo-operazioni le istruzioni x86 per i confronti in aritmetica segnata e non segnata

Pseudo-operazioni per confronti aritmetici

Istruzione	Descrizione	Aritmetica	Condizione controllata
`jb`	Jump if below	non segnata	CF = 1
`jnae`	Jump if not above or equal		
`jnb`	Jump if not below	non segnata	CF = 0
`jae`	Jump if above or equal		
`jbe`	Jump if below or equal	non segnata	CF = 1 o ZF = 1
`jna`	Jump if not above		
`ja`	Jump if above	non segnata	CF = 0 e ZF = 0
`jnbe`	Jump if not below or equal		
`jl`	Jump if less	segnata	SF ≠ OF
`jnge`	Jump if not greater or equal		
`jge`	Jump if greater or equal	segnata	SF = OF
`jnl`	Jump if not less		
`jle`	Jump if less or equal	segnata	ZF = 1 o SF ≠ OF
`jng`	Jump if not greater		
`jg`	Jump if greater	segnata	ZF = 0 e SF = OF
`jnle`	Jump if not less or equal		

Switch case

```
switch (var) {
   case 1:
      // CODE;
      break;
   case 2:
      // CODE;
      break;
   case 3:
      // CODE;
      break;
   case 4:
      // CODE;
      break;
   default:
      // CODE;
}
```

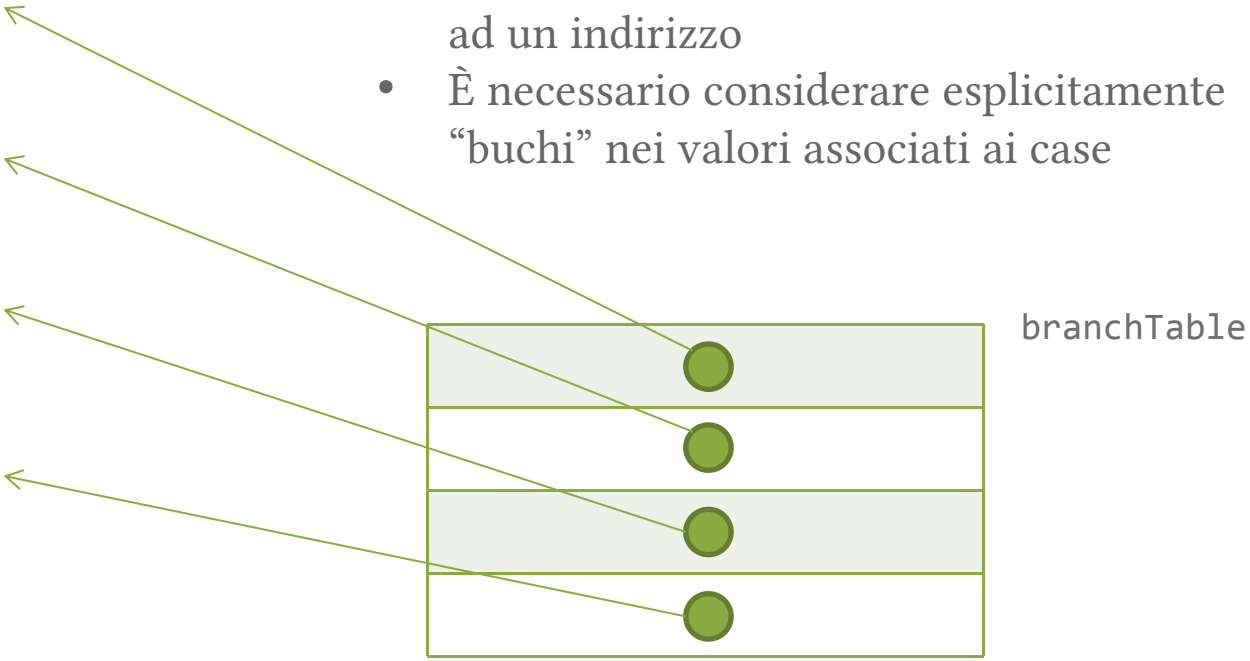
```
   cmpl    $5, var
   jnc      .default
   cmpl    $4, var
   jz      .case4
   cmpl    $3, var
   jz      .case3
   cmpl    $2, var
   jz      .case2
   # case 1
   jmp .end
.case2:
   # CODE
   jmp     .end
.case3:
   # CODE
   jmp     .end
.case4:
   # CODE
   jmp     .end
.default:
   # CODE
.end:
```

- La lunghezza del codice è elevata
- È necessario fare un confronto per tutti i valori di ciascun case
- Si può fare di meglio?

Switch case

```
switch (var) {
   case 1:
      // CODE;
      break;
   case 2:
      // CODE;
      break;
   case 3:
      // CODE;
      break;
   case 4:
      // CODE;
      break;
   default:
      // CODE;
}
```

- Si possono utilizzare *tabelle di salto*
  - Vettori in memoria che associano un “case” ad un indirizzo
  - È necessario considerare esplicitamente “buchi” nei valori associati ai case



Switch case

```
switch (var) {
   case 1:
      // CODE;
      break;
   case 2:
      // CODE;
      break;
   case 3:
      // CODE;
      break;
   case 4:
      // CODE;
      break;
   default:
      // CODE;
}
```

```
   cmpl $5, var
   jnc .default
   movzlq var, %rax
   shll $3, %rax # Indirizzi a 64 bit
   movq branchTable(%rax), %rax
   jmp *%rax
   ...
.default:
```

Salti e chiamate a funzioni

```
if(ERROR_TEST) {
  goto fail;
}
// CODE;
fail:
  // CODE;
```

- Lo statement **goto** equivale ad un'istruzione `jmp` o `jX` (es: `jc`, `jnz`, `js`, ...)

```
cmp ...
jX .fail
...
.fail:
```

```
int f(int arg1)
{
   // CODE;
   return 0;
}

f(2);
```

- L'attivazione delle funzioni corrisponde a un'istruzione `call`

```
call f
```

- `f` è codificato come spiazzamento da RIP (dopo la fase di fetch)

Cicli

```
while(TEST) {
    CODE;
}
```

```
.test:
    cmpb ...
    jnz .skip
    # <codice>
    jmp .test
.skip:
```

```
do {
    CODE;
} while(TEST);
```

```
.begin:
    # <codice>
    cmpb ...
    jz .begin
```

```
for(INIT; TEST; POST) {
    CODE;
}
```

```
movq $0, %rcx
movq $3, %rbx
.test: cmpq %rbx, %rcx
    jz .end
    # <codice>
    addq $1, %rcx
    jmp .test
.end:
```

- Come implementare `break` e `continue`?
- Si possono utilizzare delle istruzioni `jmp`

Tipi di dato

Variabili in C/Assembly

- Ogni variabile, in C, **ha un tipo**
  - Differente dal concetto di “duck typing”, ad esempio in Python: *Se parla e si comporta come una papera, allora è una papera*
  - Perché?

- Le variabili in C possono essere raggruppate in tre tipologie:
  - Tipi primitivi (interi, virgola mobile, ...)
  - Tipi aggregati (strutture, unioni)
  - Puntatori
- I tipi primitivi ed i puntatori sono gli unici tipi di variabili che hanno un corrispettivo in istruzioni assembly
- I tipi aggregati vengono automaticamente convertiti dal compilatore in accesso a tipi primitivi, utilizzando laddove possibile le modalità di indirizzamento

Ambito delle variabili

- Ogni variabile dichiarata nel programma ha un certo *ambito* (scope), che determina la *visibilità* della variabile a determinate porzioni del programma:
  - variabili **globali**: occupano memoria all'interno delle sezioni `.data` e `.bss`: visibili da tutte le funzioni
  - variabili **locali** (o *automatiche*): occupano memoria all'interno dello stack: visibili alla funzione stessa (o a funzioni chiamate, se passate tramite puntatori)

- Dal punto di vista del processore, ogni accesso a variabile viene rappresentato da un accesso all'indirizzo di memoria associato alla variabile
- Le variabili locali sono "valide" fino a che l'area di stack in cui sono memorizzate è valida

La finestra di stack

- Le variabili automatiche di una funzione occupano memoria all'interno dello stack
- È possibile distinguere il "contesto" di esecuzione di una funzione poiché all'ingresso viene automaticamente creato una *finestra di stack* (o *record di attivazione*)
- Durante la creazione, viene riservato spazio per le variabili automatiche
- Il contesto contiene anche l'indirizzo di ritorno all'istruzione successiva al salto a funzione
- Al termine della funzione, il record di attivazione viene *invalidato logicamente*
  - Il suo contenuto permane in memoria fino a successiva sovrascrittura!

La finestra di stack (o record di attivazione)

```
void function() {
    int x = 128;
    ...
    return;
}
```

```
function:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $4, %rsp
        movl    $128, -4(%rbp)
        ...
        leave
        ret
```

- L'istruzione x86 `leave` invalida la finestra di stack corrente
- Corrisponde a: `movl %rbp, %rsp; popl %rbp`
- Esiste anche l'istruzione `enter`, ma non è efficiente
- I debugger usano le finestre di stack per ricostruire le chiamate effettuate

Passaggio di parametri: convenzioni di chiamata

- Affinché una subroutine chiamante possa correttamente dialogare con la subroutine chiamata, occorre mettersi d'accordo su come passare i parametri ed il valore di ritorno
- Le *calling conventions* definiscono, *per ogni architettura e sistema*, come è opportuno passare i parametri
- Le convenzioni principali permettono di passare i parametri tramite:
  - lo stack
  - i registri
  - un misto delle due tecniche
- Generalmente il valore di ritorno viene passato in un registro perché la finestra di stack viene distrutta al termine della subroutine
  - Se la subroutine chiamante vuole conservare il valore nel registro, deve memorizzarlo nello stack prima di eseguire la chiamata

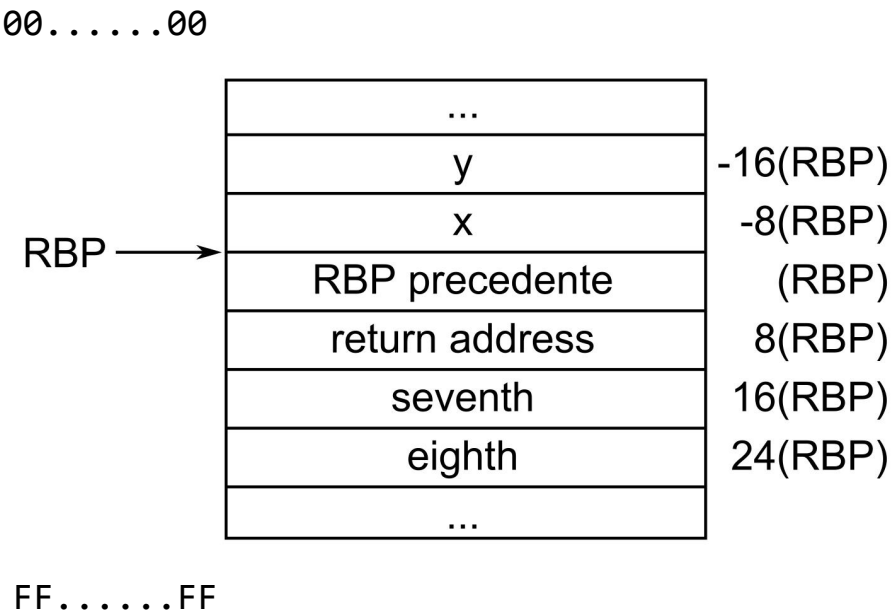
Calling convention z64/x86

- I primi sei parametri (interi e indirizzi) di una subroutine vengono passati tramite registri:
  - RDI, RSI, RDX, RCX, R8, R9
- Se una subroutine accetta più di sei parametri, si utilizza lo stack per quelli aggiuntivi
- Si utilizzano due registri per il valore di ritorno:
  - RAX e RDX

- I registri sono divisi in *callee save* e *caller save*
  - callee-save: RBP, RBX, R12–R15
  - caller-save: tutti gli altri

Anatomia dello stack

```
void f(int first, int second, int third, int fourth, int fifth,
       int sixth, int seventh, int eighth,) {
       int x, y;
       ...
}
```



Tipi primitivi

- In C, ci sono soltanto 4 tipi *primitivi*:
  - `char`, `int`, `float`, `double`
- Questi tipi possono essere *alterati* (in termini di rappresentazione e dimensione) utilizzando dei *modificatori*:
  - `signed`, `unsigned`, `short`, `long`

- Questi tipi vengono associati ai "tipi" che l'ISA è in grado di gestire (non c'è una dimensione standard per tutte le architetture)
- I modificatori `signed`, `unsigned` indicano al compilatore se devono essere utilizzate istruzioni aritmetiche per lavorare o meno in complemento a 2 e quali sono i bit di FLAGS da controllare per effettuare salti condizionali

Tipi primitivi

- Le seguenti corrispondenze tra tipi primitivi C e dimensioni dei dati gestiti dal processore sono validi per l'architettura x86/z64

Tipo C	Dimensione del dato
`char`	byte
`short int`	word
`int`	longword
`long int`	quadword

Cast: conversione tra tipi

- L'operazione di *cast* converte un dato da un tipo all'altro:

```
(tipo di destinazione)variabile;
```

- Ad esempio:

```
int integer = 10;
double real = (double)integer;
```

- La conversione tra tipi interi viene realizzata mediante le istruzioni `movsX` e `movzX` oppure troncando la parte più significativa

Il tipo booleano

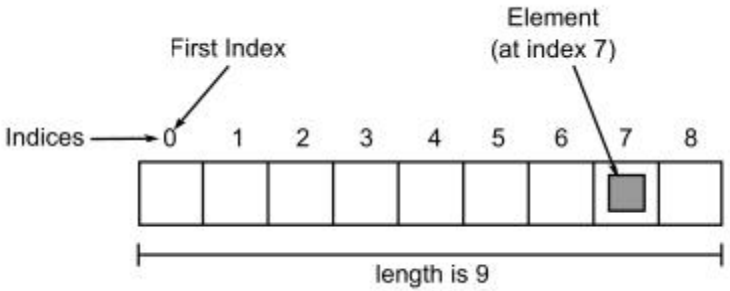
- Dal punto di vista architetturale, il tipo “booleano” non esiste
  - Non esiste alcuna istruzione in nessuna ISA che gestisca i booleani
  - Per convenzione: 0 è `false`, un valore diverso da 0 è `true`

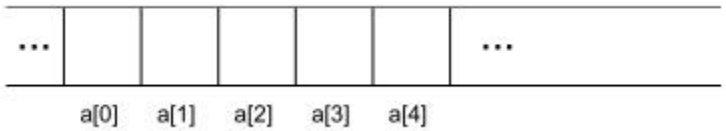
```
char booleano = 1;
if(booleano)
    printf(“true\n”);
```

- Il controllo sui booleani può essere realizzato con le istruzioni `jz`/`jnz`.

Array

- Un array (impropriamente tradotto con *vettore*) è una *collezione di elementi*, ciascuno identificato da un *indice*
- In memoria, gli elementi del vettore sono conservati in maniera contigua
- L'indirizzo dell'elemento cui si cerca di accedere viene calcolato con una *trasformazione lineare* dell'indice
  - ciascun elemento ha una dimensione differente in funzione del suo *tipo*





Manipolazione di array in assembly

- Le modalità di indirizzamento in memoria consentono di scandire facilmente array di tipi primitivi
- Si può utilizzare la base o lo spiazzamento per individuare l'inizio dell'array in memoria
- L'indice e la scala possono essere usati per calcolare l'indirizzo di un elemento ben preciso

```
    movq $0, %rcx # %rcx viene usato come indice
    movq $array, %rax # %rax viene usato come base
.loop:
    movq (%rax, %rcx, 8), %rdx # carica il dato nella CPU
    # <processamento>
    addq $1, %rcx
    cmpq $size, %rcx
    jnz .loop
```

Manipolazione di array in assembly

- Le modalità di indirizzamento in memoria consentono di scandire facilmente array di tipi primitivi
- Si può utilizzare la base o lo spiazzamento per individuare l'inizio dell'array in memoria
- L'indice e la scala possono essere usati per calcolare l'indirizzo di un elemento ben preciso

```
    movq $0, %rcx # %rcx viene usato come indice
.loop:
    movq array(, %rcx, 8), %rdx # carica il dato nella CPU
    # <processa i dati>
    addq $1, %rcx
    cmpq $size, %rcx
    jnz .loop
```

Manipolazione di array in assembly

- Se il vettore non è composto da tipi primitivi, non si può usare la scala
- In questo caso, si utilizza un registro per puntare all'area di memoria con l'elemento successivo
  - Di fatto, stiamo usando un puntatore
- Tale registro deve essere incrementato manualmente

```
    movq $array, %rax # carica indirizzo del primo elemento
    movq $size, %rcx
.loop:
    movq (%rax), %rdx # carica il dato nella CPU
    # <processa i dati>
    addq $element_size, %rax
    subq $1, %rcx
    jnz .loop
```

Stringhe

- Le stringhe sono rappresentate in memoria come un vettore di caratteri
- È necessario utilizzare un carattere speciale per identificare il punto in cui la stringa termina in memoria
- Per questo scopo si utilizza un valore speciale nella codifica ASCII: il *terminatore di* stringa o `NUL` (pari al byte zero, indicato con il carattere `'\0'`)

- Data questa rappresentazione in memoria, in C le stringhe sono dei vettori di `char`, il cui ultimo elemento è il terminatore di stringa.
- Possono essere manipolate come array

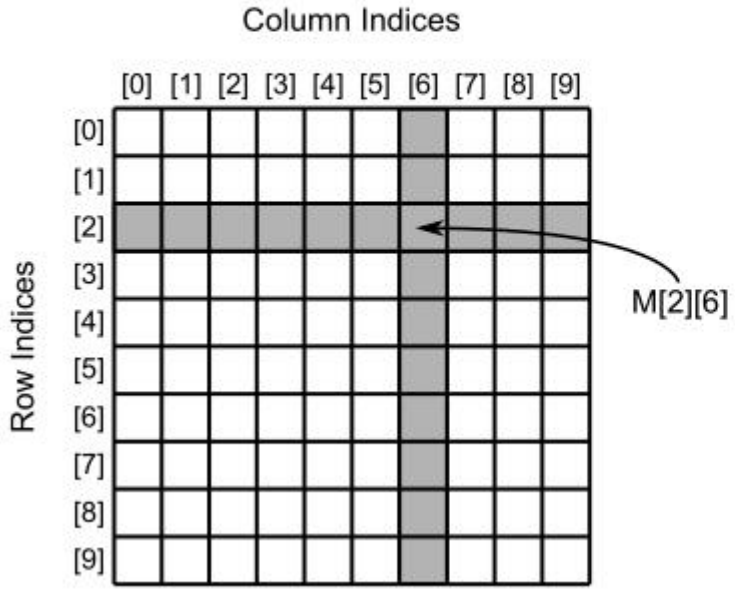
Matrici

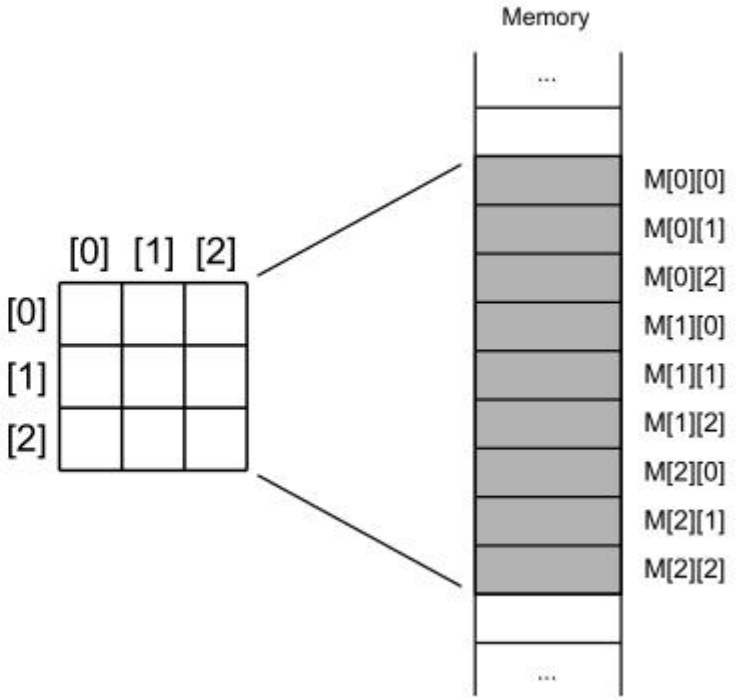
- Le matrici sono strutture dati a più dimensioni
- Tuttavia, il modello di memoria è lineare
- Pertanto, le matrici vengono “linearizzate” in memoria
- Il comportamento dell’operatore `[]` è differente nel caso delle matrici:
  - Quando si accede utilizzando un doppio spiazzamento `[i][j]`, questo spiazzamento viene linearizzato nella forma:

$$\texttt{address} \Leftrightarrow \texttt{i} * \texttt{N} * \texttt{size} + \texttt{j} * \texttt{size}$$

- Ciò è possibile solo se le dimensioni delle matrici sono note a tempo di compilazione!
- Per estensione, lo stesso funzionamento è valido per le matrici *n*-dimensionali

Matrici


Column Indices
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
Row Indices
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
M[2][6]


Memory
...
[0] [1] [2]
[0]
[1]
[2]
M[0][0]
M[0][1]
M[0][2]
M[1][0]
M[1][1]
M[1][2]
M[2][0]
M[2][1]
M[2][2]
...

Puntatori

- Ciascun tipo `T` ha il corrispondente tipo *puntatore a T*.
- Un puntatore è un tipo di dato che contiene l'*indirizzo* dell'area di memoria che contiene una variabile di quel tipo.
- Un puntatore differisce dalla variabile di quel tipo poiché si utilizza il *dichiaratore di tipo asterisco* (*) tra il tipo e il nome della variabile:

```
int *intptr;
```

- Attenzione: in C l'asterisco modifica la variabile, non il tipo!

```
int *var1, var2;
```

- Esiste un generico puntatore a memoria: `void *`

Puntatori: equivalente in assembly

- L'utilizzo dei puntatori in assembly è immediato
  - L'indirizzo dell'area da puntare viene scritto in un registro
  - Si utilizza quel registro come registro base
  - Il contenuto del registro può essere manipolato a piacimento

```
int var;
int *ptr = &var;

*ptr = 42;

ptr++;
```

```
movq $var, ptr

movq ptr, %rax
movl $42, (%rax)

addq $4, %rax
```

Attenzione ai puntatori ed all'ambito delle variabili!

file: doomed-pointer.c

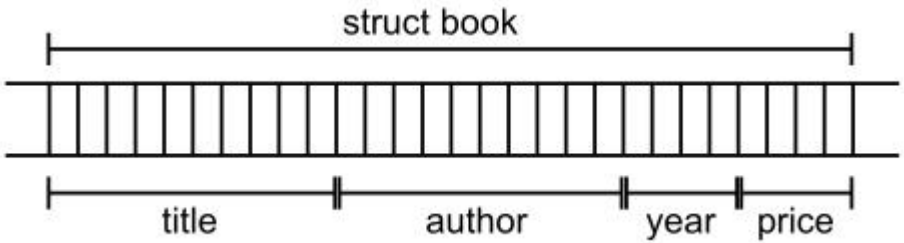
```c
#include <stdio.h>

int *buggy_return(void)
{
        int a_variable = 10;
        return &a_variable;
}

void using_stack(void)
{
        char a_string[] = "Hello World!";
        printf("%s\n", a_string);
}
```

```c
int main(int argc, char *argv[])
{
        int *dummy = buggy_return();
        printf("%d\n", *dummy);
        using_stack();
        printf("%d\n", *dummy);
        return 0;
}
```

Strutture

- Le strutture sono un modo per *aggregare* in maniera logica tipi primitivi, altre strutture, o vettori.

```
struct book {
    char title[10];
    char author[10];
    int publication_year;
    float price;
};
```



Strutture in assembly

- Per accedere ai membri di una struttura in assembly, si può utilizzare la modalità di indirizzamento generale
- Si calcola uno spiazzamento a partire dalla base della struct
- Si carica in un registro l'indirizzo iniziale della struct
- Si utilizza un operando in memoria composto da spiazzamento e base
- Attenzione al *padding* se si interagisce con codice generato da un compilatore

```
struct book {
    char title[10];
    char author[10];
    int publication_year;
    float price;
};

struct book b;
b.publication_year = 1987;
```

```
movq $b, %rax
movl $1987, 20(%rax)
```

Passaggio di parametri di tipo struttura

- In C, il passaggio di parametri è sempre fatto *per valore*
  - La funzione riceve *una copia* del parametro

- Nel caso di struct, questo vuol dire che viene effettuata una copia di tutta la struttura
  - il costo computazionale della chiamata *aumenta drasticamente*
  - se non si hanno problemi di *side effect*, è opportuno evitarlo

- È possibile utilizzare i puntatori per effettuare un *passaggio per riferimento* (o per indirizzo)

Passaggio di parametri

```
struct huge {
    char member[4096];
};

extern void f1(struct huge s);
extern void f2(struct huge *s);

struct huge glbl;

void f(void) {
    f2(&glbl);
}
```

```
pushq   %rbp
movq    %rsp, %rbp
movq    $glbl, %rdi
call    f2
popq    %rbp
ret
```

Passaggio di parametri

```
struct huge {
    char member[4096];
};

extern void f1(struct huge s);
extern void f2(struct huge *s);

struct huge glbl;

void f(void) {
    f1(glbl);
}
```

```
pushq   %rbp
movq    %rsp, %rbp
subq    $4096, %rsp
movq    %rsp, %rdi
movq    $glbl, %rsi
movq    $512, %rcx
cld
rep movsq
call    f1
addq    $4096, %rsp
leave
ret
```

`movs` e `stos`

- Si tratta di istruzioni per operare su stringhe (buffer di dati)
- Utilizzano dei registri impliciti:
  - RCX: contatore del numero di operazioni elementari da eseguire
  - RSI: indirizzo sorgente (per il movimento)
  - RDI: indirizzo destinazione
  - RAX: valore cui impostare la memoria (per l'impostazione)
- Il direction flag (DF) identifica la direzione dell'operazione:
  - DF = 0: l'operazione di copia si svolge in avanti
  - DF = 1: l'operazione di copia si svolge all'indietro

movs e stos

`movs`: move data from string to string

```
movq $source, %rsi
movq $destination, %rdi
movq $size/8, %rcx
cld
movsq
```

`stos`: store string

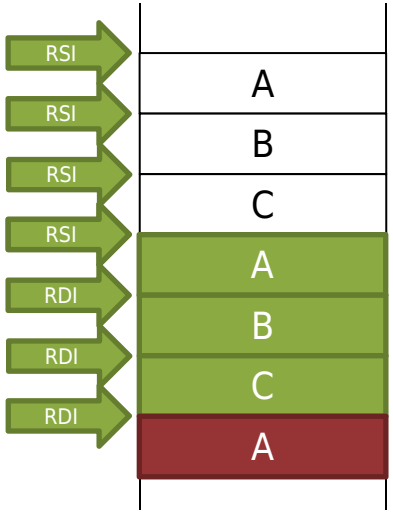
```
movq $0x0, %rax
movq $destination, %rdi
movq $size/8, %rcx
cld
stosq
```

- Il microcodice dello z64 incrementa/decrementa i valori di RDI e RSI in funzione di DF
- RCX viene sempre decrementato
- Se RCX è diverso da zero, RIP viene decrementato di 8

Sorgente e destinazione sovrapposte

- Se la sorgente e la destinazione sono sovrapposte, una copia in avanti può portare ad un risultato errato

```
movb $src, %rsi
movb $dst, %rdi
movb $size, %rcx
cld
movsb
```



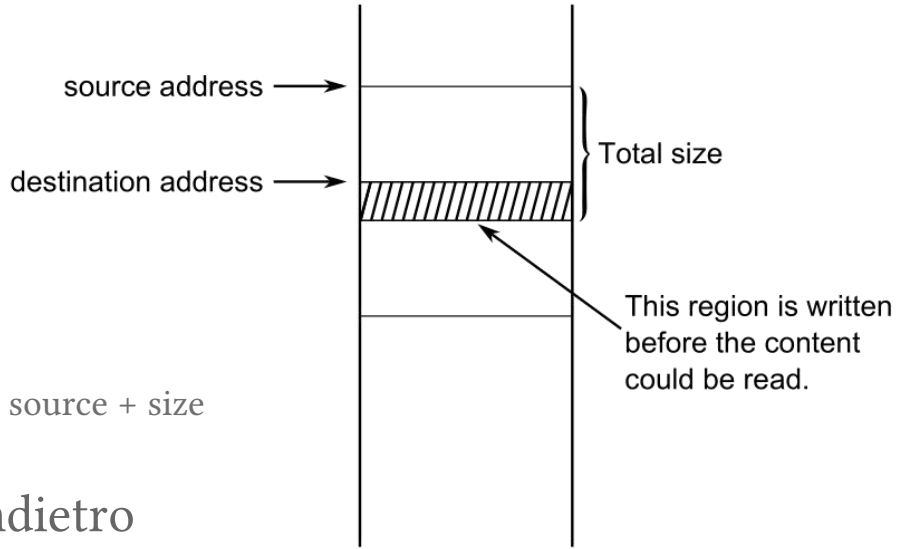
- Soluzione: effettuare la copia all'indietro

Sorgente e destinazione sovrapposte

- Se la sorgente e la destinazione sono sovrapposte, una copia in avanti può portare ad un risultato errato

```
movb $0x800, %rsi
movb $0x810, %rdi
movb $0x20, %rcx
cld
movsb
```

destination < source + size



- Soluzione: effettuare la copia all'indietro
  - RSI e RDI puntano alla fine dei buffer coinvolti
  - `std` configura il processore per fare la copia all'indietro

Equivalenti in C

```
#include <string.h>
```

```
void *memcpy(void *restrict dest, const void *restrict src, size_t n);
```

Effettua una copia memoria/memoria in avanti

```
void *memmove(void *dest, const void *src, size_t n);
```

Effettua una copia memoria/memoria all'indietro (i buffer possono sovrapporsi)

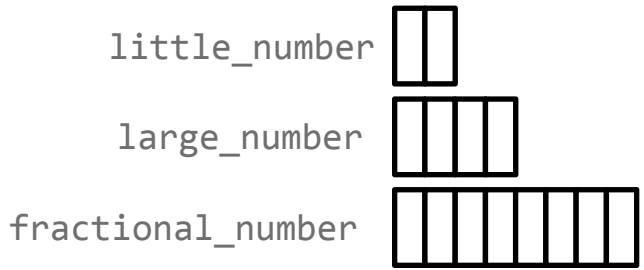
```
void *memset(void *s, int c, size_t n);
```

Imposta un'area di memoria al valore `c`

Tipo unione

- Un'unione è un valore che può assumere una qualsiasi tra diverse rappresentazioni o formati all'interno della stessa posizione in memoria
- Si tratta quindi di un blocco di memoria che viene utilizzata per conservare, una per volta, variabili di tipo differente

```
union number {
  short little_number;
  int large_number;
  double fractional_number;
} numbers;
```



Union cast

file: union-cast.c

```c
#include <stdio.h>
#include <stdint.h>

union binary_float_t {
        float real;
        uint32_t integer; // Assumes float is 32 bits wide
};

int main(void)
{
        union binary_float_t f;
        f.real = 3.141592F;
        printf("Hex representation of %f is %#04x\n", f.real, f.integer);
        return 0;
}
```

Campi di bit

- In alcuni casi può essere utile modificare singoli *flag* all'interno di un tipo primitivo
  - registri di controllo
  - *bitmap*, per raggruppare insieme più variabili booleane
- I *bit field* permettono di specificare operazioni su singoli bit o gruppi di bit all'interno dei membri di una `struct`
- A livello assembly, queste operazioni sono realizzate utilizzando le operazioni logiche della ALU implementando operazioni bit a bit *con maschere di bit* (*bit fiddling*)

Campi di bit: IEEE 754

file: ieee754.c

```
union flt {
        struct ieee754 {
                uint32_t mantissa: 23;
                uint32_t exponent: 8;
                uint32_t sign: 1;
        } raw;
        float f;
};

number.raw.sign = 1;
number.raw.exponent = 120;
number.raw.mantissa = 1685475;

printf("\fConverting %d %s %s to float:\n", number.raw.sign, exponent,
        mantissa);
printf("\t%f\n", number.f);
```

Maschere di bit: forzatura

- Per forzare dei bit ad un valore specifico, si usano maschere di bit
- Per forzare un bit a 1, si utilizza l'istruzione or:
  - Forzatura a 1 del bit più significativo: `orl $0x80000000, %eax`
- Per forzare un bit a 0, si utilizza l'istruzione and:
  - Forzatura a 0 del bit più significativo: `andl $0x7FFFFFFF, %eax`
- per invertire un bit, si utilizza l'istruzione xor:
  - Inversione dell'ultimo bit: `xorl $0x80000000, %eax`
  - Per azzerare un registro, si possono usare due istruzioni equivalenti:
    - `movq $0, %rax`
    - `xorq %rax, %rax`
  - La seconda è preferibile perché più efficiente
- Si possono comporre maschere di bit per forzare più bit contemporaneamente

Maschere di bit: estrazione

- Supponiamo di avere un numero a 32 bit e di voler estrarre il valore dei 3 bit meno significativi
- Si costruisce una maschera di bit del tipo 000....00111, equivalente a 7
- Estrazione dei bit: `andl $7, %eax`
- Per verificare se i bit sono a zero: `testl $7, %eax`

- Che cosa fa l'istruzione `testq $2, %rax`?
- Che cosa fa l'istruzione `testq %rax, %rax`?

Puntatori a funzione

- In C è possibile utilizzare dei *puntatori a funzione*
- Si tratta di *variabili* a cui possono essere assegnati indirizzi di memoria all'interno della sezione `.text`
- Tramite questi puntatori è possibile invocare le funzioni puntate

- Sono uno degli strumenti fondamentali per il supporto della programmazione a oggetti in cui è possibile utilizzare l'*overloading delle funzioni*
- In generale, permettono di selezionare dinamicamente una funzione da chiamare
  - per rispondere ad eventi di un determinato tipo
  - per realizzare strutture dati generiche

Puntatori a funzione

file: function-pointers.c

```c
#include <stdio.h>

int somma(int a, int b) {
    return a + b;
}

int sottrazione(int a, int b) {
    return a - b;
}

int main() {
    int (*operazione)(int, int); // Puntatore a funzione

    operazione = somma;
    printf("Somma: %d\n", operazione(5, 3));

    operazione = sottrazione;
    printf("Sottrazione: %d\n", operazione(5, 3));

    return 0;
}
```

Puntatori a funzione in assembly

- L'uso dei puntatori a funzione è particolarmente semplice in assembly
- Le calling convention catturano il passaggio di parametri indipendentemente dalla funzione chiamata
  - Tuttavia se si passano i parametri nel modo sbagliato ci può essere undefined behaviour
- Per chiamare una funzione memorizzata in un puntatore, si utilizza una chiamata assoluta:

```
movq $function, f_pointer
movq f_pointer, %rax
call *%rax
```

puntatore a funzione