Programmazione Assembly

SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

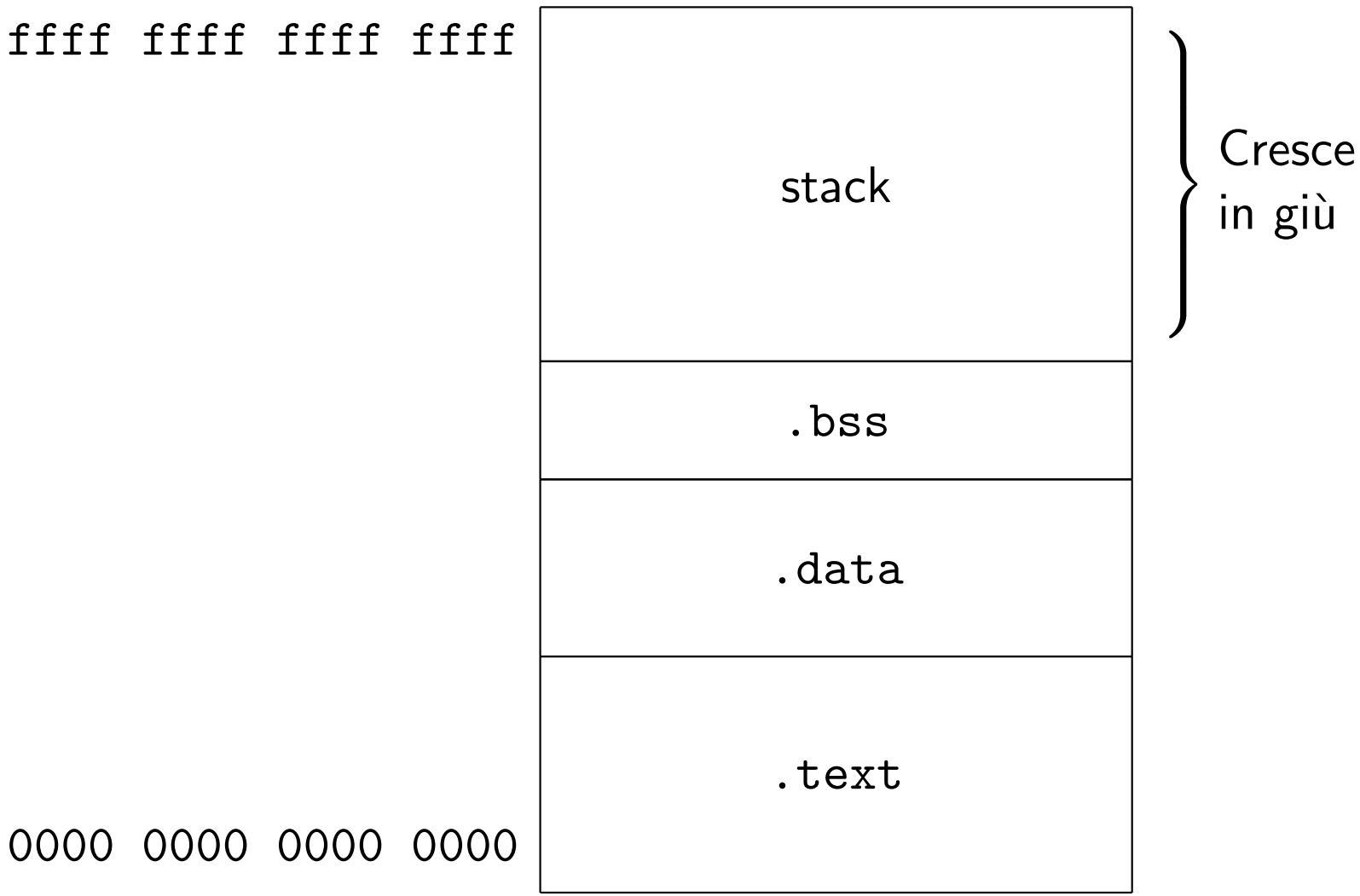
Architettura dei Calcolatori Elettronici
Sapienza, Università di Roma

A.A. 2017/2018

Anatomia di un programma

```
1  #include <stdio.h>
2
3  int square_int(int x) {
4          return x*x;
5  }
6
7  int value = 10;
8
9  int main(void) {
10         printf("%d\n", square_int(value));
11         return 0;
12 }
```

Anatomia di un programma (2)



Scheletro di un programma assembly

```
1  .org [INDIRIZZO CARICAMENTO]
2
3  .data
4
5  # Dichiarazione costanti e variabili globali
6
7  .text
8
9          # Corpo del programma
10
11     hlt # Per arrestare l'esecuzione
```

Direttive Assembly

- **Labels**: 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 (2)

- **.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`

Direttive Assembly (3)

- **.quad expressions**: riserva memoria (di dimensione quadword) per `expressions`
- **.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 idn**/**.handler idn**: identifica l'inizio della routine di servizio associato al codice `idn`

Esempio di programma assembly

```
1  .org 0x800
2  .data
3  message: .ascii "Hello World!"
4  counter: .byte 0
5  .text
6  main:
7          movq $message, %rax
8      .repeat:
9          cmpb $0, (%rax)
10         jz .end
11         addb $1, counter
12         addq $1, %rax
13         jmp .repeat
14     .end:
15         hlt
```

Confronti: aritmetica non segnata

L'aritmetica non segnata impone che `source` e `dest` siano ≥ 0.

Condition	Aritmetica non segnata	Aritmetica segnata
`dest` $>$ `source`	CF $= 0$ and ZF $= 0$	ZF $= 0$ and SF $=$ OF
`dest` $\geq$ `source`	CF $= 0$	SF $=$ OF
`dest` $=$ `source`	ZF $= 1$	ZF $= 1$
`dest` $\leq$ `source`	CF $= 1$ or ZF $= 1$	ZF $= 1$ or SF $\neq$ OF
`dest` $<$ `source`	CF $= 1$	SF $\neq$ OF
`dest` $\neq$ `source`	ZF $= 0$	ZF $= 0$

Confronti: un esempio

```
1  .org 800h
2
3  .data
4
5  x: .word 3
6  y: .word -2
7
8  .text
9      # Imposta a 1 l'indirizzo 0x1280 solo se x > y
10     # Assumo che x ed y possano assumere valori negativi
11     movw x, %ax
12     movw y, %bx
13     cmpw %bx, %ax
14     jz .nonImpostare
15
```

Confronti: un esempio (2)

```
16      js .SFset
17      jo .nonImpostare # eseguita se SF = 0. Se OF = 1 allora SF != OF
18      jmp .set
19  .SFset:
20      jno .dont # eseguita se SF = 1. Se OF = 0 allora Sf != OF
21
22  .set:
23       movb $1, 0x1280
24
25  .dont:
26      hlt
```

If-Then-Else

- I confronti tra registri possono essere utilizzati come condizioni di costrutti if-then-else
- Ogni blocco di codice dovrà avere alla fine un salto al termine del costrutto if-then-else
- Ogni controllo di condizione, se non verificato, dovrà saltare al controllo successivo

```
1 if(condizione 1) {
2     < blocco A >
3 } else if (condizione 2) {
4     < blocco B >
5 } else {
6     < blocco C >
7 }
```

If-Then-Else

Un semplice (sciocco!) esempio:

```
1  int x = 1;
2  int val;
3
4  if(x == 2) {
5      val = 2;
6  } else if (x == 1) {
7      val = 1;
8  } else {
9      val = 0;
10 }
```

If-Then-Else

```
1  .org 0x800
2
3  .data
4
5  x: .byte 1
6  val: .byte 0
7
8  .text
9
10     movb x, %al
11
12     cmpb $2, %al # Test prima condizione
13     jnz .elseif
14     movb $2, val # blocco A
15     jmp .endif # Altrimenti andrei all'istruzione successiva
```

If-Then-Else (2)

```
16
17 .elseif:
18     cmpb $1, %al # Test seconda condizione
19     jnz .else
20     movb $1, val # blocco B
21     jmp .endif
22
23 .else:
24     movb $0, val
25
26 .endif:
27     hlt
```

Le variabili booleane?

- Il tipo *booleano* non esiste realmente nei processori
- Si utilizzano degli interi (tipicamente dei byte), e per convenzione si assume:
  - `false` = 0
  - `true` = !`false`
    (cioè qualsiasi valore diverso da 0, la libreria `stdbool.h` usa 1)

```
1 boolean var = true;
2
3 if(var) {
4     < blocco A >
5 } else {
6     < blocco B >
7 }
```

Le variabili booleane?

```
1  .org 0x800
2
3  .data
4  var: .byte 1 # considerato come 'true'
5
6  .text
7      cmpb $0, var
8      jz .elsebranch
9      nop # blocco A
10     jmp .endif
11 .elsebranch: nop # blocco B
12 .endif: hlt
```

Un errore comune

```
1       jnz .elsebranch
2       jmp .here # Un puro spreco di cicli di clock!
3                 # da notare che jz non sarebbe stato meglio!
4 .here: nop # blocco A
5       jmp .endif
6 .elsebranch: nop # blocco B
7 .endif: hlt
```

Saltare all'istruzione successiva non richiede alcun tipo di `jmp` particolare: è il comportamento comune di tutte le istruzioni!

Cicli `while`

Un ciclo while ha due forme, a seconda di dove si effettua il controllo sulla condizione:

```
while(<condizione>) {
    <codice>
}
```

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

```
do {
    <codice>
} while(<condizione>);
```

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

La seconda forma è nettamente più chiara in assembly, e dovrebbe essere utilizzata laddove possibile

Cicli `for`

Un ciclo for, in generale, ha un numero limitato di iterazioni:

```
int i;
for(i = 0; i < 3; i++) {
    <codice>
}
```

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

Per risparmiare un registro, si può riscrivere il controllo come:

```
cmpq $3, %rcx
```

Campo, vettore, o mappa di bit

- Utilizzare valori booleani è una pratica comune
- Eppure, 7 degli 8 bit sono sprecati!
- La *maschera di bit* utilizza un byte (o dato superiore) come vettore di valori booleani
  - Il registro FLAGS è un esempio di applicazione

- La CPU non sa operare su meno di un byte
- Dobbiamo trovare il modo di implementare almeno queste operazioni:
  - *Bit setting*: uno specifico bit viene impostato a 1
  - *Bit clearing*: uno specifico bit viene impostato a 0
  - *Bit testing*: il valore di un bit viene estratto

Operazione bit a bit: forzatura

- Per forzare dei bit ad un valore specifico, si usano ancora delle maschere di bit
- Per forzare un bit a 1, si utilizza l'istruzione `OR`
- Per forzare un bit a 0, si utilizza l'istruzione `AND`
- Per invertire un bit, di utilizza l'istruzione `XOR`

Per forzare a 1 l'ultimo bit in `R0`:

```
1 orl $0x80000000, %eax
```

Per forzare a 0 l'ultimo bit in `R0`:

```
1 andl $0x7FFFFFFF, %eax
```

Per invertire l'ultimo bit in `R0`:

```
1 xorl $0x80000000, %eax
```

Operazione bit a bit: reset di un registro

- L'istruzione `xor` permette di invertire un bit particolare in un registro
- Ciò vale perché:
  - $0 \oplus 1 = 1$
  - $1 \oplus 1 = 0$
- Questa stessa tecnica può essere utilizzata per azzerare un registro:

```
1 movq $0, %rax
2 xorq %rax, %rax
```

- Queste due istruzioni producono lo stesso risultato
- Tuttavia, è preferibile la seconda, perché è più efficiente

Operazione bit a bit: estrazione

- Supponiamo di avere un numero a 32 bit e di voler sapere qual è il valore dei tre bit meno significativi
- Si può costruire una maschera di bit del tipo `00...00111`, in cui gli ultimi tre bit sono impostati a 1
- La maschera di bit `00...00111` corrisponde al valore decimale 7 e al valore esadecimale `0x7`
- Si può quindi eseguire un `AND` tra il dato e la maschera di bit

```
1 testl $7, %eax
```

Altri usi di `test`: cosa fa l'istruzione `testl %eax, %eax`?

Manipolazioni di vettori

- Iterare su degli array è una delle operazioni più comuni
- Questa operazione può essere fatta in più modi
  - **Modo 1**: si carica in un registro l'indirizzo del primo elemento e si incrementa manualmente il puntatore per scandire uno alla volta gli elementi. La fine del vettore viene individuata con un confronto con il contenuto di un secondo regsitro (numero di elementi) che viene decrementato
  - **Modo 2**: come il modo 1, ma il primo elemento fuori dal vettore viene individuato dal suo indirizzo
  - **Modo 3**: in caso di tipi primitivi, si utilizza il registro indice per tenere traccia dell'elemento corrente

Manipolazioni di vettori

Modo 1:

```
1     movq $array, %rax # array: indirizzo primo elemento del vettore
2     movq $num, %rcx # num: numero di elementi
3 .loop:
4     movq (%rax), %rdx # carica un dato
5     # <processa i dati>
6     addq $8, %rax # Va all'elemento successivo
7     subq $1, %rcx # decrementa il contatore degli elementi
8     jnz .loop
```

Manipolazioni di vettori

Modo 2:

```
1  .org 0x800
2
3  .data
4      array: .long 1,2,3,4,5,6,7,8,9,10
5      endarr: .long 0xdeadc0de
6
7  .text
8      movq $array, %rax # array: indirizzo primo elemento del vettore
9      movq $endarr, %rbx # endarr: primo indirizzo fuori dal vettore
10   .loop:
11     addq (%rax), %rdx # processa un dato
12     addq $4, %rax # vai all'elemento successivo
13     cmpq %rax, %rbx
14     jnz .loop
15     hlt
```

Manipolazioni di vettori

Se non si ha a disposizione l'indirizzo del primo elemento fuori del vettore, ma si ha a disposizione il numero di elementi, si può calcolare l'indirizzo in questo modo:

```
1 movq $array, %rax # array: indirizzo primo elemento del vettore
2 movq $num, %rcx # num: numero di elementi
3 shlq $3, %rcx # left shift per trasformare il numero in una taglia
4 # Ogni elemento ha dimensione 8 byte ed uno shift di tre posizioni
      corrisponde a moltiplicare per 8
5 # %rcx contiene quindi la dimensione (in byte) dell'array. Sommando
       il valore dell'indirizzo di base si ottiene il primo indirizzo
       fuori dall'array
6 addq %rax, %rcx
```

e si può poi iterare utilizzando il modo 2

Manipolazioni di vettori

Modo 3:

```
1     xorq %rcx, %rcx # %rcx viene usato come indice
2     movq $array, %rax # %rax viene usato come base
3  .loop:
4     movq (%rax, %rcx, 8), %rbx # sposta i dati dove serve
5     # <processa i dati>
6     addq $1, %rcx
7     cmpq $num, %rcx
8     jnz .loop
```

Manipolazioni di vettori

Modo 3 senza utilizzare la base:

```
1     xorq %rcx, %rcx # %rcx viene usato come indice
2 .loop:
3     movq array(, %rcx, 8), %rbx # sposta i dati dove serve
4     # <processa i dati>
5     addq $1, %rcx
6     cmpq $num, %rcx
7     jnz .loop
```

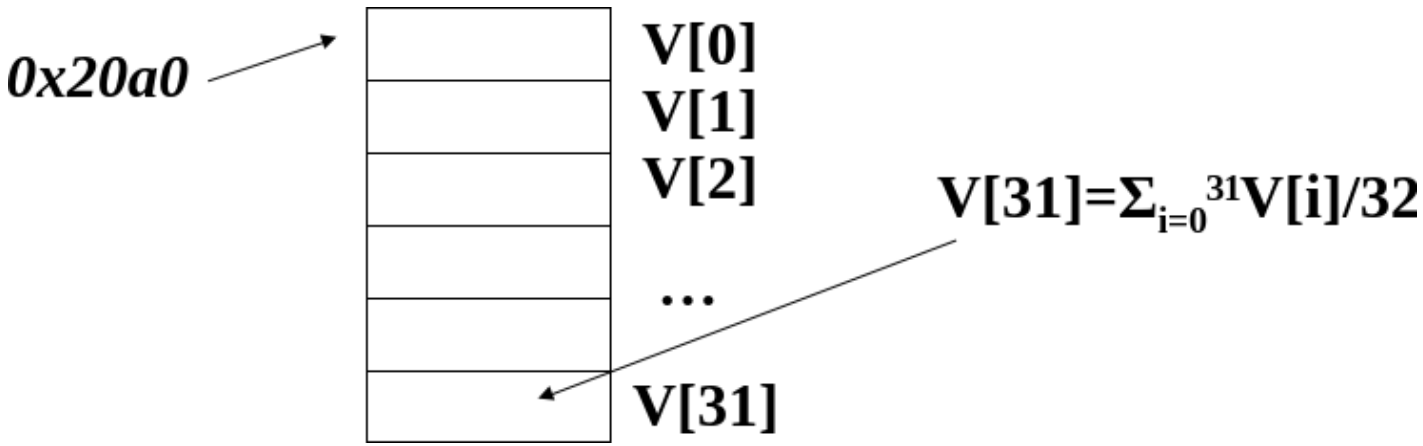
Manipolazioni di vettori

Se dobbiamo saltare degli elementi in un vettore, si può decrementare il contatore degli elementi ancora da controllare. Nel controllo di terminazione del ciclo dobbiamo però accertarci che il puntatore non sia andato oltre l'ultimo elemento!

```
1     movq $array, %rax
2     movq $num, %rcx
3 .loop:
4     addq (%rax), %rdx
5     addq $16, %rax # Salto un elemento!
6     subq $2, %rcx # Considero solo gli elmenti in posizione dispari
7     js .skip # %rcx puo' diventare negativo senza passare per 0!
8     jnz .loop
9 .skip:
```

Esercizio: calcolo della media di un vettore

Dato un vettore `V` di 32 byte senza segno, memorizzato a partire dalla locazione `0x20a0`, calcolarne la media e memorizzarne il valore sovrascrivendo l'ultima posizione del vettore. Se viene rilevato un overflow, in `V[31]` è posizionato il valore `0xff` (-1)



Esercizio: calcolo della media di un vettore

```
1  .org 0x800
2      .equ array, 0x20a0 # indirizzo base vettore
3      .equ dim, 32 # num elementi array
4      .equ log2dim, 5 # log base 2 di dim
5  .text
6      xorq %rax, %rax # L'accumulatore contiene il risultato parziale
7      xorq %rcx, %rcx # Il contatore viene usato come indice
8    .loop:
9      addb array(, %rcx, 1), %al # somma i-esimo elem. i=0..DIM-1
10     jc .error # se CF = 1 => Overflow
11     addq $1, %rcx
12     cmpq $dim, %rcx
13     jnz .loop  # continua se non siamo alla fine
14     shrb $log2dim, %al # dividi per il numero di elementi (lsrb per
           gli unsigned)
```

Esercizio: calcolo della media di un vettore (2)

```
15      jmp .scrivi
16    .error:
17      movb $0xFF, %al # Sovrascrivi la media con -1
18      movq $dim, %rcx
19    .scrivi:
20      subq $1, %rcx
21      movb %al, array(, %rcx, 1)
22      hlt
```

Esercizio: ordinamento di un vettore

```
1  .org 0x800
2  .data
3          .comm array, 4096 # vettore di byte
4          .comm dim, 4 # intero per dimensione
5
6  .text
7          xorq %rbx, %rbx # %rbx contiene l'indice del minimo
8
9    .ciclo_esterno:
10         leaq 1(%rbx), %rcx # %rcx contiene l'indice dell'elemento da
               confrontare
11         movq %rbx, %rdx # il minimo si trova all'indice del minimo
12
13   .ciclo_interno:
14         movb array(, %rdx, 1), %al # carica il minimo corrente
```

Esercizio: ordinamento di un vettore (2)

```
15          cmpb array(, %rcx, 1), %al # confronta con l'elemento da
                confrontare
16          js .skip # se array[%rdx] < array[%rcx]
17          jz .skip
18          movq %rcx, %rdx # ho trovato un nuovo minimo, salvo l'indice
19      .skip:
20          addq $1, %rcx # passa all'elemento di confronto successimo
21          cmpq dim, %rcx # continua se non sono alla fine
22          jnz .ciclo_interno
23
24          cmpq %rbx, %rdx # verifica se ho trovato un nuovo minimo
25          jz .dontswap
26          movb array(, %rbx, 1), %r8b # scambia i valori
27          movb array(, %rdx, 1), %r9b
28          movb %r9b, array(, %rbx, 1)
29          movb %r8b, array(, %rdx, 1)
```

Esercizio: ordinamento di un vettore (3)

```
30   .dontswap:
31     addq $1, %rbx # l'indice del minimo viene incrementato
32         cmpq dim, %rbx # ripeti se non sono alla fine
33         jnz .ciclo_esterno
34         hlt
```

Esercizio: moltiplicazione tra numeri positivi

- Lo z64 non supporta, in hardware, l'operazione di moltiplicazione
- Se si vuole fornire ad un programma la capacità di esegiure moltiplicazioni, si può implementare la funzionalità tramite software
- Esistono due soluzioni: una naif ed una ottimizzata
- La soluzione naif è semplice, e si basa sull'uguaglianza:

$$a \cdot b = \sum_{i=0}^{b-1} a$$

Moltiplicazione: soluzione naïf

```
1  .org 0x800
2  .data
3      op1: .long 6
4      op2: .long 3
5
6  .text
7      xorl %eax, %eax # Accumulatore del risultato
8      movl op2, %ecx # Registro utilizzato per la terminazione
9    .ciclo:
10     addl op1, %eax # Un gran numero di accessi in memoria!
11     jc .overflow # Se incorriamo in overflow...
12     subl $1, %ecx
13     cmpl $0, %ecx
14     jz .fine
15     jmp .ciclo
```

Moltiplicazione: soluzione naïf (2)

```
16
17   .overflow:
18     movl $-1, %eax # ...impostiamo il risultato a -1
19
20   fine:
21     hlt
```

Moltiplicazione tra numeri positivi: shift-and-add

- Un algoritmo efficiente per la moltiplicazione è lo *shift-and-add*
- Consente di caricare da memoria gli operandi una sola volta
- Ha un costo logaritmico nella dimensione del moltiplicando
- Si basa sull'iterazione di operazioni di shift e somma:
  - Imposta il risultato a 0
  - Ciclo:
    - Shift del moltiplicatore verso sinistra, fino ad allinearlo all'1 meno significativo del moltiplicando
    - Somma il moltiplicatore shiftato al risultato
    - Imposta a 0 il bit meno significativo del moltiplicando impostato a 1
  - Ripeti finché il moltiplicando è uguale a 0

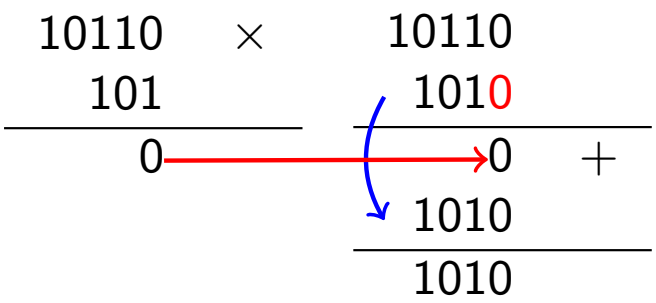
Shift-and-add: un esempio

Eseguiamo la moltiplicazione tra 22 e 5:

$$\begin{array}{rl} 10110 & \times \\ 101 & \\ \hline 0 & \end{array}$$

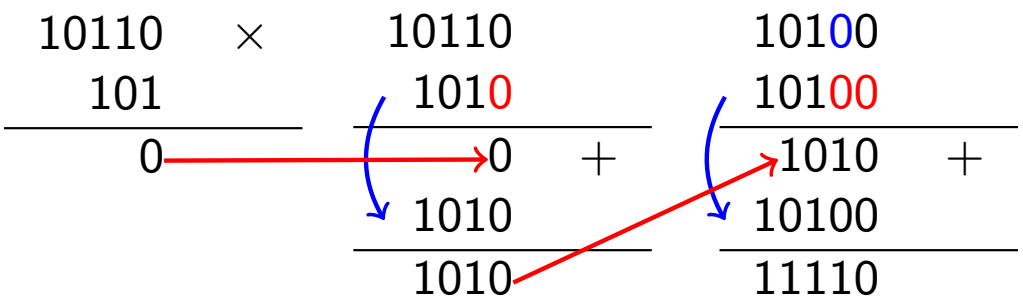
Shift-and-add: un esempio

Eseguiamo la moltiplicazione tra 22 e 5:



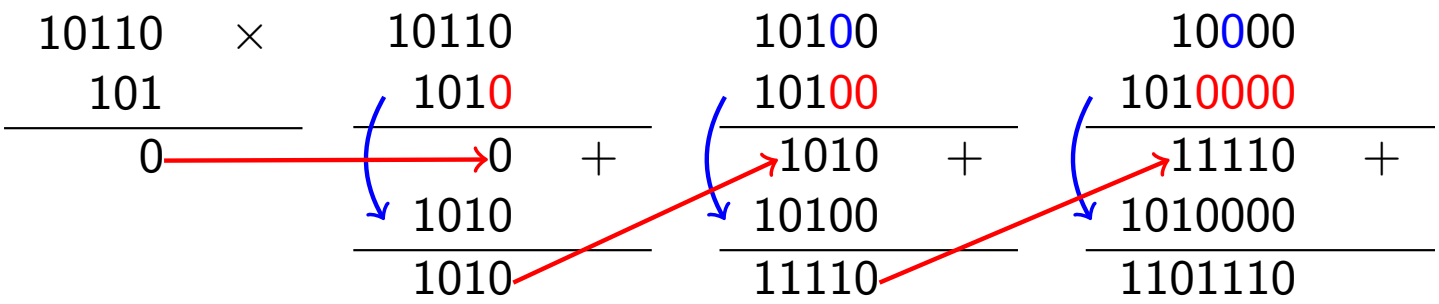
Shift-and-add: un esempio

Eseguiamo la moltiplicazione tra 22 e 5:



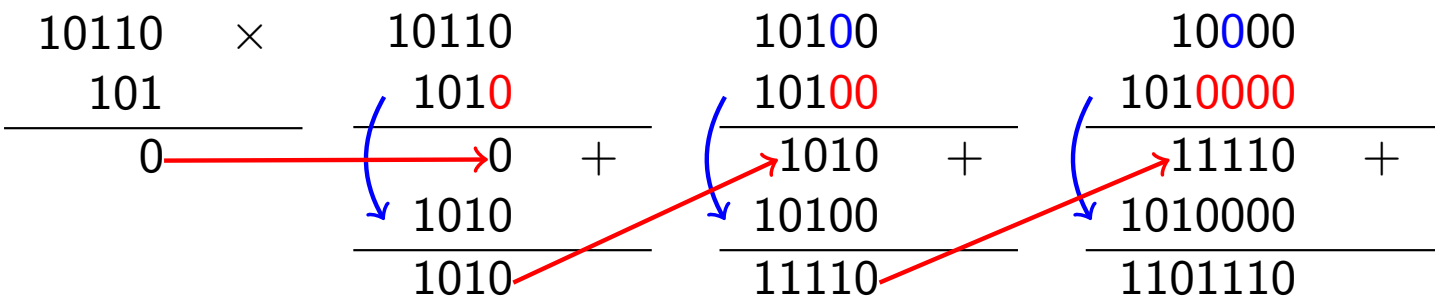
Shift-and-add: un esempio

Eseguiamo la moltiplicazione tra 22 e 5:



Shift-and-add: un esempio

Eseguiamo la moltiplicazione tra 22 e 5:



Il risultato della moltiplicazione è 1101110 (110)

Shift-and-add: adattiamo il codice

- Ogni algoritmo, per essere efficiente, deve essere adattato all'architettura sottostante
- Lo z64 non ha un'istruzione per calcolare il *least significant bit set* (cioè la posizione del bit meno significativo impostato a 1)
  - Sarebbe necessario utilizzare un ciclo con all'interno uno shift e un contatore

Shift-and-add: adattiamo il codice

- Ogni algoritmo, per essere efficiente, deve essere adattato all'architettura sottostante
- Lo z64 non ha un'istruzione per calcolare il *least significant bit set* (cioè la posizione del bit meno significativo impostato a 1)
  - Sarebbe necessario utilizzare un ciclo con all'interno uno shift e un contatore
- Non ridurremmo il costo computazionale, ma lo aumenteremmo!
- Inoltre, il codice risultante sarebbe più lungo e più difficile da correggere in caso di errori!

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	10110	$\texttt{C} = ?$
Moltiplicatore:	101	
Risultato:	0	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	1011	$\texttt{C} = 0$
Moltiplicatore:	101	
Risultato:	0	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	1011	$\texttt{C} = 0$
Moltiplicatore:	1010	
Risultato:	0	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	101	$\texttt{C} = 1$
Moltiplicatore:	1010	
Risultato:	0	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\mathtt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	101	$\mathtt{C} = 1$
Moltiplicatore:	1010	
Risultato:	1010	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	101	$\texttt{C} = 1$
Moltiplicatore:	10100	
Risultato:	1010	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	10	$\texttt{C} = 1$
Moltiplicatore:	10100	
Risultato:	1010	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\mathtt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	10	$\mathtt{C} = 1$
Moltiplicatore:	10100	
Risultato:	11110	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\mathtt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	10	$\mathtt{C} = 1$
Moltiplicatore:	101000	
Risultato:	11110	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\mathtt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	1	$\mathtt{C} = 0$
Moltiplicatore:	101000	
Risultato:	11110	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\mathtt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	1	$\mathtt{C} = 0$
Moltiplicatore:	1010000	
Risultato:	11110	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	0	$\texttt{C} = 1$
Moltiplicatore:	1010000	
Risultato:	11110	

Shift-and-add: adattiamo il codice (2)

- Possiamo utilizzare un doppio shift, sul moltiplicando e sul moltiplicatore:
  - Shift a destra del moltiplicando: se $\mathtt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	0	$\mathtt{C} = 1$
Moltiplicatore:	1010000	
Risultato:	1101110	

Shift-and-add: il codice

```
1  .org 0x800
2  .data
3      op1: .long 3
4      op2: .long 2
5
6  .text
7      movl op1, %ebx # carico il moltiplicando
8      movl op2, %edx # carico il moltiplicatore
9      xorl %eax, %eax # %eax conterra' il risultato
10
11     cmpl $0, %edx
12     jz .fine # Non devo moltiplicare per zero!
13
14   .moltiplica:
15     cmpl $0, %ebx
```

Shift-and-add: il codice (2)

```
16      jz .fine # Quando il moltiplicando e' zero ho finito
17      shrl $1, %ebx
18      jnc .nosomma # Se C = 0 non sommo
19      addl %edx, %eax
20    .nosomma:
21      shll $1, %edx
22      jc .overflow # Controllo l'overflow
23      jmp .moltiplica
24
25    .overflow:
26      movl $-1, %eax # In caso di overflow, imposto il risultato a -1
27
28    .fine:
29      hlt
```

Le Subroutine

- Possono essere considerate in maniera analoga alle funzioni/metodi dei linguaggi di medio/alto livello
- Garantiscono una maggiore semplicità, modularità e riusabilità del codice
- Riducono la dimensione del programma, consentendo un risparmio di memoria utilizzata dal processo
- Velocizzano l'identificazione e la correzione degli errori

Salto a sottoprogramma

- Per eseguire un salto a sottoprogramma si utilizza l'istruzione `call`
- La sintassi è la stessa del salto incondizionato:

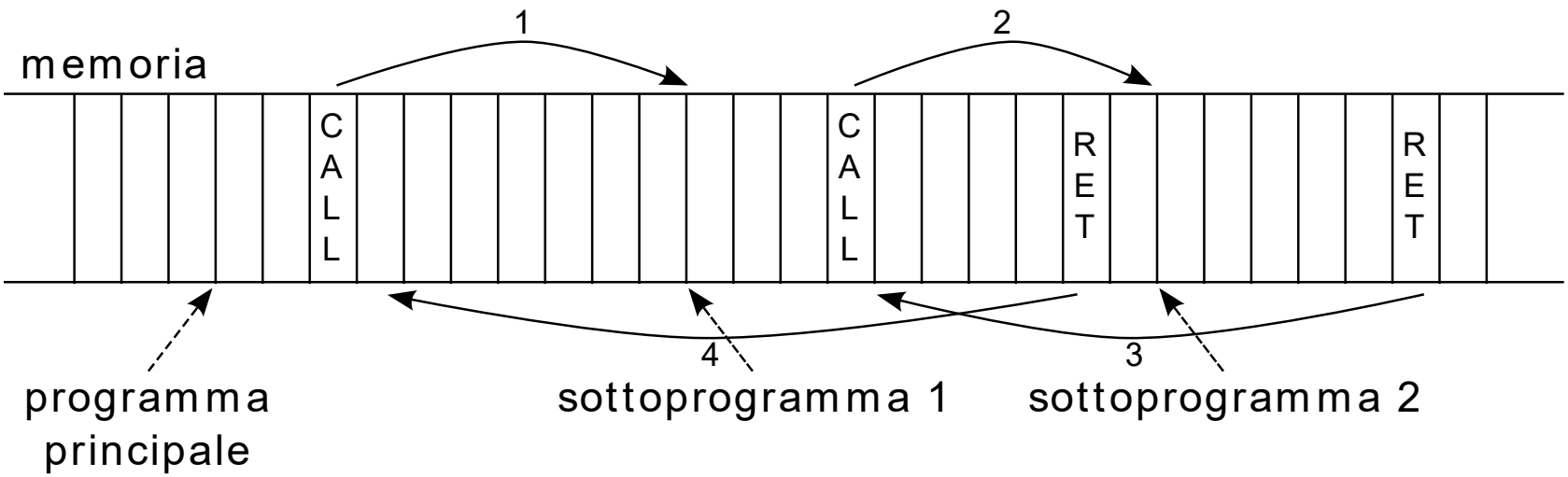
```
call sottoprogramma
```

- L'esecuzione del sottoprogramma termina con l'istruzione *return* (`ret`), che fa "*magicamente*" riprendere il flusso d'esecuzione dall'istruzione successiva alla `call` che aveva attivato il sottoprogramma (ritorno al programma *chiamante)*

Differenza tra `jmp` e `call`

- A differenza della `jmp`, il microcodice della `call` memorizza l'indirizzo dell'istruzione successiva (*indirizzo di ritorno*) prima di aggiornare il valore contenuto nel registro RIP
- L'unico posto in cui può memorizzare quest'informazione è la memoria
- Quest'area di memoria deve essere organizzata in modo tale da gestire correttamente lo scenario in cui un sottoprogramma chiama un altro sottoprogramma (*subroutine annidate*)
- La struttura dati utilizzata per questo scopo si chiama *stack* (pila)

Subroutine annidate



Lo stack

- Gli indirizzi di ritorno vengono memorizzati automaticamente dalle istruzioni `call` nello stack
- È una struttura dati di tipo **LIFO** (*Last-In First-Out*): il primo elemento che può essere prelevato è l'ultimo ad essere stato memorizzato
- Si possono effettuare due operazioni su questa struttura dati:
  - **push**: viene inserito un elemento sulla sommità (*top*) della pila
  - **pop**: viene prelevato l'elemento affiorante (*top element*) dalla pila
- Lo stack può essere manipolato esplicitamente
  - Oltre gli indirizzi di ritorno inseriti dall'istruzione `call` possono essere inseriti/prelevati altri elementi

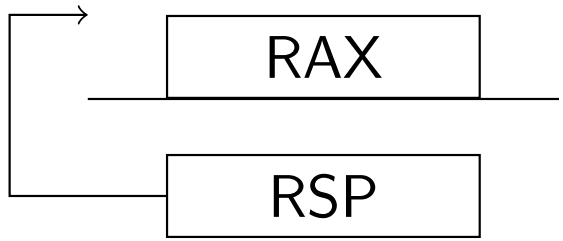
Gestione dello stack

- Lo stack è composto da *quadword* (non si può eseguire una push di un singolo byte)
- La cima dello stack è individuata dall'indirizzo memorizzato in un registro specifico chiamato SP (*Stack Pointer*)
  - Modificare il valore di SP coincide con il perdere il riferimento alla cima dello stack, e quindi a tutto il suo contenuto
- Lo stack "*cresce*" se il valore contenuto in SP diminuisce, "*decresce*" se il valore contenuto in SP cresce
  - Lo stack è posto in fondo alla memoria e cresce "*all'indietro*"

Gestione dello stack

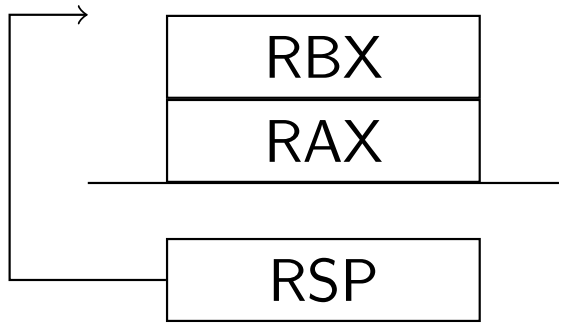
Gestione dello stack

```
pushq %rax
```



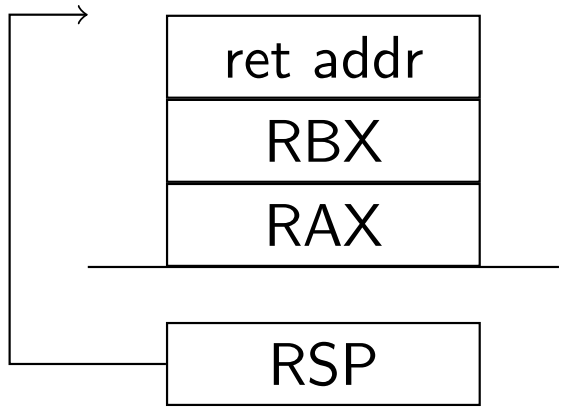
Gestione dello stack

```
pushq %rax
pushq %rbx
```



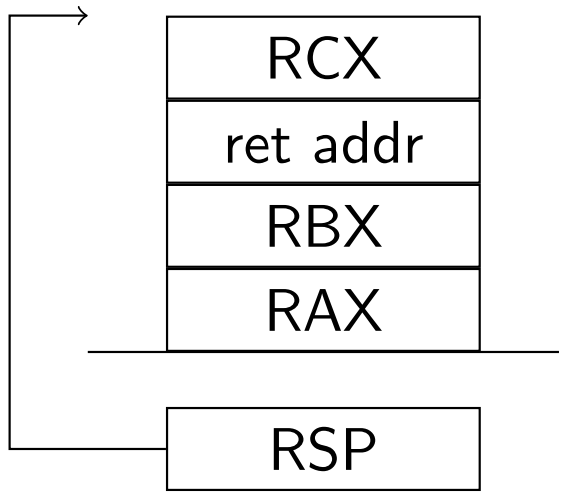
Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
```



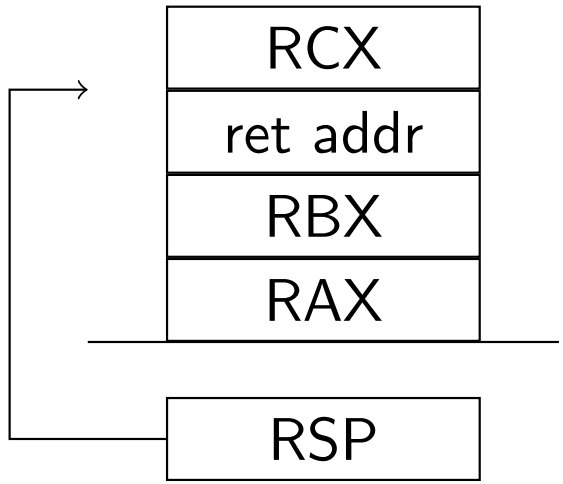
Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
pushq %rcx
```



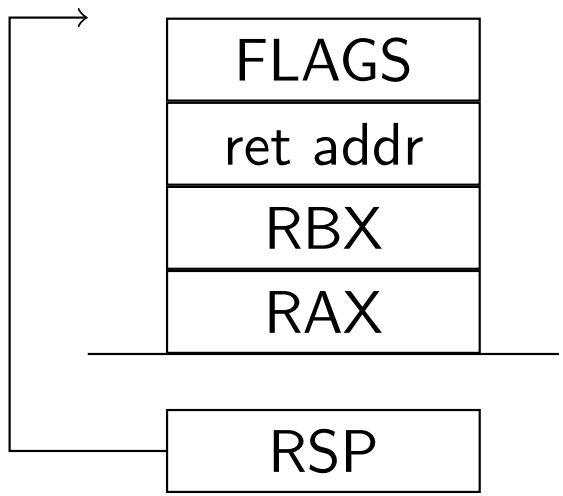
Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
pushq %rcx
popq %rcx
```



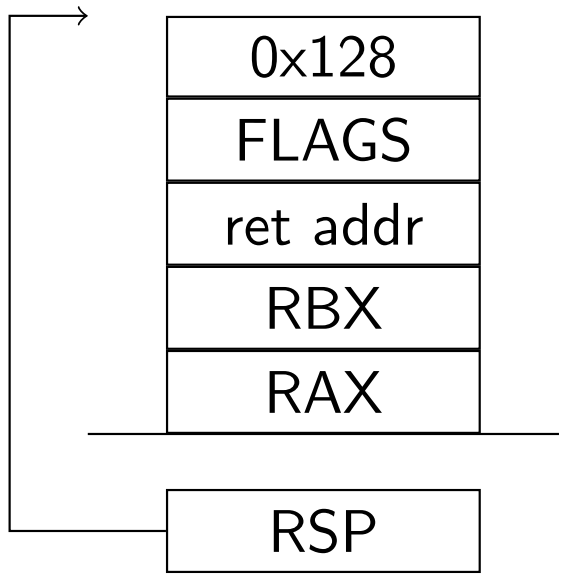
Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
pushq %rcx
popq %rcx
pushfq
```



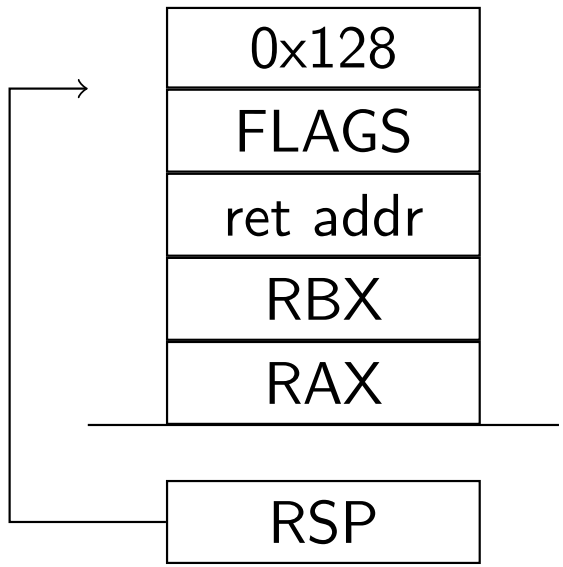
Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
pushq %rcx
popq %rcx
pushfq
pushq $0x128
```



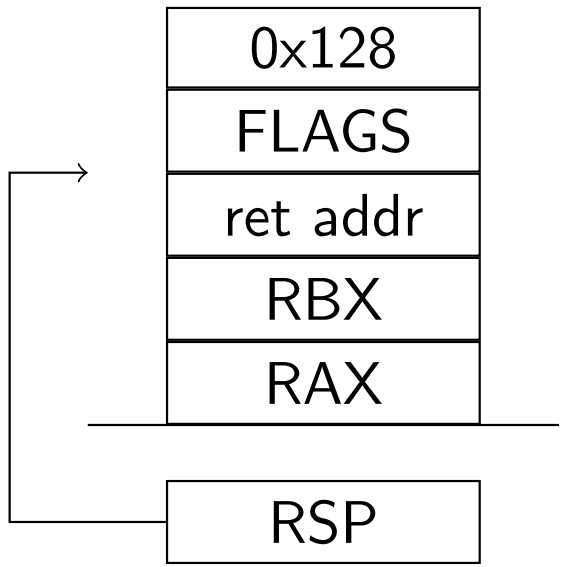
Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
pushq %rcx
popq %rcx
pushfq
pushq $0x128
addq $8, %rsp
```



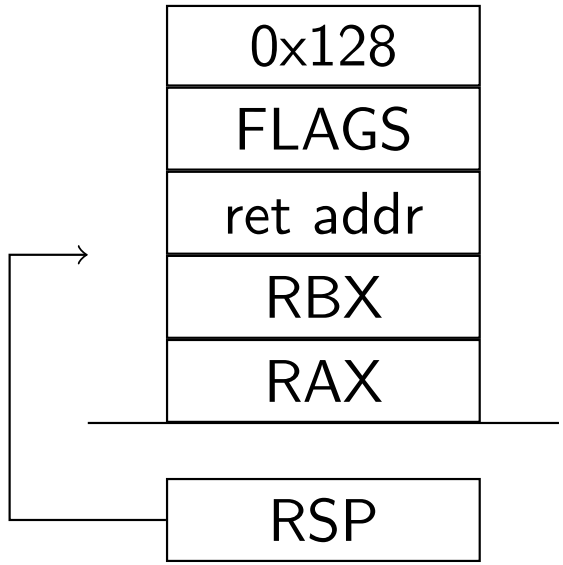
Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
pushq %rcx
popq %rcx
pushfq
pushq $0x128
addq $8, %rsp
popfq
```



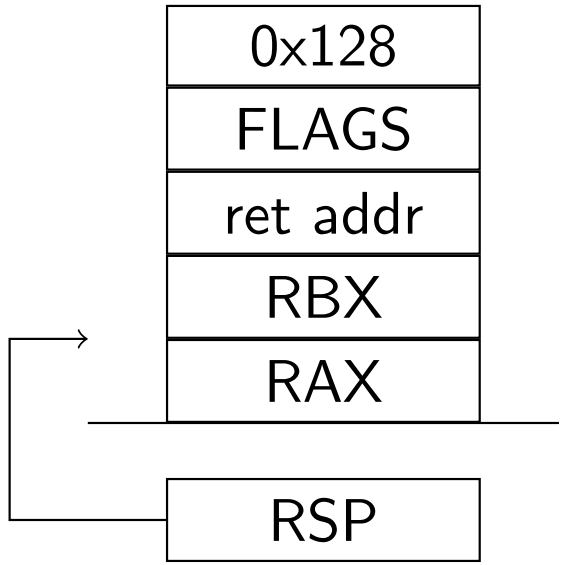
Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
pushq %rcx
popq %rcx
pushfq
pushq $0x128
addq $8, %rsp
popfq
ret
```



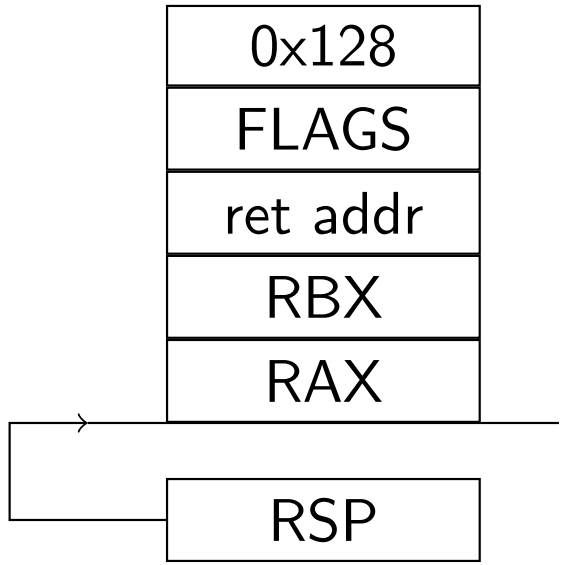
Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
pushq %rcx
popq %rcx
pushfq
pushq $0x128
addq $8, %rsp
popfq
ret
popq %rbx
```



Gestione dello stack

```
pushq %rax
pushq %rbx
call subroutine
pushq %rcx
popq %rcx
pushfq
pushq $0x128
addq $8, %rsp
popfq
ret
popq %rbx
popq %rax
```



La finestra di stack

- Abbiamo visto come creare variabili globali e come definire costanti
- Dove "*vivono*" le variabili locali (o automatiche)?

La finestra di stack

- Abbiamo visto come creare variabili globali e come definire costanti
- Dove "*vivono*" le variabili locali (o automatiche)?
- Una subroutine può utilizzare lo stack per memorizzare variabili locali

```
1 void function() {
2    int x = 128;
3    ...
4    return;
5 }
```

```
1 function:
2     pushq $128
3     ...
4     addq $8, %rsp
5     ret
```

La finestra di stack

- Abbiamo visto come creare variabili globali e come definire costanti
- Dove "*vivono*" le variabili locali (o automatiche)?
- Una subroutine può utilizzare lo stack per memorizzare variabili locali

```
1 void function() {
2    int x = 128;
3    ...
4    return;
5 }
```

```
1 function:
2     pushq $128
3     ...
4     addq $8, %rsp
5     ret
```

- Dopo aver fatto il push di tutte le variabili, si può accedere ad esse tramite RSP:

(%rsp), 8(%rsp), 16(%rsp), ...

La finestra di stack (2)

- Utilizzare RSP per le variabili locali può non essere sempre comodo
- Se lo stack cresce in punti diversi del codice, gli spiazzamenti cambiano
- La definizione della finestra di stack usa il *base pointer* `RBP`
  - Il base pointer non cambia durante l'esecuzione di una subroutine
  - Gli spiazzamenti (negativi!) non cambiano anche se lo stack cresce

```
1 pushq %rbp # Salva il frame pointer corrente
2 movq %rsp, %rbp # Crea un nuovo frame, sull cima dello stack
3 subq $20, %rsp # Alloca 20 byte per le variabili locali su stack
4 movl %eax, -4(%rbp) # Salva %eax nella prima variabile locale
5 movl -8(%rbp), %ebx # Carica %ebx dalla seconda variabile locale
```

La finestra di stack (3)

- Al termine della funzione, si esegue l'istruzione `ret`
- Questa recupera dallo stack l'indirizzo di ritorno
- La finestra di stack va quindi "distrutta" prima di eseguire `ret`:
  - Lo stack pointer deve puntare all'indirizzo di ritorno
  - Il valore precedente di RBP va recuperato dallo stack

```
1 addq $20, %rsp # Invalida lo spazio usato dalle variabili locali
2 popq %rbp # Rimette a posto RBP precedente
```

Convenzioni di chiamata

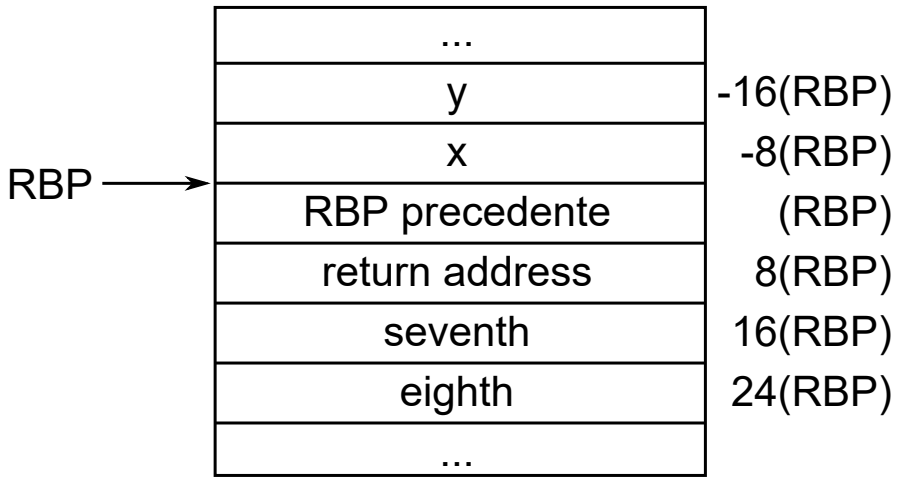
- Affinché una subroutine chiamante possa correttamente dialogare con la subroutine chiamata, occorre mettersi d'accordo su come passare i parametri ed il valore di ritorno
- Le *calling conventions* definiscono, per ogni architettura, come è opportuno passare i parametri
- Le convenzioni principali permettono di passare i parametri tramite:
  - Lo stack
  - I registri
- 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 `call`

System V ABI a 64 bit

- I primi sei parametri di una subroutine vengono passati tramite registri:
  - RDI, RSI, RDX, RCX, R8, R9
  - R10 viene usato al posto di RCX per le system call
- Se una subroutine accetta più di 6 parametri, si utilizza lo stack per quelli aggiuntivi
  - I parametri vengono inseriti sullo stack *in ordine inverso* rispetto alla segnatura della funzione
  - La pulizia dello stack è a carico del chiamante (*caller cleanup*)
- Il valore di ritorno viene memorizzato all'interno di RAX
- Alcuni registri devono essere salvati dalla funzione chiamata (*callee-save*):
  - RBP, RBX, R12–R15

Anatomia dello stack

```
1 void f(int first, int second, int third, int fourth, int fifth, int
      sixth, int seventh, int eighth) {
2         long long x;
3         long long y;
4         ...
5 }
```



Esempio: calcolo del fattoriale

```
1  .org 0x800
2  .data
3     numero: .long 20
4     risultato: .fill 1, 8
5
6  .text
7      movl numero, %edi
8      call fattoriale
9      jnc corretto
10     movl $-1, risultato
11     hlt
12   corretto:
13     movl %eax, risultato
14     hlt
15
```

Esempio: calcolo del fattoriale (2)

```
16
17 # Subroutine per la moltiplicazione
18 # Parametri in %edi e %esi
19 # Valore di ritorno in %eax
20 # Se avviene overflow, ritorna con C=1
21 moltiplica:
22     xorl %eax, %eax
23   .loop:
24     testl %edi, %edi
25     jz .donemult
26     addl %esi, %eax
27     jc .donemult
28     subl $1, %edi
29     jmp .loop
30   .donemult:
31     ret
```

Esempio: calcolo del fattoriale (3)

```
32
33 # Subroutine per il fattoriale
34 # Parametro in %edi
35 # Valore di ritorno in %eax
36 # Se avviene overflow, ritorna con C=1
37 fattoriale:
38     xorl %eax, %eax
39     push %edi
40
41     cmpl $1, %edi # Se %eax = 1, esegui il passo base
42     jnz .passoricorsivo
43
44     movl %edi, %esi
45     jmp .passobase
46   .passoricorsivo:
47     subl $1, %edi
```

Esempio: calcolo del fattoriale (4)

```
48      call fattoriale # Chiama la stessa subroutine
49      jnc .nooverflow
50      pop %edi # Con overflow, tolgo dallo stack i valori salvati
51      jmp .doneFATT
52
53    .nooverflow:
54      movl %eax, %esi
55
56    .passobase:
57      pop %edi
58      call moltiplica # %eax = n * fatt(n-1)
59
60    .doneFATT:
61      ret
```

Istruzioni su stringhe

- Lo z64 offre istruzioni che possono eseguire istruzioni su *stringhe di dati*
  - Copia da memoria a memoria di dati di dimensione arbitraria
  - Imposta aree di memoria di dimensioni arbitrarie ad un valore prestabilito
- Vari registri sono coinvolti da queste istruzioni:
  - 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*

Istruzioni su stringhe (2)

`movs`: move data from string to string

```
1 movq $S, %rsi
2 movq $D, %rdi
3 movq $size/8, %rcx
4 cld
5 movsq
```

`stos`: store string

```
1 movq $0x0, %rax
2 movq $D, %rdi
3 movq $size/8, %rcx
4 cld
5 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

Il "pericolo" della `movs`

```
1 movb $0x800, %rsi
2 movb $0x810, %rdi
3 movb $0x20, %rcx
4 cld
5 movsb
```