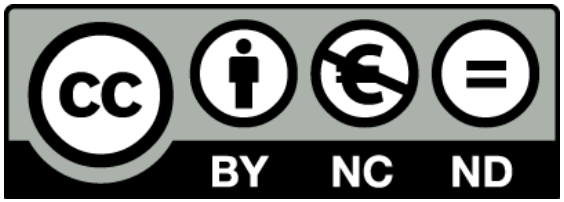
Laboratorio di Programmazione di Sistema

Programmazione Procedurale 3

Luca Forlizzi, Ph.D.

Versione 23.1



Routine in *ASM*

- In una precedente presentazione è stata illustrata la realizzazione di routine in *ASM*
- In particolare si è sottolineata l'esigenza di memorizzare
  - indirizzo di ritorno al chiamante
  - variabili locali
  - parametri e risultato, che sono particolari variabili locali
  - registri da preservare
- Si è discusso di come queste informazioni (collettivamente chiamate *activation frame*) possano essere memorizzate in
  - registri
  - memoria allocata staticamente

Routine in *ASM*

- L'uso, eventualmente combinato, di tali tecniche per memorizzare gli activation frame, permette di realizzare la maggior parte delle procedure comunemente usate, benché non sia ideale in quanto non sempre riesce a sfruttare la località temporale per utilizzare al meglio la memoria
- In questa presentazione si mostrerà invece che c'è un'importante categoria di procedure che non possono essere realizzate mediante queste tecniche di memorizzazione, e quindi si presenterà una nuova tecnica che
  - permette la realizzazione di tali procedure
  - consente di utilizzare meglio la memoria anche nella realizzazione delle altre procedure

Routine in *ASM*

- Le particolari procedure che necessitano di una diversa tecnica di memorizzazione dell'*activation frame* sono le *procedure ricorsive*
- La ricorsione è una fondamentale tecnica di problem solving, ed è estremamente utile potersi esprimere in modo ricorsivo anche nella scrittura di programmi
- Le procedure ricorsive di un *HLL* devono essere tradotte in *ASM* mediante *routine ricorsive*
- Per realizzare routine ricorsive è necessario memorizzarne gli *activation frame*

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 che permette di memorizzare, senza sovrascritture, gli activation frame di tutte le chiamate ricorsive di una routine, è nota come *allocazione dinamica basata su stack* o semplicemente *allocazione su stack*
- Si tratta di una tecnica efficace, relativamente semplice ed anche piuttosto efficiente
- 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 F_A di **A**, viene inserito nello stack
  2. In occasione della chiamata di **B**, l'activation frame F_B di **B**, viene inserito nello stack
  3. A questo punto i dati in F_A 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 F_A prima di F_B
  4. Quando l'esecuzione di **B** termina, F_B viene rimosso: a questo punto F_A è di nuovo sulla cima dello stack, e quindi torna ad essere possibile la sua rimozione
  5. Quando l'esecuzione di **A** termina, F_A 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`