

# Laboratorio di Programmazione di Sistema

## Programmazione Procedurale 3

Luca Forlizzi, Ph.D.

Versione 20.2



Luca Forlizzi, 2020

© 2020 by Luca Forlizzi. This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode>.

# Procedure e Località dei Dati

- La Programmazione Procedurale viene usata per strutturare un programma, dividendolo in parti
- La suddivisione di un programma ne rende più semplice la realizzazione, a condizione che le parti siano molto indipendenti
- Infatti, se le parti sono indipendenti
  - Possono essere costruite e modificate senza che sia necessario tenere conto delle altre parti del programma
  - Possono essere realizzate da programmatori diversi che non conoscono i dettagli del lavoro degli altri

## Procedure e Località dei Dati

- Per poter avere procedure indipendenti le une dalle altre, è necessario che ci siano meccanismi per rendere i dati di una procedura *locali*, ovvero inaccessibili dalle altre procedure
- Inoltre, per sfruttare appieno il potere della ricorsione, è necessario che ogni distinta attivazione di una procedura abbia dati indipendenti dalle altre attivazioni
- Negli *HLL* che adottano il paradigma della Programmazione Procedurale, questo meccanismo consiste nella possibilità di definire variabili *locali* ad una procedura

## Località dei Dati in C

- Le variabili in C possono essere
  - *locali* ad una funzione
  - *esterne*, ovvero accessibili da tutte le funzioni
- Inoltre è possibile avere variabili locali a singole attivazioni di una funzione, capacità indispensabile per il calcolo ricorsivo
- I dettagli di questi meccanismi sono illustrati nel capitolo 10 di **[Ki]**

## Località dei Dati in *ASM*

- I registri sono dispositivi di memorizzazione efficienti e semplici da usare
- Tuttavia i registri sono intrinsecamente non-locali, il che pone ostacoli all'applicazione della Programmazione Procedurale
- Un primo esempio è l'utilizzo di `ra` da parte di tutte routine, che deriva dal fatto che le istruzioni di chiamata di routine di MIPS32 memorizzano in tale registro il proprio indirizzo di ritorno
- Di ciò si deve tener conto nella realizzazione di routine annidate

## Località dei Dati in *ASM*

- La realizzazione di routine annidate in altre routine, richiede che il contenuto di `ra` venga copiato in un'altra parola, di registro o di memoria, per evitare che venga sovrascritto da una successiva chiamata di routine
- Ad esempio, se la routine **main** chiama la routine **A**, la quale a sua volta chiama la routine **B**, l'istruzione che chiama **B** sovrascrive l'indirizzo di ritorno da **A** memorizzato in `ra`
- È dunque necessario che l'indirizzo di ritorno da **A**, venga copiato da `ra` in una diversa parola, prima di eseguire l'istruzione di chiamata di **B**

## Località dei Dati in *ASM*

- In generale, se una routine **A** chiama una seconda routine **B**, allora **A** deve tenere conto delle modifiche al contenuto dei registri operate da **B**
- Se **B** modifica il contenuto di un registro **R** e **A** necessita che **R** abbia il medesimo contenuto prima e dopo la chiamata di **B**, è necessario:
  - 1 *salvare* il contenuto di **R**, prima che venga modificato durante l'esecuzione di **B**
  - 2 al termine dell'esecuzione di **B** (o della parte di **B** che modifica **R**), *ripristinare* il contenuto di **R** al valore che esso aveva prima della chiamata di **B**
- Nella maggioranza dei casi, salvataggio e ripristino possono avvenire o all'interno di **A** o all'interno di **B**

## Località dei Dati in *ASM*

- Salvataggio e ripristino all'interno della *routine chiamante* **A**
  - Permette di fare tali operazioni solo per i registri di cui **A** necessita venga preservato il contenuto, e quindi porta ad avere codice più efficiente in termini di velocità d'esecuzione
  - Ha lo svantaggio che le istruzioni che effettuano tali operazioni, devono essere scritte per ciascuna chiamata di **B**
- Salvataggio e ripristino all'interno della *routine chiamata* **B**
  - Permette di scrivere una sola volta il codice che effettua tali operazioni, ottenendo programmi con meno istruzioni
    - Maggior comodità di scrittura
    - Minor rischio di errori
    - Minore occupazione di memoria

## Località dei Dati in *ASM*

- La necessità di tenere conto delle possibili modifiche apportate al contenuto dei registri dalle routine chiamate, appesantisce notevolmente il lavoro dei programmatori, soprattutto nel caso di programmi sviluppati da più persone
- Per tentare di limitare il problema, si adottano convenzioni nell'utilizzo dei registri
- In MIPS32, esiste una convenzione di riferimento, proposta dalla documentazione ufficiale
  - Attribuisce dei ruoli ai vari registri, anche in relazione alla comunicazione tra routine
  - Ripartisce il compito di salvare e ripristinare i registri in parte sulle routine chiamanti e in parte su quelle chiamate
- Per una descrizione completa della convenzione, si veda **[MIPS32]**; nel seguito descriviamo le regole principali

## Località dei Dati in *ASM*

- I registri `v0` e `v1` servono per restituire i risultati di una routine (la lettera “v” sta per “value”)
- I registri `a0`, `a1`, `a2`, `a3` sono usati per passare gli argomenti a una routine (la lettera “a” sta per “argument”)
- Otto registri, `s0–s7` sono usati per contenere dati di cui una routine chiamante ha bisogno, sia prima che dopo aver effettuato chiamate ad altre routine (la lettera “s” sta per “static”)
  - Se tali registri sono necessari sia alla routine chiamante che a quella chiamata, è compito della routine chiamata preservarne il contenuto

## Località dei Dati in *ASM*

- Dieci registri,  $t_0$ – $t_9$  sono usati per contenere dati di cui si ha bisogno solo “nel breve termine” ovvero per contenere risultati temporanei di alcune elaborazioni (“t” sta per “temporary”)
  - Se tali registri sono necessari sia a una routine chiamante che a una chiamata, è compito della routine chiamante preservarne il contenuto
- Si osservi come, con questa convenzione, il compito di salvare e ripristinare il contenuto dei registri che vengono utilizzati sia da una routine chiamante che da una chiamata, viene ripartito su entrambe

## Località dei Dati in *ASM*

- Adottare una convenzione nell'uso dei registri non è comunque sufficiente a superare le limitazioni dei registri, in modo particolare in *ASM-PM* che hanno un numero limitato di registri
- Ad esempio, in MC68000 ci sono 16 registri a disposizione (tra dati e indirizzi): difficilmente bastano per soddisfare le esigenze di tutte le routine di un programma di media complessità
- La CPU 8086 ha a disposizione 8 registri ...

## Località dei Dati in *ASM*

- Nella scrittura di programmi complessi, quindi è necessario immagazzinare dati nella memoria principale, attraverso metodi che preservino la località dei dati
- Tali meccanismi possono essere usati anche per realizzare la comunicazione tra routine, in alternativa ai registri
- L'insieme delle informazioni che una routine ha necessità di comunicare o preservare, viene detto *activation frame* o *activation record* e comprende:
  - Contenuto dei registri da preservare
  - Dati locali
  - Argomenti per le procedure chiamate
  - Risultati delle procedure chiamate

## Località dei Dati in *ASM*

- L'activation frame di una routine può essere allocato staticamente, nella sezione dati, con le modalità che già conosciamo:
  - Per ciascuna routine si alloca una determinata quantità di memoria, che essa usa in modo esclusivo per memorizzare il proprio activation frame
  - Ogni routine viene scritta in modo da non utilizzare la memoria usata per gli activation frame di altre routine, tranne la parte relativa a parametri e a risultati, al fine di comunicare dati

## Località dei Dati in *ASM*

- Si noti che nei linguaggi *ASM* il programmatore non ha a disposizione costrutti per *riservare* della memoria per una determinata routine che *impediscono* automaticamente l'uso di tale porzione di memoria da parte di altre routine
- È responsabilità del programmatore *ASM* o del traduttore, fare in modo che ogni routine non usi memoria “riservata” ad altre routine
- Questo meccanismo ha comunque due severi limiti:
  - 1 Ci sono routine che vengono usate raramente: la memoria allocata per i loro activation frame non è usata al meglio
  - 2 Non è possibile la ricorsione, in quanto attivazioni diverse della stessa routine condividono l'activation frame allocato staticamente

## Routine Ricorsive

- Non è possibile memorizzare gli activation frame di routine ricorsive in parole di memoria allocate staticamente né tantomeno in registri
- Infatti, quando un'attivazione di una routine esegue una chiamata ricorsiva, dando inizio ad una nuova attivazione, l'attivazione chiamante viene sospesa
- Tuttavia, successivamente, essa dovrà riprendere, con l'activation frame nello stato in cui si trovava al momento della chiamata
- Usando registri o memoria allocata staticamente, tutte le attivazioni di una routine memorizzerebbero il proprio activation frame negli stessi registri o parole di memoria, interferendo tra di loro

# Routine Ricorsive

- C'è bisogno di una tecnica che consenta di allocare una certa quantità di memoria per ogni distinta attivazione di routine
- L'esecuzione di una routine ricorsiva può generare un numero di chiamate ricorsive non stabilito staticamente, ovvero al momento della traduzione
- Dunque, non è possibile stabilire prima dell'esecuzione del programma, il numero di attivazioni che, in un certo momento, sono iniziate ma non sono ancora terminate
- Quindi, la quantità di memoria che è necessario allocare non può essere stabilita prima dell'esecuzione del programma

# Ricorsione e Stack

- La tecnica più semplice ed efficace per allocare la memoria necessaria per gli activation frame di ogni distinta attivazione di una routine ricorsiva, si basa sull'impiego di uno *stack*
- Uno *stack* è una struttura dati, in cui possono essere effettuati inserimenti e rimozioni di *elementi*
- La quantità di elementi che uno stack memorizza non è prefissata e può variare dinamicamente, da 0 fino a un certo valore massimo

# Ricorsione e Stack

- Più formalmente, uno stack è un contenitore di elementi costituiti da valori di uno specifico tipo  $T$
- Un elemento può essere inserito mediante una operazione detta *push*
- Un elemento può essere rimosso mediante una operazione detta *pop*
- Un'operazione di *pop* rimuove l'elemento inserito dalla più recente operazione di *push*
- Tale regola, che caratterizza il comportamento di uno stack, viene detta *Last In, First Out* o LIFO

# Ricorsione e Stack

- Si osservi che la sequenza in cui vengono eseguite le attivazioni di una serie di routine annidate, segue anch'essa una regola LIFO
- L'ultima attivazione ad iniziare (Last in) è la prima a terminare (First Out)
- Ciò rende uno stack particolarmente adatto per allocare la memoria per gli activation frame delle attivazioni di routine
  - Subito prima o subito dopo una nuova chiamata, viene inserito nello stack l'activation frame della nuova attivazione
  - Subito prima o subito dopo il ritorno, viene rimosso dallo stack l'activation frame della attivazione appena conclusa

# Ricorsione e Stack

- Poiché uno stack può memorizzare una quantità non fissata di dati, esso è in grado di contenere un numero non fissato di activation frame
- Attivazioni differenti di una routine ricorsiva, inseriscono i rispettivi activation frame in posizioni diverse dello stack, senza quindi sovrascrivere i dati memorizzati dalle attivazioni precedenti
- La strategia di gestione LIFO, propria dello stack, si adatta perfettamente all'ordine LIFO delle attivazioni di routine annidate
- Infatti non è mai necessario rimuovere un activation frame che non si trova sulla cima dello stack

# Ricorsione e Stack

- Supponiamo, ad esempio, che la routine **main** chiami una routine **A**, la quale a sua volta chiami una routine **B**
  - ① In occasione della chiamata di **A**, l'activation frame *FA* di **A**, viene inserito nello stack
  - ② In occasione della chiamata di **B**, l'activation frame *FB* di **B**, viene inserito nello stack
  - ③ A questo punto i dati in *FA* sono sempre nello stack, ma non sulla cima quindi non possono essere rimossi; ciò non è un problema perché l'esecuzione di **A** viene sospesa quando inizia l'esecuzione di **B**: di conseguenza **A** non può terminare prima che termini **B**, e non c'è bisogno di rimuovere *FA* prima di *FB*
  - ④ Quando l'esecuzione di **B** termina, *FB* viene rimosso: a questo punto *FA* è di nuovo sulla cima dello stack, e quindi torna ad essere possibile la sua rimozione
  - ⑤ Quando l'esecuzione di **A** termina, *FA* viene rimosso

## Realizzare uno stack in *ASM*

- È possibile realizzare uno stack in modo molto efficiente
- Si utilizza un'area di memoria  $S$  detta *area di stack*
  - Negli ambienti operativi che permettono a più programmi di essere presenti in memoria,  $S$  viene allocata staticamente e in modo automatico dall'architettura di livello 2 (ISA) o da quella di livello 3
  - Negli altri casi il programma può scegliere in modo arbitrario indirizzo iniziale e dimensione di  $S$
- La dimensione di  $S$  è la quantità massima di byte che lo stack può utilizzare, e in molte architetture ha dimensione fissa
- L'area  $S$  è allocata staticamente dal programma, ma i byte o parole che la formano vengono poi assegnati dinamicamente alle attivazioni di routine per memorizzarvi gli activation frame

## Realizzare uno stack in *ASM*

- I dati effettivamente memorizzati nello stack occupano una porzione  $S'$  di  $S$ , detta *area in uso dello stack*
- Dunque  $S'$  è un'area di memoria contenuta in  $S$
- Il byte di indirizzo massimo di  $S'$  coincide con il byte di indirizzo massimo di  $S$
- Ovvero  $S'$  si estende all'interno di  $S$ , nella direzione degli indirizzi decrescenti
- Chiamiamo *base dello stack*, il byte che ha indirizzo successivo al byte di  $S$  che ha indirizzo massimo (quest'ultimo è anche il byte di  $S'$  che ha indirizzo massimo)
- Si noti che la base dello stack non appartiene all'area di stack

## Realizzare uno stack in *ASM*

- Chiamiamo cima dello stack, un'area di memoria  $E$ , contenuta in  $S'$  tale che:
  - $E$  è grande abbastanza da contenere un activation frame
  - il byte di  $E$  che ha indirizzo minimo, è il byte che ha indirizzo minimo tra quelli di  $S'$
- Si noti che l'indirizzo della cima di uno stack è uguale all'indirizzo dell'area in uso dello stack  $S'$
- Ad ogni operazione di *push*, che inserisce un nuovo activation frame nello stack, la dimensione di  $S'$  viene aumentata, aggregando ad  $S'$  un insieme di byte che:
  - È grande abbastanza da contenere l'activation frame
  - Ha come massimo degli indirizzi dei byte che lo formano, l'indirizzo precedente l'indirizzo che  $S'$  ha prima di effettuare l'operazione di *push*

## Realizzare uno stack in *ASM*

- I byte aggiunti, formano la nuova cima dello stack e, quindi, la cima dello stack “si sposta” in direzione di indirizzi decrescenti
- Supponiamo, ad esempio, che  $S$  sia l'area compresa tra gli indirizzi 100 e 200 (dunque l'indirizzo della base dello stack è 200)
- Se, in un certo momento,  $S'$  è grande 20 byte, allora essa è l'area compresa tra gli indirizzi 180 e 200
- Se una operazione di *push* inserisce un nuovo activation frame, che richiede 10 byte per essere memorizzato,  $S'$  diventa l'area compresa tra 170 e 200 mentre la cima è l'area compresa tra 170 e 180

## Realizzare uno stack in *ASM*

- Ogni operazione di *pop* elimina un activation frame dallo stack, e quindi diminuisce la dimensione di  $S'$ , eliminando da  $S'$  l'insieme di byte che costituisce la cima dello stack
- L'activation frame che prima dell'operazione si trovava negli indirizzi successivi alla cima, diventa la nuova cima
- Ad ogni *pop*, quindi, la cima dello stack “si sposta” in direzione di indirizzi crescenti
- Facendo riferimento all'esempio precedente, se viene eseguita una operazione di *pop*, l'activation frame inserito dalla precedente *push* viene rimosso:  $S'$  torna ad essere l'area compresa tra 180 e 200 e la cima diventa l'area compresa tra 180 e 190

## Realizzare uno stack in *ASM*

- Per realizzare lo stack si usa una variabile, detta *Stack Pointer* (*SP*), che:
  - contiene l'indirizzo della base dello stack quando lo stack è vuoto (ad esempio quando inizia un programma)
  - contiene l'indirizzo della cima dello stack quando lo stack contiene almeno un dato
- In ogni momento, dunque,  $S'$  è l'insieme di byte che hanno indirizzi compresi tra l'indirizzo memorizzato in *SP* (incluso) e l'indirizzo della base dello stack (escluso)

## Realizzare uno stack in *ASM*

- Per effettuare *push*, si decrementa SP di un valore pari alla dimensione dei dati da inserire nello stack
- In questo modo si aggiunge ad  $S'$  spazio in cui memorizzare la nuova cima
- Per effettuare *pop*, si incrementa SP di un valore pari alla dimensione dei dati da rimuovere dallo stack
- In questo modo i byte in cui è memorizzata la (vecchia) cima non fanno più parte di  $S'$
- Dopo l'incremento, SP contiene l'indirizzo della nuova cima, ovvero l'indirizzo del dato che era stato inserito subito prima dei dati appena rimossi

## Realizzare uno stack in *ASM*

- Si osservi che per rimuovere dati dallo stack non è necessario sovrascrivere l'area di memoria che li contiene
- Tale area di memoria verrà sovrascritta, al momento del bisogno, da eventuali operazioni di *push* successive
- Dunque, la rimozione di dati dallo stack consiste semplicemente nell'incrementare SP, ed è per questo molto efficiente

## Realizzare uno stack in *ASM*

- Per permettere la ricorsione, dunque, l'activation frame di ciascuna attivazione di routine viene allocato all'interno dell'area di stack
- In questi casi, gli activation frame delle routine vengono anche chiamati *stack frame*
- Per uniformarci alla terminologia di C Standard, chiamiamo *allocazione automatica* questa modalità di allocazione, anche se negli *ASM* non viene effettuata in modo automatico

## Realizzare uno stack in *ASM*

- Capire la differenza tra allocazione statica e automatica è cruciale nella programmazione di sistema, per questo motivo è opportuno sottolinearla ancora una volta
- Nell'allocazione statica, per ogni singola variabile viene allocata un'area prima che inizi l'esecuzione del programma
- Nell'allocazione automatica
  - l'area di stack nel suo complesso viene allocata per il programma prima che inizi l'esecuzione
  - per ciascuna distinta attivazione delle routine, vengono allocate aree di memoria all'interno dell'area di stack, in cui memorizzare le singole variabili durante l'esecuzione del programma

## Stack e Ricorsione in MIPS32

- Per la gestione dello stack, MIPS32 non prevede né istruzioni né registri speciali, in quanto è sufficiente usare un comune registro per rappresentare la variabile SP
- La convenzione proposta dalla documentazione ufficiale MIPS32, prevede la possibilità di utilizzare 2 registri per la gestione dello stack
- Il registro 29, che ha nome simbolico `sp`, viene usato per contenere la variabile SP

## Stack e Ricorsione in MIPS32

- Il registro 30, che ha nome simbolico `fp` (da *frame pointer*), viene usato per conservare l'indirizzo dell'activation frame della routine in esecuzione, nei casi in cui esso non si trovi più nella cima dello stack, a seguito di operazioni di *push*
- L'utilizzo del *frame pointer* per contenere l'indirizzo dell'activation frame può essere conveniente o meno: traduttori C diversi fanno scelte differenti al riguardo

## Stack e Ricorsione in MC68000

- Le istruzioni di chiamata di routine di MC68000 non memorizzano l'indirizzo di ritorno in un registro, ma usano invece uno stack
- Dunque in MC68000 si usa uno stack per implementare qualunque routine, anche non ricorsiva
- MC68000 prevede l'esistenza di due stack realizzati con il supporto dell'hardware:
  - Uno viene impiegato nel modo di funzionamento *user*
  - L'altro nel modo di funzionamento *supervisor*

## Stack e Ricorsione in MC68000

- Per entrambi gli stack, come variabile SP viene usato il registro indirizzi a7 che è quindi un general purpose register with special functions
  - Ad ogni passaggio tra stato *user* e stato *supervisor*, a7 viene modificato automaticamente
  - Le istruzioni di chiamata di routine effettuano *push* dell'indirizzo di ritorno, nello stack di cui a7 è SP
  - Le istruzioni di ritorno da una routine effettuano *pop* dell'indirizzo di ritorno, dallo stack di cui a7 è SP

## Stack e Ricorsione in MC68000

- MC68000 ha due istruzioni di chiamata di routine (entrambe incondizionate)
  - `bsr` specifica staticamente un indirizzo di destinazione del salto mediante l'indirizzamento PC-indicizzato
  - `jsr` specifica l'indirizzo di destinazione del salto mediante diversi modi di indirizzamento per dati in memoria, tra cui diretto-memoria e indiretto-registro
- Entrambe effettuano *push* dell'indirizzo di ritorno nello stack attualmente in uso ed effettuano il salto alla routine
- Per effettuare il ritorno dalla routine sono disponibili istruzioni dedicate
- Quella di uso più comune è `rts`, che esegue *pop* dell'indirizzo di ritorno dallo stack attualmente in uso e copia tale indirizzo nel PC

## Stack e Ricorsione in MC68000

- Le operazioni di *push* e di *pop* dell'activation frame nello stack in uso, vengono fatte di norma mediante l'istruzione *move* usando:
  - il modo di indirizzamento indiretto-registro con pre-decremento per fare *push*
  - il modo di indirizzamento indiretto-registro con post-incremento per fare *pop*
- È inoltre disponibile l'istruzione speciale *movem* che effettua, con una singola operazione, *push* o *pop* di un qualunque insieme di registri scelto dal programmatore
- Infine, MC68000 dispone della coppia di istruzioni *link* e *unlk* che permettono di allocare e deallocare un activation frame nello stack in uso, inserendo l'indirizzo del *frame pointer* in un registro indirizzi diverso da *a7*

- Per assimilare e approfondire i contenuti di questa presentazione, si consiglia di studiare anche
  - Esempi *Routine 3*, *Routine 4*, *Routine 5* e *Routine 6* disponibili su *Edu99*
  - Capitolo 10 di **[Ki]**